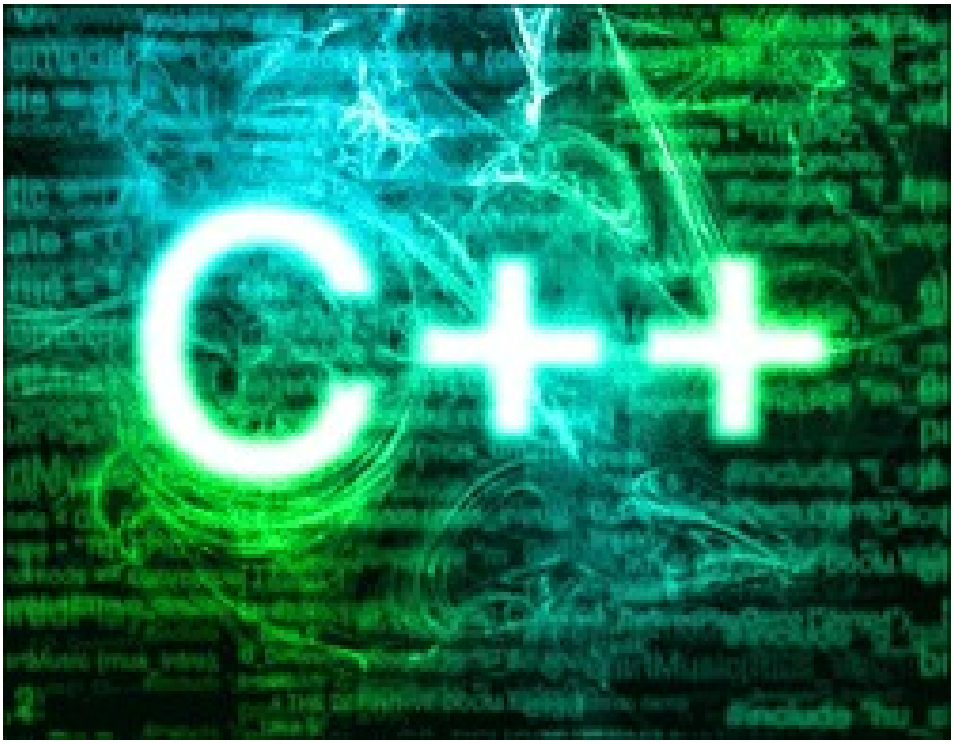
Підручник з предмету: Системне програмування



Зміст

Вступ

Мова C (читається "Сі") створений на початку 70х років, коли Кен Томпсон і Денніс Рітчі з Bell Labs розробляли операційну систему UNDC Спочатку вони створили частина компілятора C, потім використовували се для компіляції іншої частини компілятора C і, нарешті, застосували отриманий в результаті компілятор для компіляції UNIX. Операційна система UNIX спочатку поширювалася у вихідних кодах на C серед університетів і лабораторій, а одержувач міг відкомпілювати вихідний код на C в машинний код за допомогою відповідного компілятора C.

Поширення вихідного коду зробило операційну систему UNIX унікальної; програміст міг змінити операційну систему, а вихідний код міг бути перенесений з одного апаратної платформи на іншу. Сьогодні стандарт POSIX визначає стандартний набір системних викликів UNIX, доступних в C, які повинні бути реалізовані у версіях UNIX, є POSIX-сумісними. З був третьою мовою, який розробили Томсон і Рітчі в процесі створення UNIX; першими двома були, зрозуміло, A і B.

У порівнянні з більш раннім мовою - BCPL, C був поліпшений шляхом додавання типів даних певної довжини. Наприклад, тип даних int міг застосовуватися для створення змінної з певним числом бітів (зазвичай 16), в той час як тип даних long міг використовуватися для створення цілої змінної з великим числом бітів (зазвичай 32). На відміну від інших мов високого рівня, C міг працювати з адресами пам'яті безпосередньо за допомогою покажчиків і посилань. Оскільки C зберіг здатність прямого доступу до апаратного забезпечення, його часто відносять до мов середнього рівня або в жарт називають "мобільним мовою асемблера".

Що стосується граматики і синтаксису, то C є структурним мовою програмування. У той час як багато сучасних програмісти мислять в категоріях класів та об'єктів, програмісти на C думають у категоріях процедур і функцій. В C можна визначити власні абстрактні типи даних, використовуючи ключове слово struct. Аналогічно можна описувати власні цілі типи (перерахування) і давати інші назви існуючих типів даних за допомогою ключового слова typedef. У цьому сенсі C є структурним мовою з зародками об'єктно-орієнтованого програмування. Широке поширення мови C на різних типах комп'ютерів (іноді званих апаратними платформами) призвело, на жаль, до багатьох варіацій мови. Вони були схожі, але несумісні один з одним. Це було серйозною проблемою для розробників програм, які потребували написанні сумісних програм, які можна було б виконувати на декількох платформах. Стало ясно, що необхідна стандартна версія C. У 1983р. ANSI (Американський Національний Комітет Стандартів) сформував технічний комітет X3J11 для створення стандарту мови C (щоб "забезпечити недвозначне і машинно-незалежне визначення мови"). У 1989 стандарт був затверджений. ANSI скооперувався з ISO (Міжнародною Організацією Стандартів), щоб стандартизувати C в міжнародному масштабі; спільний

стандарт був опублікований в 1990 році і названий ANSI / ISO 9899:1990. Цей стандарт вдосконалюється до сих пір і підтримується більшістю фірм розробників компіляторів.

Бьерн Страуструп вивільнив об'єктно-орієнтована потенціал З шляхом перенесення можливостей класів Simula 67 в С. Спочатку нову мову носив ім'я "С з класами" і тільки потім став називатися С + +. Мова С + + досяг популярності, будучи розробленими в Bell Labs, пізніше він був перенесений в інші індустрії та корпорації. Сьогодні це один з найбільш популярних мов програмування у світі. С + + успадковує як хороші, так і погані сторони С.

Бьерн Страуструп: "Я придумав С + +, записав його первісне визначення і виконав першу реалізацію. Я вибрав і сформулював критерії проектування С + +, розробив його основні можливості і відповідав за долю пропозицій з розширення мови в комітеті з стандартизації С + +, - пише автор самого популярного мови програмування . - Мова С + + багатьом зобов'язаний мови С, і мова С залишається підмножиною мови С + + (але в С + + усунені декілька серйозних проломів системи типів С). Я також зберіг кошти С, які є досить низькорівневими, щоб справлятися з самими критичними системними завданнями. Мова С , в свою чергу багатьом зобов'язаний своєму попередникові, BCPL; до речі, стиль коментарів / / був узятий в С + + з BCPL. Іншим основним джерелом натхнення була мова Simula67. Концепція класів (з похідними класами і віртуальними функціями) була запозичена з нього. Засоби перевантаження операторів і можливість приміщення оголошень в будь-якому місці, де може бути записана інструкція, нагадує Algol68.

Назва С + + вигадав Рик Массітті. Назва вказує на еволюційну природу переходу до нього від С. "+ +" - Це операція збільшення в С. Трохи більше коротке ім'я С + є синтаксичної помилкою; крім того, воно вже було використано як ім'я зовсім іншої мови. Знавці семантики С знаходять, що С + + гірше, ніж + + С. Назви D мова не отримав, оскільки він є розширенням С і в ньому не робиться спроб зцілюватися від проблем шляхом викидання різних особливостей ... Спочатку С + + був розроблений, щоб автору і його друзям не доводилося програмувати на асемблері, С або інших сучасних мовах високого рівня. Основним його призначенням було зробити написання хороших програм більш простим і приємним для окремого програміста. Плану розробки С + + на папері ніколи не було; проект, документація та реалізація рухалися одночасно. Зрозуміло, зовнішній інтерфейс С + + був написаний на С + +. Ніколи не існувало "Проекту С + +" і "Комітету з розробки С + +". Тому С + + розвивався і продовжує розвиватися у всіх напрямках, щоб справлятися зі складнощами, з якими стикаються користувачі, а також в процесі дискусій автора з його друзями і колегами ".

У мові С + + повністю підтримуються принципи об'єктно-орієнтованого програмування, включаючи три кити, на яких воно стоїть: інкапсуляцію, успадкування і поліморфізм. Інкапсуляція в С + + підтримується за допомогою створення нестандартних (користувача) типів даних, які

називаються класами. Мова С + + підтримує спадкування. Це означає, що можна оголосити новий тип даних (клас), який є розширенням існуючого.

Хоча мова С + + справедливо називають продовженням С і будь-яка працездатна програма на мові С буде підтримуватися компілятором С + +, при переході від С до С + + був зроблений досить істотний стрибок. Мова С + + вигравав від своєї спорідненості з мовою С протягом багатьох років, оскільки багато програмісти виявили, що для того, щоб повною мірою скористатися перевагами мови С + +, їм потрібно відмовитися від деяких своїх колишніх знань і придбати нові, а саме : вивчити новий спосіб концептуальності і рішення проблем програмування. Перед тим як починати освоювати С + +, Страуструп і більшість інших програмістів, іспользущіх С + + вважають вивчення мови З необов'язковим.

С + + в даний час вважається пануючою мовою, використовуваним для розробки комерційних продуктів, 90% ігор пишуться на С + + з прменення DirectX.

1. Модуль 1.
Загальніа характеристика мови.Найпростіші програми

1.1 Історія виникнення мови С.Переваги мови С.

Завдяки чому склався такий статус мови С? Історично ця мова невіддільний від операційної системи Unix, яка в наші дні переживає своє друге народження. 60-і роки були епохою становлення операційних систем та мов програмування високого рівня. У той період для кожного типу комп'ютерів незалежно розроблялися ОС і компілятори, а нерідко навіть свої мови програмування (згадаємо, наприклад, PL / I). У той же час, спільність виникають при цьому проблем вже стала очевидною. Відповіддю на усвідомлення цієї спільності стала спроба створити універсальну мобільну операційну систему, а для цього знадобився не менш універсальний і мобільний мову програмування. Таким мовою стала С, а Unix стала першою ОС, практично повністю написаною на мові високого рівня.

Тісний зв'язок з Unix дала мови З такою полігон для обкатки, якого не було в той час ні у одного іншої мови. Завдання системного програмування по праву вважалися в той час найскладнішими в галузі. У більшості своїй вони були настільки машинно-залежними, що багато хто взагалі не мислили їх рішення інакше, ніж на асемблері. Мови високого рівня призначалися для прикладного програмування і лише дуже обмежено реалізовували функції, необхідні для системних робіт, причому найчастіше тільки для певного типу машин.

Мова С з самого початку створювався так, щоб на ньому можна було писати системні завдання. Творці С не стали розробляти абстрактну модель виконавця мови, а просто реалізували в ньому ті можливості, в яких найбільше потребували практиці системного програмування. Це в першу чергу були кошти безпосередньої роботи з пам'яттю, структурні конструкції управління і модульна організація програми. І по суті більш нічого в мову включено не було. Все інше було віднесено до бібліотеки часу виконання. Тому недоброзичливці інший раз відзиваються про мову З як про структурний асемблері. Але що б вони не базікали, підхід виявився дуже вдалим. Завдяки йому був досягнутий новий рівень по співвідношенню простоти і можливостей мови.

Є, втім, ще один фактор, який визначив успіх мови. Творці дуже вміло розділили в ньому машинно-залежні та незалежні властивості. Завдяки цьому більшість програм вдається писати універсально - їх працездатність не залежить від архітектури процесора і пам'яті. Нечисленні ж апаратно-залежні частини коду можна локалізувати в окремих модулях. А користуючись препроцесором, можна створювати такі модулі, які при компіляції на різних платформах будуть породжувати відповідний машинно-залежний код.

Багато суперечок викликав синтаксис мови С. Застосовані в ньому прийоми скорочення запису при непомірному використанні можуть зробити програму абсолютно нечитаною. Але, як говорив Дейкстра, - кошти не винні в тому, що їх безграмотно використовують. Насправді ж, запропоновані в С скорочення синтаксису відповідають найбільш часто зустрічається на практиці стереотипним ситуацій. Якщо вважати скорочення ідіомами для виразного і компактного представлення таких ситуацій, то користь від них стає безумовною і очевидною.

Отже, С виник як універсальна мова системного програмування. Але він не залишився в цих рамках. До кінця 80-х років мова С, відтіснивши Fortran з позиції лідера, завоював масову популярність серед програмістів у всьому світі і став використовуватися в самих різних прикладних задачах. Чималу роль тут зіграло розповсюдження Unix (а значить і С) в університетському середовищі, де проходило підготовку нове покоління програмістів.

Як і всі мови, С поступово вдосконалювався, але більшість удосконалень не носило радикального характеру. Найбільш істотним з них, мабуть, слід вважати введення суворої специфікації типів функцій, яка значно підвищила надійність міжмодульних взаємодії на С. Все такі удосконалення були в 1989 році закріплені в стандарті ANSI який і понині визначає мову С.

Але якщо все так безхмарно, то чому ж ще продовжують використовуватися всі інші мови, що підтримує їх існування? Ахіллесовою п'ятою мови З стало те, що він виявився занадто низькорівневим для тих завдань, які поставили на порядок денний 90-і роки. Причому у цієї проблеми є два аспекти. З одного боку, в мову були вбудовані дуже низькорівневі засоби - перш за все це робота з пам'яттю і адресна арифметика. Недарма зміна розрядності процесорів дуже болісно відбивається на багатьох С-програмах. З іншого боку, в С бракує коштів високорівневих - абстрактних типів даних і об'єктів, поліморфізму, обробки виключень. Як наслідок, в програмах на С техніка реалізації завдання часто домінує над її змістовною стороною.

Перші спроби виправити ці недоліки стали вживатися ще на початку 80-х років. Уже тоді Бьерн Страуструп в AT & T Bell Labs став розробляти розширення мови С під умовною назвою. Стиль ведення розробки цілком відповідав духу, в якому створювався і сама мова С, - в нього вводилися ті чи інші можливості з метою зробити більш зручною роботу конкретних людей і груп. Перший комерційний транслятор нової мови, що отримав назву С + + з'явився в 1983 році. Він представляв собою препроцесор, що транслював програму в код на С. Однак фактичним народженням мови можна вважати вихід в 1985 році книги Страуструпа. Саме з цього моменту С + + починає набирати всесвітню популярність.

Головне нововведення С + + - механізм класів, що дає можливість визначати і використовувати нові типи даних. Програміст описує внутрішнє представлення об'єкту класу і набір функцій-методів для доступу до цього подання. Однією із заповітних цілей при створенні С + + було прагнення збільшити відсоток повторного використання вже написаного коду.

Концепція класів пропонувала для цього механізм наслідування. Спадкування дозволяє створювати нові (похідні) класи з розширеним уявленням і модифікованими методами, не зачіпаючи при цьому скомпільований код вихідних (базових) класів. Разом з тим спадкоємство забезпечує один з механізмів реалізації поліморфізму - базової концепції об'єктно-орієнтованого програмування, згідно з якою, для виконання однотипної обробки різних типів даних може використовуватися один і той же код. Власне, поліморфізм - теж один із методів забезпечення повторного використання коду.

Введення класів не вичерпує всіх новацій мови C + +. У ньому реалізовані повноцінний механізм структурної обробки виключень, відсутність якого в C значно ускладнювало написання надійних програм, механізм шаблонів - витончений механізм макрогенерації, глибоко вбудований в мову, що відкриває ще один шлях до повторної використовуваної коди, і багато іншого.

Таким чином, генеральна лінія розвитку мови була направлена на розширення його можливостей шляхом введення нових високорівневих конструкцій при збереженні скільки можна повної сумісності з ANSI C. Звичайно, боротьба за підвищення рівня мови йшла і на другому фронті - ті ж класи дозволяють при грамотному підході ховати низькорівневі операції, так що програміст фактично перестає безпосередньо працювати з пам'яттю і системно-залежними сутностями. Проте мова не містить механізмів, які змушують розробника правильно структурувати програму, а автори не випустили ніяких систематичних рекомендацій по використанню його досить витончених конструкцій. Не подбали вони своєчасно і про створення стандартної бібліотеки класів, що реалізує найбільш часто зустрічаються структури даних.

Все це привело до того, що багато розробників змушені були самі досліджувати лабіринти мовної семантики і самостійно відшукувати успішно працюючі ідіоми. Так, наприклад, на першому етапі розвитку мови багато творців бібліотек класів прагнули побудувати єдину ієрархію класів із загальним базовим класом Object. Ця ідея була запозичена з Smalltalk - одного з найбільш відомих об'єктно-орієнтованих мов. Проте вона виявилася абсолютно нежиттєздатною в C + + - ретельно продумані ієрархії бібліотек класів виявлялися негнучкими, а робота класів - неочевидною. Для того щоб бібліотеками класів можна було користуватися, їх доводилося поставляти в початкових текстах.

Поява темплетних класів і зовсім спростувало цей напрям розвитку. Спадкуванням стали користуватися тільки в тих випадках, коли було потрібно породження спеціалізованої версії наявного класу. Бібліотеки стали складатися з окремих класів і невеликих незв'язаних один з одним ієрархій. Однак на цьому шляху стало знижуватися повторне використання коду, так як в C + + неможливо поліморфну використання класів з незалежних ієрархій. Повсюдне ж застосування темплетов веде до неприпустимого

зростання обсягу скомпільованого коду - не будемо забувати, темплети реалізуються методами макрогенерації.

Один із найважчих недоліків С + +, успадкований ним від синтаксису С, полягає в доступності компілятору опису внутрішньої структури всіх використаних класів. Як наслідок, зміна внутрішньої структури представлення якого-небудь бібліотечного класу призводить до необхідності перекомпіляції всіх програм, де ця бібліотека використовується. Це сильно обмежує розробників бібліотек в частині їх модернізації, адже, випускаючи нову версію, вони повинні зберігати двійкову сумісність з попередньою. Саме ця проблема змушує багатьох фахівців вважати, що С + + непридатний для ведення великих і надвеликих проектів.

І все ж, незважаючи на перераховані недоліки і навіть на неготовність стандарту мови (це після п'ятнадцяти з гаком років використання!), С + + залишається одним з найбільш популярних мов програмування. Його сила насамперед у практично повній сумісності з мовою С. Завдяки цьому програмістам С + + доступні всі напрацювання, виконані на С. При цьому С + + навіть без використання класів привносить в С ряд настільки важливих додаткових можливостей і зручностей, що багато хто користується їм просто як поліпшеним З .

Що стосується об'єктної моделі С + +, то поки ваша програма не стала дуже великою (сотні тисяч рядків), нею цілком можна користуватися. Намітилася останнім часом тенденція переходу до компонентного програмного забезпечення тільки підсилює позиції С + +. При розробці окремо взятих компонентів недоліки С + + ще не виявляються, а зв'язування компонентів в працюючу систему проводиться вже не на рівні мови, а на рівні операційної системи.

У світлі всього сказаного перспективи С + + не виглядають похмурими. Хоча й монополія на ринку мов програмування йому не світить. Мабуть, з упевненістю можна стверджувати лише те, що ще однією модернізації-розширення ця мова не переживе. Недарма, коли з'явилася Java, на неї звернули таку пильну увагу. Мова, близький по синтаксису до С + +, а значить, що здається знайомим багатьом програмістам, був позбавлений від найбільш кричущих недоліків С + +, успадкованих ним з 70-х років. Проте не схоже, щоб Java справлялася з покладеної на неї деякими роллю.

1.2 Елементи мови С. Структура програми мови С. Стандарти мови

Безліч символів використовуваних у мові СІ можна розділити на п'ять груп.

1.Символи, які використовуються для утворення ключових слів і ідентифікаторів.У цю групу входять великі та малі літери англійського алфавіту, а також символ підкреслення. Слід зазначити, що

однакові прописні і малі літери вважаються різними символами, тому що мають різні коди.

Прописні букви латинського алфавіту:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Рядкові букви латинського алфавіту:

a b c d e f g h i j k l m n o p q r s t u v w x y z

Символ підкреслення:

_

2.Група великих і малих літер російського алфавіту і арабські цифри.

Прописні букви російського алфавіту:

А Б В Г Д Е Ж З И К Л М Н О П Р С Т У Ф Х Ц Ч Ш Щ Ы Ь Э Ю Я

Малі літери російського алфавіту:

а б в г д е ж з и к л м н о п р с т у ф х ц ч ш щ ъ ы ь э ю

Арабські цифри:

0 1 2 3 4 5 6 7 8 9

3.Знаки нумерації та спеціальні символи . Ці символи використовуються з одного боку для організації процесу обчислень, а з іншого - для передачі компілятору певного набору інструкцій.

,	Кома	)	кругла дужка права
.	Крапка	(	кругла дужка ліва
;	Крапка з комою	}	фігурна дужка права
:	Двокрапка	{	фігурна дужка ліва
?	Знак питання	<	менше
'	Апостроф	>	більше
!	Знак оклику	[	квадратна дужка
|	Вертикальна риса	]	квадратна дужка
/	Дробова риса	#	номер
\	Зворотній риса	%	відсоток
~	Тильда	&	амперсанд
*	Зірочка	^	логічне не
+	Плюс	=	рівно
-	Мінус	"	лапки

4.Керуючі і роздільників.До тієї групи символів відносяться: пробіл, символи табуляції, перекладу рядка, повернення каретки, нова сторінка і новий рядок. Ці символи відокремлюють один від одного об'єкти, визначені користувачем, до яких відносяться константи і ідентифікатори. Послідовність розділових символів розглядається компілятором як один символ (послідовність пробілів).

5.Крім виділених груп символів в мові СІ широко використовуються так звані, керуючі послідовності, тобто спеціальні символьні комбінації, що використовуються у функціях введення і виведення інформації. Керуюча послідовність будується на основі використання зворотного дробової риски (\) (обов'язковий перший символ) і комбінацією латинських букв і цифр.

Керуюча послідовність	Найменування	Шеснадцатерічная заміна
\ a	Дзвінок	007
\ b	Повернення на крок	008
\ t	Горизонтальна табуляція	009
\ n	Перехід на новий рядок	00A
\ v	Вертикальна табуляція	00B
\ r	Повернення каретки	00C
\ f	Переклад формату	00D
\ "	Лапки	022
\ '	Апостроф	027
\ 0	Нуль-символ	000
\ \	Зворотній дробова риса	05C
\ ddd	Символ набору кодів ПЕОМ в вісімковому поданні	
\xddd	Символ набору кодів ПЕОМ в шістнадцятковому представленні	

Послідовності виду \ ddd і \ xddd (тут d позначає цифру) дозволяє представити символ із набору кодів ПЕОМ як послідовність вісімкових або шістнадцятиричних цифр відповідно. Наприклад символ повернення каретки може бути представлений різними способами:
\ R - загальна керуюча послідовність,
\ 015 - вісімкова керуюча послідовність,
\ X00D - шістнадцяткова керуюча послідовність.

Слід зазначити, що в строкових константах завжди обов'язково задавати всі три цифри в керуючої послідовності. Наприклад окрему керуючу послідовність \ n (перехід на новий рядок) можна представити як \ 010 або \ xA, але в строкових константах необхідно задавати всі три цифри, в іншому випадку символ або символи наступні за керуючої послідовністю будуть розглядатися як її відсутня частина. Наприклад:
"ABCDE \ x009FGH" дана строкою команда буде надрукована з використанням певних функцій мови CI, як два слова ABCDE FGH, розділені 8-ю пробілами, в цьому випадку якщо вказати неповну керуючу рядок "ABCDE \ x09FGH", то на друку з'явиться ABCDE = | = GH, так як компілятор сприйме послідовність \ x09F як символ "= + =".

Відзначимо той факт, що, якщо зворотна подрібнена риса передує символу не є керуючою послідовністю (тобто не включеному в табл.4) і не є цифрою, то ця риса ігнорується, а сам символ представляється як літеральний. Наприклад:
символ \ h представляється символом h в строкової або символьної константи.

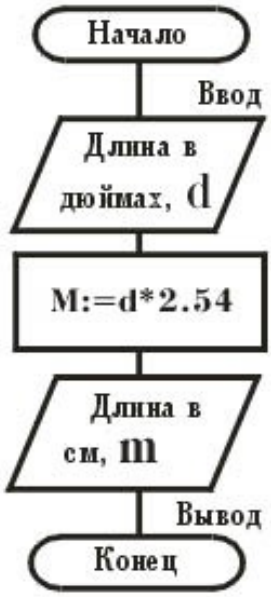
Крім визначення керуючої послідовності, символ зворотної дробової риски (\) використовується також як символ продовження. Якщо за (\) слід (\ n), то обидва символи ігноруються, а наступна рядок є продовженням

попередньої. Ця властивість може бути використано для запису довгих рядків.

1.3 Практична робота №1:"Опис середовища Visual C+ +.Програмування лінійних алгоритмів"

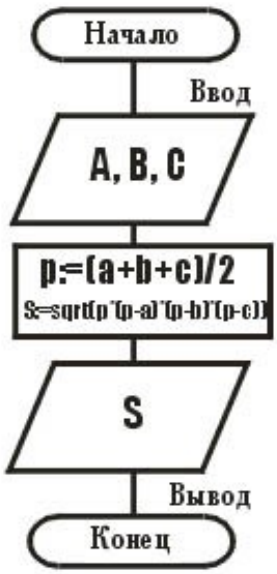
Розглянемо приклад програми лінійного алгоритму.

Приклад 1. Довжина відрізка задана в дюймах (1 дюйм = 2,54 см). Перевести значення довжини в метричну систему, тобто виразити її в сантиметрах.




```
program Example_dlina;
var d,m: real;{ оголошення змінних }
begin
write('Длина в дюймах:');     { висновок пояснення }
readln(d);  { введення вихідних даних }
m:=d*2.54;  { обчислення }
 writeln  (' Довжина в сантиметрах ',m);     { виведення результату }
end.
```

Приклад 2. Трикутник заданий величинами своїх сторін. Знайти його площу.




```
program Example1_S;
 var a,b,c,p,s : real;
 begin
 write( 'A=' ) ;
 readln (a);
 write ( 'B=');
 readln(b);
 write('C=');
 readln (c);
 p:=(a+b+c)/2;
 s:=sqrt(p*(p-a)*(p-b)*(p-c)
 writeln('S=',s:6:2);
 end.
```

Багато складні на перший вид завдання вирішуються простими обчисленнями, заснованими на простих логічних міркуваннях або готових математичних формулах. Хорошим прикладом є наступна задача:

Власник автомобіля придбав новий карбюратор, який економить 50% палива, нову систему запалювання, яка економить 30% палива, і нові поршневі кільця, які економлять 20% палива. Чи правда, що його автомобіль тепер зможе обходитися зовсім без палива? Знайти фактичну економію палива.

Відповідь на перше питання виникає природно: ні, в будь-якому випадку якась кількість бензину машина буде витрачати. Але яке? Просте підсумовування або знаходження середнього значення вихідних даних дасть нам невірний результат. Практично той же підхід використовується і при вирішенні більшості завдань «на час», на переклад величин у різних системах заходів, на економію операцій при виконанні деяких дій.

Тут добрими помічниками є функції залишку від ділення і цілого ділення, а також математичні хитрощі.

Приклад 3. Вирішити задачу обміну значеннями двох змінних, не використовуючи додаткових змінних і припускаючи, що значеннями цілих змінних можуть бути довільні цілі числа.

Позначимо початкові значення змінних a і b через a0 і b0. Тоді математично очевидно наступне:

```
a=a+b {a=a0+b0,    b=b0}
b=a-b  {a=a0+b0,    b=a0}
a=a-b  {a=b0,         b=a0}
```

(Блок-схему алгоритму скласти самостійно)

Реалізувавши ці обчислення на мові програмування, отримаємо рішення нашої задачі:

```
program Example_obmen;
 var a,b:integer;
 begin
 write('Введите A: ');
 readln (a);
 write('Введите  B : ');
 readln(b);
 a:=a+b;
 b:=a-b;
 a:=a-b;
 writeln('A = ',a);
 writeln('B = ',b);
 readln;
 end.
```

Приклад 4. Годинна стрілка утворює кут <j з променем, що проходить через центр циферблата і через точку, відповідну 12 години. За значенням кута визначити час, показуване годинами.

Циферблат розбитий на 12 рівних частин (годин). Циферблат, як коло, має 360 градусів. Значить, один повний годину дорівнює 30 градусам.

Тобто кількість повних годин визначається виразом j div 30. За один повний годину, тобто за 30 градусів ходу годинникової стрілки, хвилинна робить повний оборот, тобто 360 градусів. Значить, 1 хвилина дорівнює 2 градусам і кількість хвилин в поточному годині дорівнюватиме:

(j mod 30)*2

Точність визначення часу за кутом годинникової стрілки дорівнює ± 1 хвилина.

(Блок-схему алгоритму скласти самостійно)

```
program Example_ugol;
 var fi,chas,min: integer;
 begin
 write ('Введіть кут ' ) ;
 readln (fi) ;
 chas:=fi div 30;
 min:=(fi mod 30)*2;
 write('Поточний час',godina,'  година',hvl,'хвл  ');
 end.
```

1.4 Змінні, константи, типи даних

Типи даних

Програми оперують з різними даними, які можуть бути простими і структурованими. Прості дані - це цілі і речові числа, символи і покажчики (адреси об'єктів в пам'яті). Цілі числа не мають, а речові мають дробову частину. Структуровані дані - це масиви та структури, вони будуть розглянуті нижче.

У мові розрізняють поняття "тип даних" і "модифікатор типу". Тип даних - це, наприклад, цілий, а модифікатор - зі знаком або без знаку. Ціле із знаком буде мати як позитивні, так і негативні значення, а ціле без знака - тільки позитивні значення. У мові Сі можна виділити п'ять базових типів, які задаються наступними ключовими словами:

char - символьний;
int - цілий;
float - речовий;
double - речовинний подвійної точності;
void - не має значення.

Дамо їм коротку характеристику:

1.Змінна типу char має розмір 1 байт, її значеннями є різні символи з кодової таблиці, наприклад: 'ф', ':', 'j' (при запису в програмі вони полягають в одинарні лапки).

2.Розмір змінної типу int в стандарті мови Сі не визначений. У більшості систем програмування розмір змінної типу int відповідає розміру цілого

машинного слова. Наприклад, в компіляторах для 16-розрядних процесорів змінна типу int має розмір 2 байта. В цьому випадку знакові значення цієї змінної можуть лежати в діапазоні від -32768 до 32767.

3.Ключове слово float дозволяє визначити змінні дійсного типу. Їх значення мають дробову частину, відокремлювану крапкою, наприклад: -5.6, 31.28 і т.п. Речові числа можуть бути записані також у формі з плаваючою крапкою, наприклад:-1.09e +4. Число перед символом "e" називається мантиси, а після "e" - порядком. Змінна типу float займає в пам'яті 32 біта. Вона може приймати значення в діапазоні від 3.4e-38 до 3.4e +38.

4.Ключове слово double дозволяє визначити речову змінну подвійної точності. Вона займає в пам'яті в два рази більше місця, ніж змінна типу float (тобто її розмір 64 біта). Змінна типу double може приймати значення в діапазоні від 1.7e-308 до 1.7e +308.

5.Ключове слово void (не має значення) використовується для нейтралізації значення об'єкта, наприклад, для оголошення функції, не повертає ніяких значень.

Об'єкт деякого базового типу може бути модифікований. З цією метою використовуються спеціальні ключові слова, які називають модифікаторами. У стандарті ANSI мови Сі є наступні модифікатори типу:

unsigned
signed
short
long

Модифікатори записуються перед специфікаторами типу, наприклад: unsigned char. Якщо після модифікатора опущений специфікатор, то компілятор припускає, що цим специфікатором є int. Таким чином, наступні рядки:

```
long a;
long int a;
```

є ідентичними і визначають об'єкт a як довгий цілий. Табл. 1 ілюструє можливі поєднання модифікаторів (unsigned, signed, short, long) зі специфікаторами (char, int, float і double), а також показує розмір і діапазон значень об'єкта (для 16-розрядних компіляторів).

Тип	Розмір в байтах (бітах)	Інтервал зміни
char	1 (8)	від -128 до 127
unsigned char	1 (8)	від 0 до 255
signed char	1 (8)	від -128 до 127
int	2 (16)	від -32768 до 32767
unsigned int	2 (16)	від 0 до 65535

signed int	2 (16)	від -32768 до 32767
short int	2 (16)	від -32768 до 32767
unsigned short int	2 (16)	від 0 до 65535
signed short int	2 (16)	від -32768 до 32767
long int	4 (32)	від -2147483648 до 2147483647
unsigned long int	4 (32)	від 0 до 4294967295
signed long int	4 (32)	від -2147483648 до 2147483647
float	4 (32)	від 3.4E-38 до 3.4E+38
double	8 (64)	від 1.7E-308 до 1.7E+308
long double	10 (80)	від 3.4E-4932 до 3.4E+4932

Змінні і константи

Всі змінні до їх використання повинні бути визначені (оголошені). При цьому задається тип, а потім йде список з однієї або більше змінних цього типу, розділених комами. наприклад:

```
int a, b, c;
char x, y;
```

У мові розрізняють поняття оголошення змінної і її визначення. Оголошення встановлює властивості об'єкта: його тип (наприклад, цілий), розмір (наприклад, 4 байта) і т.д. Визначення разом з цим викликає виділення пам'яті (в наведеному прикладі дано визначення змінних).

Змінні можна розділяти по рядках довільним чином, наприклад:

```
float a;
float b;
```

Змінні в мові Сі можуть бути ініційовані при їх визначенні:

```
int a = 25, h = 6;
char g = 'Q', k = 'm';
float r = 1.89;
long double n = r*123;
```

З'ясуємо тепер, де в тексті програми визначаються дані. У мові можливі глобальні та локальні об'єкти. Перші визначаються поза функцій і, отже, доступні для будь-якої з них. Локальні об'єкти по відношенню до функцій є внутрішніми. Вони починають існувати, при вході у функцію і знищуються після виходу з неї. Нижче показана структура програми на Сі і можливі місця в програмі, де визначаються глобальні та локальні об'єкти.

```
int a;                    /* Визначення глобальної змінної */

int function (int b, char c);   /* Оголошення функції (тобто опис її заголовка)*/

void main (void)
{              //Тіло програми
    int d, e;          // Визначення локальних змінних
    float f;           // Визначення локальної змінної
      ...
}
int function (int b, char c) /* Визначення функції і формальних
                        параметрів (по суті – локальних змінних) b и c */
{              //Тело функции
    char g;            // Визначення локальної змінної
      ...
}
```

Зазначимо, що виконання програми завжди починається з виклику функції main (), яка містить тіло програми. Тіло програми, як і тіло будь-якої іншої функції, поміщається між відкриває і закриває фігурними дужками.

У мові Сі всі визначення мають слідувати перед операторами, складовими тіло функції. У мові Сі + + це обмеження знято і визначення можуть перебувати в будь-якому місці програми. Якщо вони зроблені у функції, то відповідні об'єкти будуть локальними, а якщо поза функцій, то глобальними.

Поряд із змінними в мові існують такі види констант:

-речові, наприклад 123.456, 5.61e-4. Вони можуть забезпечуватися суфіксом F (або f), наприклад 123.456F, 5.61e-4f;

-цілі, наприклад 125;

-короткі цілі, в кінці запису яких додається буква (суфікс) H (або h), наприклад 275h, 344H;

-довгі цілі, в кінці запису яких додається буква (суфікс) L (або l), наприклад 361327L;

-беззнакові, в кінці запису яких додається буква U (або u), наприклад 62125U; вісімкові, в яких перед першою значущою цифрою записується нуль (0), наприклад 071;

-шістнадцяткові, в яких перед першою значущою цифрою записується пара символів нуль-ікс (0x), наприклад 0x5F;

-символьні - єдиний символ, укладений в одинарні лапки, наприклад "Про ', '2', '.' і т.п. Символи, які не мають графічного представлення, можна записувати, використовуючи спеціальні комбінації, наприклад \ n (код 10), \ 0 (код 0). Ці комбінації виглядають як два символи, хоча фактично це один символ. Так само можна уявити будь двійковий образ одного байта: '\ NNN', де NNN - від однієї до трьох вісімкових цифр. Допускається і шістнадцяткове завдання кодів символів, яке представляється у вигляді: '\ x2B', '\ x36' і т.п.;

-рядкові - послідовність з нуля символів і більше, укладена в подвійні лапки, наприклад: "Це строкою константа". Лапки не входять в рядок, а лише обмежують її. Рядок являє собою масив з перерахованих елементів, в кінці якого поміщається байт з символом '\ 0'. Таким чином, число байтів, необхідних для зберігання рядки, на одиницю перевищує число символів між подвійними лапками;
-константне вираз, що складається з одних констант, яке обчислюється під час трансляції (наприклад: a = 60 +301);
-типу long double, в кінці запису яких додається буква L (або l), наприклад: 1234567.89L.

2. Модуль 2. Бібліотечні функції. Операції

2.1 Операції присваювання, додавання, віднімання, множення, ділення,
Функції стандартного введення/виведення даних scanf, printf

Вирази і присвоювання

Комбінація знаків операцій та операндів, результатом якої є певне значення, називається виразом. Знаки операцій визначають дії, які повинні бути виконані над операндами. Кожен операнд у виразі може бути вираженням. Значення виразу залежить від розташування знаків операцій і круглих дужок у виразі, а також від пріоритету виконання операцій.

У мові СІ присвоювання також є вираженням, і значенням такого виразу є величина, яка присвоюється.

При обчисленні виразів тип кожного операнда може бути перетворений до іншого типу. Перетворення типів можуть бути неявними, при виконанні операцій і викликів функцій, або явними, при виконанні операцій приведення типів.

Операнд - це константа, літерал, ідентифікатор, виклик функції, індексне вираз, вираз вибору елемента або більш складне вираз, сформований комбінацією операндів, знаків операцій і круглих дужок. Будь операнд, який має константні значення, називається константним виразом. Кожен операнд має тип.

Якщо як операнд використовується константа, то йому відповідає значення і тип представляє його константи. Ціла константа може бути типу int, long, unsigned int, unsigned long, в залежності від її значення і від форми запису. Символьна константа має тип int. Константа з плаваючою точкою завжди має тип double.

Строковий літерал складається з послідовності символів, укладених в лапки, і представляється в пам'яті як масив елементів типу char, ініціалізіруемих зазначеної послідовністю символів. Значенням строкового літерала є адреса першого елемента рядки і синтаксично рядковий літерал є немодіфіціруемих покажчиком на тип char. Строкові літерали можуть бути використані в якості операндів у виразах, що допускають величини типу покажчиків. Однак так як рядки не є змінними, їх не можна використовувати в лівій частині операції привласнення.

Слід пам'ятати, що останнім символом рядка завжди є нульовий символ, який автоматично додається при зберіганні рядка в пам'яті.

Ідентифікатори змінних і функцій. Кожен ідентифікатор має тип, який встановлюється при його оголошенні. Значення ідентифікатора залежить від типу наступним чином:

- Ідентифікатори об'єктів цілих і плаваючих типів представляють значення відповідного типу;

- Ідентифікатор об'єкта типу enum представлений значенням однієї константи з безлічі значень констант в перерахуванні. Значенням ідентифікатора є константні значення. Тип значення є int, що випливає з визначення перерахування;

- Ідентифікатор об'єкта типу struct або union представляє значення, визначене структурою або об'єднанням;

- Ідентифікатор, оголошений як покажчик, представляє покажчик на значення, задане в оголошенні типу;

- Ідентифікатор, оголошений як масив, представляє покажчик, значення якого є адресою першого елемента масиву. Тип адресуються покажчиком величин - це тип елементів масиву. Зазначимо, що адреса масиву не може бути змінений під час виконання програми, хоча значення окремих елементів може змінюватися. Значення індексу, репрезентована ідентифікатором масиву, не є змінною і тому ідентифікатор масиву не може з'являтися в лівій частині оператора присвоювання.

- Ідентифікатор, оголошений як функція, представляє покажчик, значення якого є адресою функції, що повертає значення певного типу. Адреса функції не змінюється під час виконання програми, міняється тільки значення, що повертається. Таким чином, ідентифікатори функцій не можуть з'являтися в лівій частині операції привласнення.

Виклик функцій складається з виразу, за яким слід необов'язковий список виразів в круглих дужках:

вираз-1 ([список виразів])

Значенням виразу-1 повинен бути адреса функції (наприклад, ідентифікатор функції). Значення кожного виразу зі списку виразів передається у функцію в якості фактичного аргументу. Операнд, що є викликом функції, має тип і значення повертається функцією значення.

Індексне вираз задає елемент масиву і має вигляд:

вираз-1 [вираз-2]

Тип індексного виразу є типом елементів масиву, а значення становить величину, адреса якої обчислюється за допомогою значень вираз-1 і вираз-2.

Зазвичай вираз-1 - це вказівник, наприклад, ідентифікатор масиву, а вираз-2 - це ціла величина. Однак потрібно тільки, щоб одне з виразів було покажчиком, а друге целочисленной величиною. Тому вираз-1 може бути целочисленной величиною, а вираз-2 покажчиком. У будь-якому випадку вираз-2 повинна бути укладена в квадратні дужки. Хоча індексне вираз зазвичай використовується для посилань на елементи масиву, проте індекс може з'являтися з будь-яким покажчиком.

Індексні вирази для посилання на елементи одновимірного масиву обчислюються шляхом складання цілої величини зі значеннями покажчика з подальшим застосуванням до результату операції разадресації (*).

Так як один з виразів, зазначених в індексному вираженні, є покажчиком, то при складанні використовуються правила адресної арифметики, згідно з якими ціла величина перетвориться до адресного поданням, шляхом множення її на розмір типу, що адресується вказівником. Нехай, наприклад, ідентифікатор arr оголошений як масив елементів типу double.

double arr [10];

Таким чином, щоб отримати доступ до і-тому елементу масиву arr можна написати arr [i], що, в силу сказаного вище, еквівалентно і [a]. При цьому величина і множиться на розмір типу double і являє собою адресу і-го елемента масиву arr від його початку. Потім це значення складається зі значенням покажчика arr, що в свою чергу дає адресу і-го елемента масиву. До отриманого адресою застосовується операція разадресації, тобто здійснюється вибірка елемента масиву arr по сформованому адресою.

Таким чином, результатом індексного виразу arr [i] (або і [arr]) є значення і-го елемента масиву.

Вираз з кількома індексами посилається на елементи багатовимірних масивів. Багатовимірний масив - це масив, елементами якого є масиви. Наприклад, першим елементом тривимірного масиву є масив з двома вимірами.

Для посилання на елемент багатовимірного масиву індексне вираз повинен мати кілька індексів ув'язнених до квадратні дужки:

вираз-1 [вираз-2] [вираз-3] ...

Таке індексне вираз інтерпретується зліва направо, тобто спочатку розглядається Перший індексний вираз:

вираз-1 [вираз-2]

Результат цього виразу є адресний вираз, з яким складається вираз-3 і т.д. Операція разадресації здійснюється після обчислення останнього індексного виразу. Зазначимо, що операція разадресації не застосовується, якщо значення останнього вказівника адресує величину типу масиву.

Приклад:

int mass [2] [5] [3];

Розглянемо процес обчислення індексного виразу mass [1] [2] [2].

1. Обчислюється вирази mass [1]. Посилання індекс 1 множиться на розмір елемента цього масиву, елементом ж цього масиву є двомірний масив містить 5x3 елементів, що мають тип int. Одержуване значення складається зі значенням покажчика mass. Результат є вказівник на другий двомірний масив розміром (5x3) в тривимірному масиві mass.

2. Другий індекс 2 вказує на розмір масиву з трьох елементів типу int, і складається з адресою, відповідним mass [1].

3. Так як кожен елемент тривимірного масиву - це величина типу int, то індекс 2 збільшується на розмір типу int перед складанням з адресою mass [1] [2].

4. Нарешті, виконується разадресація отриманого покажчика. Результуючим виразом буде елемент типу int.

Якщо було б вказано mass [1] [2], то результатом був би покажчик на масив з трьох елементів типу int. Відповідно значенням індексного виразу mass [1] є вказівник на двовимірний масив.

Вираз вибору елемента застосовується, якщо в якості операнда треба використовувати елемент структури або об'єднання. Такий вираз має значення і тип обраного елемента. Розглянемо дві форми вираження вибору елемента:

вираз.ідентіфікатор,

вираз-> ідентифікатор.

У першій формі вираження представляє величину типу struct або union, а ідентифікатор - це ім'я елемента структури або об'єднання. У другій формі вираження повинно мати значення адреси структури або об'єднання, а ідентифікатор - ім'ям обраного елемента структури або об'єднання.

Обидві форми вираження вибору елемента дають однаковий результат. Дійсно, запис, що включає знак операції вибору (->), є скороченою версією записи з точкою для випадку, коли висловом стоїть перед точкою передує операція разадресації (*), застосована до покажчика, тобто запис

вираз -> код

еквівалентна записи

(* Вираз). ідентифікатор

у випадку, якщо вираз є покажчиком.

Приклад:

```
struct tree {float num;
             int spisoc [5];
             struct tree * left;} tr [5], elem;
```

```
elem.left = & elem;
```

У наведеному прикладі використовується операція вибору (.) Для доступу до елемента left структурної змінної elem. Таким чином елементу left структурної змінної elem присвоюється адресу самої змінної elem, тобто змінна elem зберігає посилання на себе саму.

Приведення типів це зміна (перетворення) типу об'єкта. Для виконання перетворення необхідно перед об'єктом записати в дужках потрібний тип:

(Ім'я-типу) операнд.

Приведення типів використовуються для перетворення об'єктів одного скалярного типу в інший скалярний тип. Однак висловом з приведенням типу не може бути присвоєно інше значення.

Приклад:

```
int i;
bouble x;
b = (double) i +2.0;
```

У цьому прикладі ціла змінна і за допомогою операції приведення типів приводиться до плаваючого типу, а потім вже бере участь в обчисленні виразу.

Константне вираз - це вираз, результатом якого є константа. Операндом константного виразу можуть бути цілі константи, символьні константи, константи з плаваючою точкою, константи перерахування, вирази приведення типів, вирази з операцією sizeof та інші вирази. Однак на використання знаків операцій у константних виразах накладаються наступні обмеження:

1. В константних виразах можна використовувати операції привласнення і послідовного обчислення (,).
2. Операція "адреса" (&) може бути використана тільки при деяких ініціалізація.

Вирази зі знаками операцій можуть брати участь у виразах як операнди. Вирази зі знаками операцій можуть бути унарний (з одним операндом), бінарними (з двома операндами) і тернарних (з трьома операндами).

Унарне вираз складається з операнда і попереднього йому знаку унарний операції і має наступний формат:

знак-унарний-операції операнд.

Бінарне вирази складається з двох операндів, розділених знаком бінарної операції:

операнд1 знак-бінарної-операції операнд2.

Тернарної вираз складається з трьох операндів, розділених знаками тернарної операції (?) I (:), і має формат:

операнд1? операнд2: операнд3.

Операції. За кількістю операндів, що беруть участь в операції, операції поділяються на унарні, бінарні і тернарние.

У мові Сі є наступні унарні операції:

- Арифметичне заперечення (заперечення і доповнення);

~ Побітове логічне заперечення (доповнення);

! логічне заперечення;

* Разадресація (непряма адресація);

& Обчислення адреси;

+ Унарний плюс;

+ + Збільшення (інкремент);

- Зменшення (декремент);

sizeof розмір.

Унарні операції виконуються справа наліво.

Операції збільшення і зменшення збільшують або зменшують значення операнда на одиницю і можуть бути записані як праворуч так і ліворуч від операнда. Якщо знак операції записаний перед операндом (префіксная форма), то зміна операнда відбувається до його використання в вираженні. Якщо знак операції записаний після операнда (постфіксной форма), то операнд спочатку використовується у виразі, а потім відбувається його зміна.

На відміну від унарних, бінарні операції, список яких наведено в табл.7, виконуються зліва направо.

Знак операції	Операція	Група операцій
*	Множення	Мультиплікативні
/	Розподіл	
%	Залишок від ділення	
+	Додавання	Адитивні
-	Віднімання	
<<	Зрушення вліво	Операції зсуву

>>	Зрушення вправо	
<	Менше	Операції відносини
<=	Менше або дорівнює	
> =	Більше або дорівнює	
==	Само	
! =	Не одно	
&	Порозрядне і	Порозрядного операції
|	Порозрядне АБО	
^	Порозрядне виключає АБО	
&&	Логічне І	Логічні операції
| |	Логічне АБО	
,	Послідовне обчислення	Послідовного обчислення
=	Присвоєння	Операції присвоювання
* =	Множення з привласненням	
/ =	Ділення з привласненням	
% =	Залишок від ділення з присвоюванням	
- =	Віднімання з привласненням	
+ =	Складання з привласненням	
<< =	Зрушення вліво з привласненням	
>> =	Зрушення вправо привласненням	
& =	Порозрядне І з привласненням	
| =	Порозрядне АБО з привласненням	
^ =	Порозрядне виключає АБО з привласненням	

Лівий операнд операції привласнення повинен бути виразом, що посилається на область пам'яті (але не об'єктом оголошеним з ключовим словом const), такі вирази називаються леводопустімимі до них відносяться:

- Ідентифікатори даних цілого і плаваючого типів, типів покажчика, структури, об'єднання;

- Індексні вирази, виключаючи вирази мають тип масиву або функції;

-Вирази вибору елемента (->) і (.), Якщо обраний елемент є леводопустімим;

-Вирази унарний операції разадресаціі (*), за винятком виразів, які посилаються на масив або функцію;

- Вираз приведення типу якщо результуючий тип не перевищує розміру первинного типу.

При запису виразів слід пам'ятати, що символи (*), (&), (!), (+) Можуть \ позначати унарні або бінарну операцію.

Операції заперечення та доповнення

Операція арифметичного заперечення (-) виробляє заперечення свого операнда. Операнд повинен бути цілою або плаваючою величиною. При виконанні здійснюються звичайні арифметичні перетворення.

Приклад:

```
double u = 5;
u =-u; / * змінної u присвоюється її заперечення,
                тобто u приймає значення -5 * /
```

Операція логічного заперечення "НЕ" (!) Виробляє значення 0, якщо операнд є істина (не нуль), і значення 1, якщо операнд дорівнює нулю (0). Результат має тип int. Операнд повинен бути цілого або плаваючого типу або типу покажчик.

Приклад:

```
int t, z = 0;
t =! z;
```

Мінлива t отримає значення рівне 1, так як змінна z мала значення рівне 0 (помилково).

Операція двійкового доповнення (~) виробляє двійкове доповнення свого операнда. Операнд повинен бути цілого типу. Здійснюється звичайне арифметичне перетворення, результат має тип операнда після перетворення.

Приклад:

```
char b = '9 ';
unsigned char f;
 b = ~ f;
```

Шістнадцяткове значення символу '9 'одно 39. В результаті операції ~ f буде отримано шістнадцяткове значення C6, що відповідає символу 'ц'.

Операції збільшення і зменшення

Операції збільшення (+ +) і зменшення (-) є унарний операціями присвоєння. Вони відповідно збільшують або зменшують значення операнда на одиницю. Операнд може бути цілого або плаваючого типу або типу покажчик і повинен бути модифікується. Операнд цілого або плаваючого типу збільшуються (зменшуються) на одиницю. Тип результату відповідає типу операнда. Операнд адресного типу збільшується або зменшується на розмір об'єкта, який він адресує. У мові допускається префіксная або постфіксной форми операцій збільшення (зменшення), тому значення виразу, що використовує операції збільшення (зменшення) залежить від того, яка з форм зазначених операцій використовується.

Якщо знак операції стоїть перед операндом (префіксная форма запису), то зміна операнда відбувається до його використання в вираженні і результатом операції є збільшене або зменшене значення операнда.

У тому випадку якщо знак операції стоїть після операнда (постфіксной форма запису), то операнд спочатку використовується для обчислення виразу, а потім відбувається зміна операнда.

Приклади:

```
int t = 1, s = 2, z, f;
    z = (t + +) * 5;
```

Спочатку відбувається множення t * 5, а потім збільшення t. В результаті вийде z = 5, t = 2.

```
    f = (+ + s) / 3;
```

Спочатку значення s збільшується, а потім використовується в операції ділення. В результаті отримаємо s = 3, f = 1.

У випадку, якщо операції збільшення та зменшення використовуються як самостійні оператори, префіксная і постфіксной форми запису стають еквівалентними.

z + +; / * еквівалентно * / + + z;

Бібліотечні функції

У системах програмування підпрограми для вирішення часто зустрічаються завдань об'єднуються в бібліотеки. До числа таких задач належать: обчислення математичних функцій, введення / виведення даних, обробка рядків, взаємодію із засобами операційної системи та ін Використання бібліотечних підпрограм позбавляє користувача від необхідності розробки відповідних засобів і надає йому додатковий сервіс. Включені в бібліотеки функції поставляються разом із системою програмування. Їх оголошення дані в файлах *. Н (це так звані включаються або заголовні файли). Тому, як вже згадувалося вище, на початку програми з бібліотечними функціями повинні бути рядки виду:

```
# include <включаемий_файл_тіпа_h>
```

наприклад:

```
# include <condio.h>
```

Існують також кошти для розширення і створення нових бібліотек з програмами користувача.

Функція стандартного виводу printf ()

Функція printf () є функцією стандартного виводу. За допомогою цієї функції можна вивести на екран монітора рядок символів, число, значення змінної ..

Функція printf () має прототип у файлі stdio.h

int printf (char * керуюча рядок, ...);

У разі успіху функція printf () повертає число виведених символів.

Керуюча рядок містить два типи інформації: символи, які безпосередньо виводяться на екран, і специфікатори формату, що визначають, як виводити аргументи.

Функція printf () це функція форматованого виводу. Це означає, що в параметрах функції необхідно вказати формат даних, які будуть виводитися. Формат даних вказується специфікаторами формату. Специфікатор формату починається із символу% за яким слідує код формату.

Специфікатори формату:% з символ

% D ціле десяткове число

% I ціле десяткове число

% E десяткове число у вигляді x.xx e + xx

% E десяткове число у вигляді x.xx E + xx

% F десяткове число з плаваючою комою xx.xxxx

% F десяткове число з плаваючою комою xx.xxxx

% G% f або% e, що коротше

% G% F або% E, що коротше

% O вісімкове число

% S рядок символів

% U беззнаковое десяткове число

% X шістнадцяткове число

% X шістнадцяткове число

%% Символ%

% P покажчик

% N покажчик

Крім того, до команд формату можуть бути застосовані модифікатори l і h.% Ld друк long int

% Hu друк short unsigned

% Lf друк long double

В специфікатор формату, після символу% може бути вказана точність (число цифр після коми). Точність задається наступним чином:%. N <код формату>. Де n - число цифр після коми, а <код формату> - один з кодів наведених вище.

Наприклад, якщо у нас є змінна x = 10.3563 типу float і ми хочемо вивести її значення з точністю до 3-х цифр після коми, то ми повинні написати:

printf ("Змінна x =% .3 f", x);

Результат:

Мінлива x = 10.356

Ви також можете вказати мінімальну ширину поля відведеного для друку. Якщо рядок або число більше зазначеної ширини поля, то рядок або число друкується повністю.

Наприклад, якщо ви напишіть:

```
printf ("% 5d", 20);
```

то результат буде таким:

```
  20
```

Зверніть увагу на те, що число 20 надрукувати не з самого початку рядка. Якщо ви хочете, щоб невикористані місця поля заповнювалися нулями, то потрібно поставити перед шириною поля символ 0.

Наприклад:

```
printf ("% 05d", 20);
```

Результат:

```
00020
```

Крім специфікаторів формату даних в керуючої рядку можуть перебувати керуючі символи: \ b BS, забій

\ F Нова сторінка, переклад сторінки

\ N Новий рядок, переклад рядка

\ R Повернення каретки

\ T Горизонтальна табуляція

\ V Вертикальна табуляція

\ "Подвійна лапка

\ 'Апостроф

\ \ Зворотній слеш

\ 0 Нульовий символ, нульовий байт

\ A Сигнал

\ N Вісімкова константа

\ XN Шістнадцяткова константа

\? Знак питання

Частіше за все ви будете використовувати символ \ n. За допомогою цього керуючого символу ви зможете переходити на новий рядок.

Функція стандартного введення scanf ()

Функція scanf () - функція форматованого введення. З її допомогою ви можете вводити дані зі стандартного пристрою вводу (клавіатури). Вводяться даними можуть бути цілі числа, числа з плаваючою комою, символи, рядки і покажчики.

Функція scanf () має наступний прототип у файлі stdio.h:
int scanf (char * керуюча рядок);

Функція повертає число змінних яким було присвоєно значення.
Керуюча рядок містить три види символів: специфікатор формату, пробіли та інші символи. Специфікатори формату починаються з символу%.

Специфікатори формату:% с читання символу
% D читання десяткового цілого
% I читання десяткового цілого
% E читання числа типу float (плаваюча кома)
% H читання short int
% O читання восьмеричного числа
% S читання рядки
% X читання шістнадцяткового числа
% P читання покажчика
% N читання покажчика в збільшеному форматі

При введенні рядка за допомогою функції scanf () (специфікатор формату % s), рядок вводитися до першого пробілу!! тобто якщо ви вводите рядок "Привіт світ!" з використанням функції scanf ()

char str [80]; / / масив на 80 символів
scanf ("% s", str);
то після введення результуюча рядок, який буде зберігатися в масиві str буде складатися з одного слова "Привіт". ФУНКЦІЯ введення рядка ДО ПЕРШОГО ПРОБІЛ! Якщо ви хочете вводити рядки з пробілами, то використовуйте функцію

char * gets (char * buf);

За допомогою функції gets () ви зможете вводити повноцінні рядка. Функція gets () читає символи з клавіатури до появи символу нового рядка (\ n). Сам символ нового рядка з'являється, коли ви натискаєте клавішу enter. Функція повертає вказівник на buf. buf - буфер (пам'ять) для введеної рядка.
Хоча gets () не входить в тему цієї статті, але все ж давайте напишемо приклад програми, яка дозволяє ввести цілий рядок з клавіатури та вивести її на екран.

```
# Include <stdio.h>

void main (void)
{
   char buffer [100]; / / масив (буфер) для введеної рядки
```

```
    gets (buffer); / / вводимо рядок і натискаємо enter
    printf ("% s", buffer); / / вивід введеного рядка на екран
}
```

Ще одне важливе зауваження! Для введення даних за допомогою функції scanf (), їй в якості параметрів потрібно передавати адреси змінних, а не самі змінні. Щоб отримати адресу змінної, потрібно поставити перед ім'ям змінної знак & (амперсанд). Знак & означає взяття адреси.

Що значить адресу? Спробую пояснити. У програмі у нас є змінна. Мінлива зберігає своє значення в пам'яті комп'ютера. Так от адресу, яку ми отримуємо за допомогою & це адреса в пам'яті комп'ютера де зберігатися значення змінної.

Давайте розглянемо приклад програми, який показує нам як використовувати &

```
# Include <stdio.h>

void main (void)
{
    int x;

    printf ("Введіть змінну x:");
    scanf ("% d", & x);
    printf ("Змінна x =% d", x);
}
```

Тепер давайте повернемося до керуючої рядку функції scanf (). Ще раз:

```
int scanf (char * керуюча рядок);
```

Символ пробілу в керуючої рядку дає команду пропустити один або більше прогалин у потоці введення. Крім пропусків може сприйматися символ табуляції або нового рядка. Ненульовий символ вказує на читання і відкидання цього символу.

Роздільниками між двома вводяться числами є символи пробілу, табуляції або нового рядка. Знак * після% і перед кодом формату (специфікатором формату) дає команду прочитати дані зазначеного типу, але не привласнювати це значення.

Наприклад:

```
scanf ("% d% * c% d", & i, & j);
```

при введенні 50 +20 присвоїть змінній і значення 50, змінної j - значення 20, а символ + буде прочитаний і проігнорований.

У команді формату може бути вказана найбільша ширина поля, яка підлягає зчитування.

Наприклад:

scanf ("% 5s", str);

вказує необхідність прочитати з потоку введення перші 5 символів. При введенні 1234567890ABC масив str буде містити тільки 12345, інші символи будуть проігноровані. Розділювачі: пробіл, символ табуляції і символ нового рядка - при введенні символу сприймаються, як і всі інші символи.

Якщо в керуючої рядку зустрічаються будь-які інші символи, то вони призначаються для того, щоб визначити і пропустити відповідний символ. Потік символів 10plus20 оператором

scanf ("% dplus% d", & x, & y);

присвоїть змінній x значення 10, змінної y - значення 20, а символи plus пропустить, так як вони зустрілися в керуючої рядку.

Однією з потужних особливостей функції scanf () є можливість завдання безлічі пошуку (scanset). Безліч пошуку визначає набір символів, з якими будуть порівнюватися читаються функцією scanf () символи. Функція scanf () читає символи до тих пір, поки вони зустрічаються в безлічі пошуку. Як тільки символ, який введений, не зустрівся в множині пошуку, функція scanf () переходить до наступного специфікатор формату. Безліч пошуку визначається списком символів, укладених у квадратні дужки. Перед відкриває дужкою ставитися знак%.

2.2 Математичні функції

До складу стандартної бібліотеки C входить величезна кількість математичних функцій, аналогічних діям з інженерного калькулятора.

abs - абсолютне значення (модуль)
acos, acosl - арккосинус
asin, asinl - арксинус
atan, atanl - арктангенс
atan2, atan2l - арктангенс2
cabs, cabsl - абсолютне значення комплексного числа
ceil, ceill - округлення вгору, найменше ціле, не меншу x
cos, cosl - косинус
cosh, coshl - косинус гіперболічний
exp, expl - експонента
fabs, fabs - абсолютний модуль дробу
floor, floorl - округлення вниз, найбільше ціле, не більше x
fmod, fmodl - залишок від ділення, аналог операції%
frexp, frexpl - поділяє число на мантиссу і експоненту

hypot, hypotl - гіпотенуза
labs - модуль довгого цілого
ldexp, ldexpl - твір числа на два в ступені exp
log, logl - логарифм натуральний
log10, log101 - логарифм лесятічний
modf, modfl - поділяє на цілу і на дробову частину
poly, polyl - поліном
pow, powl - ступінь
pow10, pow10l - ступінь десяти
sin, sinl - синус
sinh, sinhl - синус гіперболічний
sqrt, sqrtl - квадратний корінь
tan, tanl - тангенс
tanh, tanhl - тангенс гіперболічний

Для роботи з псевдовипадковими числами застосовуються функції stdlib.h randomize (ініціалізує генератор) і rand (генерує число). Для деяких комп'ютерів randomize буде неефективний і набагато краще буде використовувати функцію srand, що використовує системний час. Для псевдовипадкових чисел, не обмежених RAND_MAX можна написати власний макрос:

Define rand2 (a, b) (rand ()% b-a) + a

Генерує псевдовипадкове число між a і b.

2.3 Старшинство операций, ділення по модулю, операції інкременту та дикременту

У мові СІ операції з вищими пріоритетами обчислюються першими. Найвищим пріоритетом є пріоритет рівний 1.

Пріоритет	Знак операції	Типи операції	Порядок виконання
2	() []. ->	Вираз	Зліва направо
1	- ~! * & + + - Sizeof приведення типів	Унарні	Справа наліво
3	* /%	Мультиплікативні	Зліва направо
4	+ -	Адитивні	
5	<< >>	Зрушення	
6	<> <=> =	Ставлення	
7	==! =	Ставлення (рівність)	
8	&	Порозрядне І	
9	^	Порозрядне виключає АБО	
10	|	Порозрядне АБО	
11	&&	Логічне І	
12	| |	Логічне АБО	
13	? :	Умовна	

14	= * = / =% = + = - = = & = | = >> = << = ^	Просте і складене присвоювання	Справа наліво
15	,	Послідовне обчислення	Зліва направо

Операції інкременту та дикременту

Інкремент "+ +" - це збільшення на одиницю. Декремент "-" - це зменшення на одиницю. Операції декремента і інкремента з легкістю замінюються арифметичними операціями чи операціями присвоєння. Але використовувати операції інкремента і декремента набагато зручніше.

Ці операції відрізняються тим, що вони можуть бути записані як в постфіксной формі, коли символ операції слідує за операндом, як у наведених прикладах, так і в префіксной формі, коли він передує операнду. У наведених прикладах застосування будь-якої з цих форм не має ніякого значення. Однак, коли операції інкремента / декремента є частиною більш складного вираження, проявляється зовні незначне, але важлива відмінність між цими двома формами. В префіксной формі значення операнда збільшується або зменшується до вилучення значення для використання в вираженні. В постфіксной формі попереднє значення витягується для використання в вираженні, і лише після цього значення операнда змінюється. Наприклад:

```
x = 42;
y = + + x;
```

В цьому випадку значення у встановлюється рівним 43, як і можна було очікувати, оскільки збільшення значення виконується перед присвоюванням значення х змінної у. Таким чином, рядок у = + + х еквівалентна наступним двом операторам:

```
x = x + 1;
У = х;
```

Однак якщо оператори записати як

```
x = 42;
y = x + +;
```

значення змінної х витягується до виконання операції інкремента, і тому значення змінної у одно 42. Звичайно, в обох випадках значення змінної х встановлено рівним 43. Отже, рядок у = х + +; еквівалентна наступним двом операторам:

```
y = x;
```

```
x = x + 1;
```

Наступна програма демонструє застосування операції інкремента.

```
/ / Демонстрація застосування операції + +.
 class IncDec {
 public static void main (String args []) {
 int a = 1;
 int b = 2;
 int c;
 int d;
 c = + + b;
 d = a + +;
 C + +;
 System.out.println ("a =" + a);
 System.out.println ("b =" + b);
 System.out.println ("c =" + c);
 System.out.println ("d =" + d);
 }
 }
```

2.4 Приорітети операцій та порядок їх обчислення

Пріоритети операцій задають послідовність обчислень в складному вираженні. Наприклад, яке значення отримає ival?

```
int ival = 6 + 3 * 4/2 + 2;
```

Якщо обчислювати операції зліва направо, вийде 20. Серед інших можливих результатів будуть 9, 14 і 36. Правильна відповідь: 14.

В C + + множення і ділення мають вищий пріоритет, ніж додавання, тому вони будуть обчислені раніше. Їх власні пріоритети рівні, тому множення і ділення будуть обчислюватися зліва направо. Таким чином, порядок обчислення цього виразу такий:

1. 3 * 4 => 12
2. 12/2 => 6
3. 6 + 6 => 12
4. 12 + 2 => 14

Наступна конструкція веде себе не так, як можна було б очікувати. Пріоритет операції привласнення менше, ніж операції порівняння:

```
while (ch = nextChar ()! = '\ n')
```

Програміст хотів привласнити змінній ch значення, а потім перевірити, чи дорівнює воно символу нового рядка. Однак насправді вираз спочатку

порівнює значення, отримане від nextChar (), з '\ n', і результат - true або false - присвоює змінної ch.

Пріоритети операцій можна змінити за допомогою дужок. Вирази в дужках обчислюються в першу чергу. Наприклад:

4 * 5 + 7 * 2 ==> 34

4 * (5 + 7 * 2) ==> 76

4 * ((5 + 7) * 2) ==> 96

Ось як за допомогою дужок виправити поведінку попереднього прикладу:

while ((ch = nextChar ())! = '\ n')

Оператори володіють і пріоритетом, і асоціативністю. Оператор присвоювання правоассоціативен, тому обчислюється справа наліво:

ival = jval = kva1 = lval

Спочатку kval отримує значення lval, потім jval - значення результату цього присвоювання, і в кінці кінців ival отримує значення jval.

Арифметичні операції, навпаки, левоассоціативни. Отже, у виразі

ival + jval + kva1 + 1va1

спочатку складаються ival і jval, потім до результату додається kval, а потім і lval.

Всі оператори деякої секції мають вищий пріоритет, ніж оператори із секцій, наступних за нею. Так, операції множення і ділення мають однаковий пріоритет, і він вище пріоритету будь-якій з операцій порівняння.

2.5 Практична робота № 2: "Використання типів даних"

Типи спеціальних даних не підходять до жодної з інших категорій типів даних. У SQL Server присутні наступні особливі типи даних: bit, hierarchyid, sql_variant, sysname, table, timestamp, а також псевдоніми типів даних.

Тип даних bit

Тип даних bit є чисельною типом даних, які приймають значення 1 або 0. Строкові значення true і false можна перетворити на значення типу bit так, як це показано в наступному прикладі.

SELECT CONVERT (bit, 'true')

SELECT CONVERT (bit, 'false')

У даному прикладі значення true перетворюється у значення 1, а значення false перетворюється у значення 0. Дані типу bit не повинні бути укладені в одинарні лапки.

Тип даних hierarchyid

Тип даних hierarchyid використовується для управління ієрархічними даними і таблицями, що мають ієрархічну структуру. Для роботи з ієрархічними даними в коді Transact-SQL слід використовувати функції hierarchyid. Додаткові відомості див Використання типів даних hierarchyid (компонент Database Engine).

Тип даних sql_variant

Тип даних sql_variant дозволяє зберігати дані різних типів в одному стовпці, параметрі або змінної. У кожному примірнику стовпця типу sql_variant зберігаються значення та метадані, що описують ці значення. Доступні наступні метадані.

Базовий тип даних

Максимальний розмір

Масштаб

Точність

Параметри сортування

Для отримання метаданих певного примірника sql_variant слід використовувати функцію SQL_VARIANT_PROPERTY.

В наступному прикладі в другій таблиці зберігається стовпець sql_variant.

```
CREATE TABLE ObjectTable (
  ObjectID int CONSTRAINT PKObjectTable PRIMARY KEY,
  ObjectName nvarchar (80),
  ObjectWeight decimal (10,3),
  ObjectColor nvarchar (20))
CREATE TABLE VariablePropertyTable (
  ObjectID int REFERENCES ObjectTable (ObjectID),
  PropertyName nvarchar (100),
  PropertyValue sql_variant,
  CONSTRAINT PKVariablePropertyTable
  PRIMARY KEY (ObjectID, PropertyName))
```

Тип даних sysname

Тип даних sysname використовується в стовпцях таблиці, змінних і параметрах збережених процедур, що містять назви об'єктів. Точне визначення sysname залежить від встановлених для ідентифікаторів правил іменування. Таким чином, воно може бути різним для різних екземплярів SQL Server. Тип даних sysname функціонально еквівалентний типу nvarchar (128), за винятком того, що за замовчуванням sysname має значення NOT NULL. У більш ранніх версіях SQL Server тип sysname визначений як varchar (30).

У базах даних, які враховують регістр або використовують двійкову сортування, тип даних sysname визначається як системний тип даних SQL Server лише тоді, коли він введений в нижньому регістрі.

Тип даних table

Тип даних table використовується для визначення тимчасових таблиць. У змінних цього типу зберігаються результуючі набори для подальшої обробки. Цей тип даних може використовуватися тільки для визначення локальних змінних типу table і значень, що повертаються користувальницької функцією.

Визначення табличній змінної або значення, що повертається включає визначення стовпців, типу даних, точності та розміру кожного стовпця, необов'язкових обмежень PRIMARY KEY, UNIQUE, NULL або CHECK. Певна користувачем таблиця не може бути використана в якості типу даних.

Формат рядків, що зберігаються у змінній table або повертаються визначається користувачем функцією, повинен бути визначений при оголошенні змінної або при створенні функції. Синтаксис заснований на синтаксисі інструкції CREATE TABLE, як показано в наступному прикладі.

```
DECLARE @ TableVar TABLE (Cola int PRIMARY KEY, Colb char (3))
INSERT INTO @ TableVar VALUES (1, 'abc')
INSERT INTO @ TableVar VALUES (2, 'def')
SELECT * FROM @ TableVar
GO
```

Змінні table і призначені для користувача функції, які повертають table, можна використовувати в певних інструкціях SELECT і INSERT і в інструкціях UPDATE, DELETE і DECLARE CURSOR, що підтримують таблиці. Змінні table і призначені для користувача функції, які повертають table, не можна використовувати в інструкціях Transact-SQL.

Індекси та інші застосовні до таблиці обмеження задаються як частина змінної DECLARE в інструкції CREATE FUNCTION. Вони не можуть бути застосовані пізніше, тому що інструкції CREATE INDEX та ALTER TABLE не можуть посилатися на табличні змінні і визначені користувачем функції.

Додаткові відомості про синтаксис, що використовується для визначення змінних і визначаються користувачем функцій типу table, див в розділах DECLARE @ local_variable (Transact-SQL) і CREATE FUNCTION (Transact-SQL).

Тип даних timestamp

Тип даних timestamp не пов'язаний з часом або з датою. Значення timestamp є двійковими числами, що вказують відносну послідовність, в якій відбувалася зміна даних в базі даних.

Ніколи не використовуйте стовпці timestamp в якості ключів, особливо первинних ключів, тому що значення timestamp змінюється щоразу при зміні рядка.

Щоб записувати час, коли відбувалася зміна даних у таблиці, використовуйте або тип даних datetime2, або smalldatetime для запису подій. Використовуйте тригери для автоматичного оновлення значень всякий раз, коли відбуваються зміни.

Псевдоніми типів даних

Псевдоніми типів даних дозволяють розширити базові типи даних SQL Server (наприклад, varchar) змістовним ім'ям і форматом, який можна пристосувати для конкретного використання. Наприклад, наступна інструкція реалізує визначений користувачем тип даних birthday, заснований на типі даних datetime, що допускає значення NULL.

EXEC sp_addtype birthday, datetime, 'NULL'

Будьте уважні при виборі базових типів, що застосовуються для визначених користувачем типів даних. Наприклад, в США номера соціального страхування (SSN) мають формат nnn-nn-nnnn. Хоча номери соціального страхування містять числа, ці числа утворюють ідентифікатор і до них не можна застосовувати математичні операції. Тому зазвичай визначається користувальницький тип даних varchar для номера соціального страхування і створюється таке перевірочне обмеження, що забезпечує необхідний формат номера соціального страхування.

EXEC sp_addtype SSN, 'VARCHAR (11)', 'NOT NULL'

GO

CREATE TABLE ShowSSNUsage (EmployeeID int PRIMARY KEY, EmployeeSSN SSN, CONSTRAINT CheckSSN CHECK (EmployeeSSN LIKE '[0-9] [0-9] [0-9] - [0-9] [0-9] - [0-9] [0-9] [0-9] [0-9] ')

)

GO

Якщо стовпці SSN використовуються в якості ключових стовпців в індексах, особливо в кластеризованих індексах, то розмір ключів можна скоротити з 11 байт до 4 байт, якщо визначений користувачем тип даних SSN визначений з використанням базового типу даних int. Зменшення розміру ключа покращує вилучення даних. Поліпшення ефективності вилучення даних і усунення необхідності перевірочного обмеження CHECK зазвичай переважують витрати, пов'язані з перетворенням типу int до символьному формату при відображенні або зміну значень SSN.

3.3 МОДУЛЬ 3: Управління обчислюючим процесом. Оператори.

3.1 Оператор вітвлення if, конструкція else-if, if-else. Оператор switch

Оператор if

Формат оператора:
if (вираз) оператор-1; [else оператор-2;]
Виконання оператора if починається з обчислення виразу.
Далі виконання здійснюється за наступною схемою:

- Якщо вираз істинно (тобто відмінно від 0), то виконується оператор-1.
- Якщо вираз помилково (тобто дорівнює 0), то виконується оператор-2.
- Якщо вираз помилково і відсутній оператор-2 (у квадратні дужки укладена необов'язкова конструкція), то виконується наступний за if оператор.

Після виконання оператора if значення передається на наступний оператор програми, якщо послідовність виконання операторів програми не буде примусово порушена використанням операторів переходу.
Приклад:

```
if (i <j) i + +:
else {j = i-3; i + +;}
```

Цей приклад ілюструє також і той факт, що на місці оператор-1, так само як і на місці оператор-2 можуть знаходитися складні конструкції.
Допускається використання вкладених операторів if. Оператор if може бути включений в конструкцію if або в конструкцію else іншого оператора if. Щоб зробити програму більш читабельною, рекомендується групувати оператори і конструкції у вкладених операторах if, використовуючи фігурні дужки. Якщо ж фігурні дужки опущені, то компілятор пов'язує кожне ключове слово else з найбільш близьким if, для якого немає else.
Приклади:

```
int main ()
 {
  int t = 2, b = 7, r = 3;
    if (t> b)
     {
        if (b <r) r = b;
     }
    else r = t;
```

```
  return (0);
 }
```

В результаті виконання цієї програми r стане рівним 2.

Якщо ж у програмі опустити фігурні дужки, що стоять після оператора if, то програма буде мати наступний вигляд:

```
int main ()
 {
  int t = 2, b = 7, r = 3;
  if (a> b)
     if (b <c) t = b;
     else r = t;
  return (0);
 }
```

В цьому випадку r отримає значення рівне 3, так як ключове слово else відноситься до другого оператору if, який не виконується, оскільки не виконується умова, що перевіряється в першому операторі if.

Наступний фрагмент ілюструє вкладені оператори if:

```
char ZNAC;
int x, y, z;
  :
 if (ZNAC == '-') x = y - z;
 else if (ZNAC == '+') x = y + z;
      else if (ZNAC == '*') x = y * z;
           else if (ZNAC == '/') x = y / z;
                else ...
```

З розгляду цього прикладу можна зробити висновок, що конструкції використовують вкладені оператори if, є досить громіздкими і не завжди достатньо надійними. Іншим способом організації вибору з безлічі різних варіантів є використання спеціального оператора вибору switch.

Оператор switch

Оператор switch призначений для організації вибору з безлічі різних варіантів. Формат оператора наступний:

```
switch (вираз)
{[Оголошення]
      :
   [Case константні-вираз1]: [список-операторов1]
   [Case константні-вираз2]: [список-операторов2]
     :
     :
   [Default: [список операторів]]
}
```

Вираз, наступне за ключовим словом switch в круглих дужках, може бути будь-яким виразом, допустимими в мові CI, значення якого повинно бути цілим. Зазначимо, що можна використовувати явне приведення до цілого типу, проте необхідно пам'ятати про ті обмеження та рекомендаціях, про які говорилося вище.

Значення цього виразу є ключовим для вибору з декількох варіантів. Тіло оператора smitch складається з декількох операторів, позначених ключовим словом case з наступним константним-виразом. Слід зазначити, що використання цілого константного виразу є істотним недоліком, властивим розглянутому оператору.

Так як константні вираз обчислюється під час трансляції, воно не може містити змінні або виклики функцій. Звичайно як константного виразу використовуються цілі або символьні константи.

Всі вирази в операторі switch повинні бути унікальні. Крім операторів, позначених ключовим словом case, може бути, але обов'язково один, фрагмент позначений ключовим словом default.

Список операторів може бути порожнім, або містити один або більше операторів. Причому в операторі switch не потрібно укладати послідовність операторів у фігурних дужках.

Відзначимо також, що в операторі switch можна використовувати свої локальні змінні, оголошення яких знаходяться перед першим ключовим словом case, проте в оголошеннях не повинна використовуватися ініціалізація.

Схема виконання оператора switch наступна:

- Обчислюється вираз в круглих дужках;
- Обчислені значення послідовно порівнюються з константними виразами, наступними за ключовими словами case;
- Якщо одне з константних виразів збігається зі значенням виразу, то управління передається на оператор, позначений відповідним ключовим словом case;
- Якщо жодна з константних виразів не одно висловом, то управління передається на оператор, позначений ключовим словом default, а в разі його відсутності управління передається на наступний після switch оператор.

Відзначимо цікаву особливість використання оператора switch: конструкція зі словом default може бути не останньою в тілі оператора switch. Ключові слова case і default в тілі оператора switch істотні тільки при початковій перевірці, коли визначається початкова точка виконання тіла оператора switch. Всі оператори, між початковим оператором і кінцем тіла, виконуються незалежно від ключових слів, якщо тільки якийсь з операторів не передасть управління з тіла оператора switch. Таким чином, програміст повинен сам подбає про вихід з case, якщо це необхідно. Найчастіше для цього використовується оператор break.

Для того, щоб виконати ті самі дії для різних значень вираження, можна помітити один і той же оператор декількома ключовими словами case.

Приклад:

```
int i = 2;
switch (i)
  {
   case 1: i + = 2;
   case 2: i * = 3;
   case 0: i / = 2;
   case 4: i - = 5;
   default:;
}
```

Виконання оператора switch починається з оператора, позначеного case 2. Таким чином, змінна і отримує значення, рівне 6, далі виконується оператор, позначений ключовим словом case 0, а потім case 4, мінлива і прийме значення 3, а потім значення -2. Оператор, позначений ключовим словом default, не змінює значення змінної.

Розглянемо раніше наведений приклад, в якому ілюструвалося використання вкладених операторів if, переписаної тепер з використанням оператора switch.

```
char ZNAC;
int x, y, z;
switch (ZNAC)
{
    case '+': x = y + z; break;
    case '-': x = y - z; break;
    case '*': x = y * z; break;
    case '/': x = u / z; break;
    default:;
  }
```

Використання оператора break дозволяє в необхідний момент перервати послідовність виконуваних операторів в тілі оператора switch, шляхом передачі управління оператору, наступного за switch.

Відзначимо, що в тілі оператора switch можна використовувати вкладені оператори switch, при цьому в ключових словах case можна використовувати однакові вирази.

Приклад:

```
 :
switch (a)
 {
  case 1: b = c; break;
  case 2:
     switch (d)
     {Case 0: f = s; break;
        case 1: f = 9; break;
        case 2: f-= 9; break;
```

```
    }
  case 3: b-= c; break;
  :
}
```

3.2 Пустий оператор. Складний оператор

Порожній оператор

Порожній оператор складається тільки з точки з комою. При виконанні цього оператора нічого не відбувається. Він зазвичай використовується в наступних випадках:

- В операторах do, for, while, if в рядках, коли місце оператора не потрібно, але по синтаксису потрібно хоча б один оператор;
- При необхідності позначити фігурну дужку.

Синтаксис мови CI вимагає, щоб після мітки обов'язково слідував оператор. Фігурна ж дужка оператором не є. Тому, якщо треба передати управління на фігурну дужку, необхідно використовувати порожній оператор.

приклад:

```
int main ()
{
    :
     {If (...) goto a; / * перехід на дужку * /
        {...
        }
 a:;}
     return 0;
 }
```

Складний оператор

Складний оператор представляє собою кілька операторів і оголошень, укладених у фігурні дужки:

```
{[| Оголошення]
   :
   оператор; [оператор];
   :
}
```

Зауважимо, що в кінці складеного оператора крапка з комою не ставиться.

Виконання складеного оператора полягає в послідовному виконанні складових його операторів.

приклад:

```
int main ()
{
 int q, b;
 double t, d;
  :
  if (...)
    {
     int e, g;
     double f, q;
      :
     }
   :
   return (0);
}
```

Змінні e, g, f, q будуть знищені після виконання складного оператора. Зазначимо, що змінна q є локальною у складеному операторі, тобто вона жодним чином не пов'язана зі змінною q оголошеної на початку функції main до типу int. Відзначимо також, що вираз стоїть після return може бути укладено в круглі дужки, хоча наявність останніх необов'язково.

3.3 Оператори циклу while, do while
Оператор циклу for

Оператор for

Оператор for - це найбільш загальний спосіб організації циклу. Він має такий формат:

for (вираз 1; вираз 2; вираз 3) тіло

Вираз 1 зазвичай використовується для встановлення початкового значення змінних, які керують циклом. Вираз 2 - це вираз, що визначає умову, при якому тіло циклу буде виконуватися. Вираз 3 визначає зміна змінних, керуючих циклом після кожного виконання тіла циклу.

Схема виконання оператора for:

1. Обчислюється вираз 1.
2. Обчислюється вираз 2.
3. Якщо значення виразу 2 відмінно від нуля (істина), виконується тіло циклу, обчислюється вираз 3 і здійснюється перехід до пункту 2, якщо вираз 2 дорівнює нулю (неправда), то управління передається на оператор, наступний за оператором for.

Суттєво те, що перевірка умови завжди виконується на початку циклу. Це означає, що тіло циклу може ні разу не виконатися, якщо умова виконання відразу буде хибним.

Приклад:

```
int main ()
{Int i, b;
  for (i = 1; i <10; i + +)
    b = i * i;
  return 0;
}
```

У цьому прикладі обчислюються квадрати чисел від 1 до 9.

Деякі варіанти використання оператора for підвищують його гнучкість за рахунок можливості використання декількох змінних, керуючих циклом.

Приклад:

```
int main ()
{Int top, bot;
  char string [100], temp;
  for (top = 0, bot = 100; top <bot; top + +, bot -)
  {Temp = string [top];
    string [bot] = temp;
  }
  return 0;
 }
```

У цьому прикладі, реализующем запис рядка символів в зворотному порядку, для управління циклом використовуються дві змінні top і bot. Зазначимо, що на місці вираз 1 і вираз 3 тут використовуються кілька висловів, записаних через кому, і виконуваних послідовно.

Іншим варіантом використання оператора for є нескінченний цикл. Для організації такого циклу можна використовувати пусте умовне вираження, а для виходу з циклу зазвичай використовують додаткову умову і оператор break.

Приклад:

```
for (; ;)
{...
  ... break;
  ...
}
```

Так як згідно синтаксису мови Сі оператор може бути порожнім, тіло оператора for також може бути порожнім. Така форма оператора може бути використана для організації пошуку.

Приклад:

```
for (i = 0; t [i] <10; i + +);
```

У даному прикладі змінна циклу і приймає значення номера першого елемента масиву t, значення якого більше 10.

Оператор while

Оператор циклу while називається циклом з передумовою і має наступний формат:

while (вираз) тіло;

У якості вираження допускається використовувати будь-який вираз мови Сі, а в якості тіла будь-який оператор, у тому числі порожній або складової. Схема виконання оператора while наступна:

1. Обчислюється вираз.
2. Якщо вираз помилково, то виконання оператора while закінчується і виконується наступний по порядку оператор. Якщо вираз істинно, то виконується тіло оператора while.
3. Процес повторюється з пункту 1.

Оператор циклу виду

for (вираз-1; вираз-2; вираз-3) тіло;

може бути замінений оператором while наступним чином:

```
вираз-1;
while (вираз-2)
{тіло
  вираз-3;
}
```

Так само як і при виконанні оператора for, в операторі while спочатку відбувається перевірка умови. Тому оператор while зручно використовувати в ситуаціях, коли тіло оператора не завжди потрібно виконувати.

Всередині операторів for і while можна використовувати локальні змінні, які повинні бути оголошені з визначенням відповідних типів.

Оператор do while

Оператор циклу do while називається оператором циклу з постусловієм і використовується в тих випадках, коли необхідно виконати тіло циклу хоча б один раз. Формат оператора має наступний вигляд:

do тіло while (вираз);

Схема виконання оператора do while:

1. Виконується тіло циклу (яке може бути складовим оператором).
2. Обчислюється вираз.
3. Якщо вираз помилково, то виконання оператора do while закінчується і виконується наступний по порядку оператор. Якщо вираз істинно, то виконання оператора триває з пункту 1.

Щоб перервати виконання циклу до того, як умова стане хибним, можна використовувати оператор break.

Оператори while і do while можуть бути вкладеними.

приклад:

```
int i, j, k;
...
i = 0; j = 0; k = 0;
do {i + +;
    j -;
    while (a [k] <i) k + +;
   }
while (i <30 && j <-30);
```

3.4 Інструкція переходу: оператор continue

Оператор continue

Оператор continue, як і оператор break, використовується тільки всередині операторів циклу, але на відміну від нього виконання програми продовжується не з оператора, наступного за перерваним оператором, а з початку перерваного оператора. Формат оператора наступний:

```
continue;
```

приклад:

```
int main ()
{Int a, b;
   for (a = 1, b = 0; a <100; b + = a, a + +)
   {If (b% 2) continue;
     ... / * Обробка парних сум * /
   }
   return 0;
}
```

Коли сума чисел від 1 до a стає непарній, оператор continue передає керування на чергову ітерацію циклу for, не виконуючи оператори обробки парних сум.

Оператор continue, як і оператор break, перериває самий внутрішній з осяжний його циклів.

3.5 Інструкція переходу: оператор goto та мітка, break

Оператор goto

Використання оператора безумовного переходу goto в практиці програмування мовою CI настійно не рекомендується, так як він ускладнює розуміння програм та можливість їх модифікацій.

Формат цього оператора наступний:

```
goto ім'я-мітки;
        ...
ім'я-мітки: оператор;
```

Оператор goto передає управління на оператор, позначений міткою ім'я-мітки. Позначений оператор повинен знаходитися в тій же функції, що і оператор goto, а використовувана мітка повинна бути унікальною, тобто одне ім'я-мітки не може бути використано для різних операторів програми. Ім'я-мітки - це ідентифікатор.

Будь-який оператор у складеному операторі може мати свою мітку. Використовуючи оператор goto, можна передавати управління всередину складеного оператора. Але потрібно бути обережним при вході в складовою оператор, що містить оголошення змінних з ініціалізацією, так як оголошення розташовуються перед виконуваними операторами і значення оголошених змінних при такому переході будуть не визначені.

Оператор break

Оператор break забезпечує припинення виконання самого внутрішнього з об'єднують його операторів switch, do, for, while. Після виконання оператора break керування передається оператору, наступного за перерваним.

3.6 Оператор return

Оператор return

Оператор return завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию, в точку, непосредственно следующую за вызовом. Функция main передает управление операционной системе. Формат оператора:

return [выражение] ;

Значение выражения, если оно задано, возвращается в вызывающую функцию в качестве значения вызываемой функции. Если выражение опущено, то возвращаемое значение не определено. Выражение может быть заключено в круглые скобки, хотя их наличие не обязательно.

Если в какой-либо функции отсутствует оператор return, то передача управления в вызывающую функцию происходит после выполнения последнего оператора вызываемой функции. При этом возвращаемое значение не определено. Если функция не должна иметь возвращаемого значения, то ее нужно объявлять с типом void.

Таким образом, использование оператора return необходимо либо для немедленного выхода из функции, либо для передачи возвращаемого значения.

Пример:

```
int sum (int a, int b)
{   renurn (a+b);    }
```

Функция sum имеет два формальных параметра a и b типа int, и возвращает значение типа int, о чем говорит описатель, стоящий перед именем функции. Возвращаемое оператором return значение равно сумме фактических параметров.

Пример:

```
void prov (int a, double b)
{  double c;
  if (a<3) return;
  else if (b>10) return;
      else { c=a+b;
           if ((2*c-b)==11) return;
          }
 }
```

В этом примере оператор return используется для выхода из функции в случае выполнения одного из проверяемых условий.

3.7 Логічні оператори. І-АБО-НІ

В С + + існує три логічні операції:

1) Логічна операція І "&&", нам вже відома;
2) Логічна операція АБО "| |";
3) Логічна операція НЕ "!" або логічне заперечення.

Логічні операції утворюють складне (складене) умова з декількох простих (два або більше) умов. Ці операції спрощують структуру програмного коду в кілька разів. Так, можна обійтися і без них, але тоді кількість іфов збільшується в кілька разів, в залежності від умови. У наступній таблиці коротко охарактеризовані всі логічні операції в мові програмування С + +, для побудови логічних умов.

Логічна операція - в програмуванні операція над виразами логічного (Булевського) типу, що відповідає деякої операції над висловлюваннями в алгебрі логіки. Як і висловлювання, логічні вирази можуть приймати одне з двох істиннісних значень - «істинно» чи «хибно». Логічні операції служать для отримання складних логічних виразів з простіших. У свою чергу, логічні вираження зазвичай використовуються як умови для управління послідовністю виконання програми.

У деяких мовах програмування (наприклад в С) замість логічного типу або одночасно з ним використовуються числові типи. У цьому випадку

вважається, що відмінне від нуля значення відповідає логічної істини, а нуль - логічної брехні.

Значення окремого біта також можна розглядати як логічне, якщо вважати, що 1 означає «істинно», а 0 - «хибно». Це дозволяє застосовувати логічні операції до окремих бітам, до бітовим векторах покомпонентно і до чисел в двійковому поданні поразрядно. Таке одночасне застосування логічної операції до послідовності бітів здійснюється за допомогою побітових логічних операцій. Побітові логічні операції використовуються для оперування окремими бітами або групами бітів, застосовуються для накладення бітових масок, виконання різних арифметичних обчислень.

Серед логічних операцій найбільш відомі кон'юнкція (&&), диз'юнкція (| |), заперечення (!). Їх нерідко плутають з бітовими операціями, хоча це різні речі. Наприклад, наступний код на мові С:

```
if (action_required && some_condition ())
{
    / * Якісь дії * /
}
```

не виконає виклик підпрограми some_condition (), якщо значення логічної змінної action_required помилково. При такій операції другий аргумент операції && взагалі не буде обчислений.

3.8 Практична робота № 3: "Складання програм за допомогою операторів"

Після розгляду окремих складових частин програми визначимо загальні правила, використовувані при розробці С-програм:

на початку програми перераховуються директиви препроцесора;

програма складається з сукупності функцій, одна з яких повинна називатися main ();

опис функції складається з заголовка і тіла функції і називається визначенням функції або її реалізацією;

заголовок функції містить тип повертається функцією значення, ім'я функції і список параметрів;

ім'я функції пізнається по круглих дужках, які йдуть за ім'ям і можуть бути порожніми;

тіло функції укладено у фігурні дужки і складається з низки інструкцій.

Отже, маємо просту програму, написану з урахуванням дотримання вимог стандарту ISO / ANSI C.

Директиви препроцесора

Тема функції main ()

Тіло функції

```
# Include ...
 int main (void)
```

```
{
  оголошення змінних;
 ініціалізація змінних;
 виклики функції;
return 0;
}
```

Слідом за відкриває фігурною дужкою йдуть оголошення змінних. Як зазначалося раніше, всі змінні, використовувані в програмі, повинні бути оголошені, тобто для них повинні бути вказані імена і типи значень, наприклад:

```
int a;

float b, c;
```

За оголошенням змінних слідують інструкції, які вказують, як і в якому порядку треба перетворювати дані для отримання необхідного відповіді. Так, оператор присвоювання задає початкові значення змінних. Присвоєння змінній початкового значення називається визначенням або ініціалізацією змінної. Нарешті функція main () закінчується інструкцією повернення return. Тіло функції завершується символом}.

Наведемо як приклад найпростішу програму обчислення виразу "двічі два дорівнює чотири".

```
/ * First C-program * /
# Include <stdio.h>
int main ()
{
float x = 2., y;
y = x * x;
}
```

Розберемо наведену програму порядково.

/ * First C-program * / - цей рядок є коментарем. Все, що знаходиться всередині конструкції / * ... * /, ігнорується компілятором і дозволяє програмісту додати будь пояснювальний текст, даючи свободу коментувати програмний код з будь-яким ступенем подробиці.

Як завжди, при цьому виникають дві крайності. Одні програмісти пишуть програмні коди взагалі без коментарів, інші на кожен рядок коду додають кілька рядків коментарів. Слід мати на увазі, що наявність коментарів є необхідним атрибутом, що полегшує читання програми і визначальним хороший стиль програмування.

Include <stdio.h> - за допомогою цього рядка здійснюється підключення заголовки stdio.h, що надає доступ до засобів вводу / виводу інформації.

Вміст цього файлу копіюється препроцесором в файл програми на місце директиви include. Файл stdio.h міститься в спеціальному каталозі. Зазвичай це каталог INCLUDE, що є складовою частиною середовища розробки.

main () - в кожній програмі має бути присутня ця функція. При запуску програма "проходить" через кожен рядок коду у функції main () і виконує її.

Можна виділити чотири невід'ємних частини у визначенні функції:

тип повертається результату;

ім'я функції;

список аргументів (параметрів);

тіло функції.

У прикладі функція main () визначена як повертає ціле значення (int). Реально функція main () не поверне нічого - в ній просто немає оператора return. При компіляції такої програми буде отримано попередження (warning) про те, що функція повинна повертати значення. Проте це попередження не перешкодить виконанню програми. Для виключення появи такого попередження перед фігурною дужкою, що закриває тіло функції, слід написати: return 0;

Отже,

повертаний тип: int

ім'я функції: main ()

список аргументів: ()

тіло функції: {...}.

float x = 2., y - в цьому рядку оголошені змінні x, y речовинного типу і задано початкове значення змінної x, рівне числу 2, причому точка праворуч від двійки, не випадкова: вона явно вказує на тип числової константи - double.

y = x * x - ну ось і обчислення "двічі два".

3.9 Практична робота №4:"Оператори розгалуження та логічні вирази"

Оператор розгалуження (умовна інструкція, умовний оператор) - оператор, конструкція мови програмування, що забезпечує виконання певної команди (набору команд) тільки за умови істинності деякого логічного

вираження, або виконання однієї з декількох команд (наборів команд) залежно від значення деякого вираження.

Оператор розгалуження застосовується у випадках, коли виконання або невиконання деякого набору команд повинно залежати від виконання або невиконання деякої умови. Галуження - одна з трьох (поряд з послідовним виконанням команд і циклом) базових конструкцій структурного програмування.

При реалізації алгоритмів завжди виникають ситуації, коли який-небудь оператор або групу операторів треба виконати в разі виникнення певних умов, тоді як іншу групу при цих же умовах взагалі не слід виконувати. У мові Perl для організації подібного розгалуження в програмі передбачені оператори if, які ми і будемо називати операторами розгалуження,

Відмінною особливістю всіх цих операторів є те, що в них обчислюється деякий вираз, зване умовою, і залежно від його істинності чи хибності виконуються або не виконуються оператори деякого блоку. Це означає, що вирази умови у всіх операторах розгалуження обчислюються в булевому контексті.

Самая найпростіша форма оператора розгалуження if представлена наступного синтаксичної конструкцією:

if (ВИРАЗ) БЛОК

Її семантика така - якщо ВИРАЗ істинно, то виконуються оператори блоку БЛОК, в іншому випадку він просто пропускається, наприклад:

```
if ($ var == "print") {
   print "Змінна \ $ text = $ text";
}
```

Логічні вирази

Керуючі конструкції програмування. (Часто вживають термін "керуючі структури")

Якщо не передбачено іншого, оператори програми виконуються один за іншим, як вони написані. Але, в певних випадках, необхідно, щоб наступним виконувався не черговий оператор, а якийсь інший. Це називається передачею управління. Всі програми можуть бути написані з використанням всього 3 керуючих конструкцій:

конструкція проходження пряма послідовність виконання операторів з якою ви вже встигли познайомитися;

конструкція вибору хід виконання програми розгалужується на 1 або декілька гілок в залежності від істинності чи хибності контрольного умови, що слід перевірити.

конструкція повторення певна частина послідовності операторів повторюється, також залежно від виконання певного условія.Проверка умови входить в синтаксис керуючих конструкцій. Тому, перш ніж перейти до їх вивчення, розберемося з правильним написанням і перевіркою операторів порівняння і логічними виразами.

Умови та логічні вирази.

Підставою для прийняття рішень в керуючих конструкціях є умовні вирази це спосіб запису умов. Такі вирази, які можуть приймати значення, або БРЕХНЯ (false), або ІСТИНА (true). В умовних виразах використовуються оператори порівняння. Т.к. вони дещо різняться в різних мовах, про них трохи пізніше. Приклад поки з математичними знаками (у прикладі нижче НЕ використовуються програмні оператори порівняння!), Щоб можна було зрозуміти суть умовних виразів:

Нехай у вас є змінна x = 2. Вам потрібно перевірити вираз x> 0. В даному випадку x дійсно більше 0, тому його поточне значення 2. Отже вираз x> 0 вірно і дорівнює (вираз одно!) True. Якщо при тому ж значенні x вам доведеться перевірити вираз x <0, то даний вираз вже буде невірним, тому що x = 0. Значить значення другого виразу буде false.

Якщо треба перевірити кілька умов в одному вираженні, тобто вирішити складне логічне вираз, що складається з декількох простих (Прості ми розібрали в прикладі вище), застосовуються спеціальні оператори. Взагалі цей розділ математики прийнято називати булевой алгеброю на ім'я вченого, який все це справа розробив. Кому буде мало наявної тут таблиці-відсилаю до відповідного розділу математики. Отже, над умовними виразами можна виконувати дії логічної математики. Як користуватися таблицею: будь-яке складне логічне вираження, складене з декількох простих завжди можна звести до виразу з двох операндів, один з яких простий, а другий може бути простим, а може бути складним. Якщо продовжувати цей процес, то ми доберемося до вираження, що складається з 2 простих операндів.

-AND-операція логічне І чи логічне множення (кон'юнкція). І одне, й інше умовні вирази повинні бути істинними, щоб все складне вираз можна було вважати істиною.

-OR-операція логічне АБО або логічне додавання (диз'юнкція). Достатньо, щоб один з виразів було істинним, щоб все складне вираз було істинним.

-XOR операція виключає АБО. Звичайне АБО дає true, коли обидва операнда true, а даний варіант виключає за принципом або-або, але не обидва разом і дає false. Отже, якщо одне і тільки одне умовне вираз має значення ІСТИНА, то результат буде ІСТИНА. Якщо обидві умови ІСТИНА або обидва FALSE, то результат буде БРЕХНЯ.

-NOT-операція "логічне НЕ" або заперечення. Це операція з одним операндом. Якщо операнд є істинним, то все вираз-буде брехнею. І навпаки.

Завдання на логічні вираження

Нехай x = 3, y = -9. Обчислити значення виразів (тобто, true або false):

x = 3;
xy;
y / 2 = 4;

7 MOD 3 = 1 (додаткова інформація: MOD це оператор, який виконує операцію взяти залишок, тобто, при розподілі першого операнда на другий ми отримуємо залишок від цього поділу);

NOT y = -50;

1x AND x5;

x4 OR y-15;

x4 OR y-15

Чи є вираз x10 істинним, якщо

x = 0;

x = 2;

x = 10;

x = 5;

x = 15

Чи є вираз x = 10 хибним, якщо а) x = 1, б) x = 3, в) x = 10, р) x = 12; д) x = 25

Записати логічні вираження із застосуванням відповідних операторів мови для математичних нерівностей:

а) 0 = x10

б) 5x8

в) x = 1 або x9

г) x = 2, x12

д) x = 0 і y = 0

Обчислити логічне значення виразу, якщо x = 5, y = 12, z = -3

(x = 1 AND y2 AND z = 0) OR (x0 XOR y = 10)

Завдання на логічні вираження

Нехай x = 3, y = -9. Обчислити значення виразів (тобто, true або false):

x = 3;

xy;

y / 2 = 4;

7 MOD 3 = 1 (додаткова інформація: MOD це оператор, який виконує операцію взяти залишок, тобто, при розподілі першого операнда на другий ми отримуємо залишок від цього поділу);

NOT y = -50;

1x AND x5;

x4 OR y-15;

x4 OR y-15

Чи є вираз x10 істинним, якщо

x = 0;

x = 2;

x = 10;

x = 5;

x = 15

Чи є вираз x = 10 хибним, якщо а) x = 1, б) x = 3, в) x = 10, р) x = 12; д) x = 25

Записати логічні вираження із застосуванням відповідних операторів мови для математичних нерівностей:

а) 0 = x10

б) 5x8

в) x = 1 або x9

г) x = 2, x12

д) x = 0 і y = 0

Обчислити логічне значення виразу, якщо x = 5, y = 12, z = -3

(x = 1 AND y2 AND z = 0) OR (x0 XOR y = 10)

4. МОДУЛЬ 4: Функції. Рекурсія.

4.1 Функції. Визначення та оголошення.

Потужність мови СІ багато в чому визначається легкістю і гнучкістю у визначенні та використанні функцій в СІ-програмах. На відміну від інших мов програмування високого рівня в мові СІ немає поділу на процедури, підпрограми і функції, тут вся програма будується тільки з функцій.

Функція - це сукупність оголошень і операторів, зазвичай призначена для вирішення певної задачі. Кожна функція повинна мати ім'я, яке використовується для її оголошення, визначення і виклику. В будь-якій програмі на СІ повинна бути функція з ім'ям main (головна функція), саме з цієї функції, в якому б місці програми вона не перебувала, починається виконання програми.

При виконанні функції їй за допомогою аргументів (формальних параметрів) можуть бути передані деякі значення (фактичні параметри), які використовуються під час виконання функції. Функція може повертати деяке (одно!) Значення. Це повертається значення і є результат виконання функції, який при виконанні програми підставляється в точку виклику функції, де б цей виклик ні зустрівся. Допускається також використовувати функції не мають аргументів і функції не повертають ніяких значень. Дія таких функцій може полягати, наприклад, у зміні значень деяких змінних, виведення на друк деяких текстів і т.п..

З використанням функцій в мові СІ пов'язані три поняття - визначення функції (опис дій, які виконуються функцією), оголошення функції (завдання форми звертання до функції) і виклик функції.

Визначення функції задає тип значення, що повертається, ім'я функції, типи і кількість формальних параметрів, а також оголошення змінних і оператори, звані тілом функції, і визначають дію функції. У визначенні функції також може бути заданий клас пам'яті.

Приклад:

```
int rus (unsigned char r)
{If (r> = 'A' && c <= '')
    return 1;
  else
    return 0;
}
```

У даному прикладі визначена функція з ім'ям rus, що має один параметр з ім'ям r і типом unsigned char. Функція повертає ціле значення, рівне 1, якщо параметр функції є буквою російського алфавіту, або 0 в іншому випадку.

У мові СІ немає вимоги, щоб визначення функції обов'язково передувало її викликом. Визначення використовуваних функцій можуть слідувати за визначенням функції main, перед ним, або знаходиться в іншому файлі.

Однак для того, щоб компілятор міг здійснити перевірку відповідності типів переданих фактичних параметрів типам формальних параметрів до виклику функції потрібно помістити оголошення (прототип) функції.

Оголошення функції має такий же вигляд, що і визначення функції, з тією лише різницею, що тіло функції відсутній, і імена формальних параметрів теж можуть бути опущені. Для функції, визначеної в останньому прикладі, прототип може мати вигляд

int rus (unsigned char r); або rus (unsigned char);

У програмах на мові СІ широко використовуються, так звані, бібліотечні функції, тобто функції попередньо розроблені і записані в бібліотеки. Прототипи бібліотечних функцій знаходяться в спеціальних заголовних файлах, що поставляються разом з бібліотеками в складі систем програмування, і включаються в програму за допомогою директиви # include.

Якщо оголошення функції не задано, то за замовчуванням будується прототип функції на основі аналізу першого посилання на функцію, будь то виклик функції або визначення. Однак такий прототип не завжди узгоджується з наступним визначенням чи викликом функції. Рекомендується завжди задавати прототип функції. Це дозволить компілятору або видавати діагностичні повідомлення, при неправильному використанні функції, або коректним чином регулювати невідповідність аргументів встановлюється при виконанні програми.

Оголошення параметрів функції при її визначенні може бути виконано в так званому "старому стилі", при якому в дужках після імені функції слідують тільки імена параметрів, а після дужок оголошення типів параметрів. Наприклад, функція rus з попереднього прикладу може бути визначена наступним чином:

```
int rus (r)
unsigned char r;
{... / * Тіло функції * / ... }
```

Відповідно до синтаксисом мови СІ визначення функції має наступну форму:

```
[Специфікатор-класу-пам'яті] [специфікатор-типу] ім'я-функції
([Список-формальних-параметрів])
{Тіло-функції}
```

Необов'язковий специфікатор-класу-пам'яті задає клас пам'яті функції, який може бути static або extern. Детально класи пам'яті будуть розглянуті в наступному розділі.

Специфікатор-типу функції задає тип значення, що повертається і може задавати будь-який тип. Якщо специфікатор-типу не заданий, то передбачається, що функція повертає значення типу int.

Функція не може повертати масив або функцію, але може повертати покажчик на будь-який тип, в тому числі і на масив і на функцію. Тип значення, що повертається, що задається у визначенні функції, повинен відповідати типу в оголошенні цієї функції.

Функція повертає значення якщо її виконання закінчується оператором return, що містить деякий вираз. Зазначене вираз обчислюється, перетвориться, якщо необхідно, до типу значення, що повертається і повертається в точку виклику функції в якості результату. Якщо оператор return не містить висловлювання або виконання функції завершується після виконання останнього її оператора (без виконання оператора return), то повертається значення не визначено. Для функцій, що не використовують повертається значення, повинен бути використаний тип void, який вказує на відсутність значення, що повертається. Якщо функція визначена як функція, що повертає деяке значення, а в операторі return при виході з неї відсутній вираз, то поведінка викликає функції після передачі їй управління може бути непередбачуваним.

Список-формальних-параметрів - це послідовність оголошень формальних параметрів, розділена комами. Формальні параметри - це змінні, використовувані усередині тіла функції і отримують значення при виконанні функції шляхом копіювання в них значень відповідних фактичних параметрів. Список-формальних-параметрів може закінчуватися комою (,) або комою з трьома крапками (, ...), це означає, що число аргументів функції змінно. Однак передбачається, що функція має, принаймні, стільки обов'язкових аргументів, скільки формальних параметрів задано перед останньої коми в списку параметрів. Такий функції може бути передано більше число аргументів, але над додатковими аргументами не проводиться контроль типів.

Якщо функція не використовує параметрів, то наявність круглих дужок обов'язково, а замість списку параметрів рекомендується вказати слово void.

Порядок і типи формальних параметрів повинні бути однаковими у визначенні функції і у всіх її оголошеннях. Типи фактичних параметрів при виклику функції повинні бути сумісні з типами відповідних формальних параметрів. Тип формального параметра може бути будь-яким основним типом, структурою, об'єднанням, перерахуванням, покажчиком або масивом. Якщо тип формального параметра не зазначений, то цим параметром присвоюється тип int.

Для формального параметра можна задавати клас пам'яті register, при цьому для величин типу int специфікатор типу можна опустити.

Ідентифікатори формальних параметрів використовуються в тілі функції в якості посилань на передані значення. Ці ідентифікатори не можуть бути перевизначені в блоці, утворюючому тіло функції, але можуть бути перевизначені у внутрішньому блоці всередині тіла функції.

При передачі параметрів у функцію, якщо необхідно, виконуються звичайні арифметичні перетворення для кожного формального параметра і кожного фактичного параметра незалежно. Після перетворення формальний параметр не може бути коротше ніж int, тобто оголошення формального параметра з типом char рівносильно його оголошенню з типом int. А параметри, які становлять дійсні числа, мають тип double.

Перетворений тип кожного формального параметра визначає, як інтерпретуються аргументи, що поміщаються при виконанні функції в стек. Невідповідність типів фактичних аргументів і формальних параметрів може бути причиною невірної інтерпретації.

Тіло функції - це складовою оператор, що містить оператори, що визначають дію функції.

Всі змінні, оголошені в тілі функції без вказівки класу пам'яті, мають клас пам'яті auto, тобто вони є локальними. При виконанні функції локальним змінним відводиться пам'ять в стеку і проводиться їх ініціалізація. Управління передається першому оператору тіла функції і починається виконання функції, яке триває до тих пір, поки не зустрінеться оператор return або останній оператор тіла функції. Управління при цьому повертається в точку, наступну за точкою виклику, а локальні змінні стають недоступними. При новому виклику функції для локальних змінних пам'ять розподіляється знову, і тому старі значення локальних змінних втрачаються.

Параметри функції передаються за значенням і можуть розглядатися як локальні змінні, для яких виділяється пам'ять при виконанні функції і проводиться ініціалізація значеннями фактичних параметрів. При виході з функції значення цих змінних втрачаються. Оскільки передача параметрів відбувається за значенням, в тілі функції не можна змінити значення змінних в викликає функції, які є фактичними параметрами. Однак, якщо в якості параметра передати покажчик на деяку змінну, то використовуючи операцію разадресаціі можна змінити значення цієї змінної.

Приклад:

```
/ * Неправильне використання параметрів * /
void change (int x, int y)
{Int k = x;
        x = y;
        y = k;
}
```

У даній функції значення змінних x і y, які є формальними параметрами, міняються місцями, але оскільки ці змінні існують тільки усередині функції change, значення фактичних параметрів, використовуваних при виконанні

функції, залишаться незмінними. Для того щоб мінялися місцями значення фактичних аргументів можна використовувати функцію наведену в наступному прикладі.

Приклад:

```
/ * Правильне використання параметрів * /
 void change (int * x, int * y)
 {Int k = * x;
      * X = * y;
      * Y = k;
 }
```

При виклику такої функції в якості фактичних параметрів повинні бути використані не значення змінних, а їх адреси

```
change (& a, & b);
```

Якщо потрібно викликати функцію до її визначення в розглянутому файлі, або визначення функції знаходиться в іншому вихідному файлі, то виклик функції слід випереджати оголошенням цієї функції. Оголошення (прототип) функції має наступний формат:

[Специфікатор-класу-пам'яті] [специфікатор-типу] ім'я-функції ([список-формальних-параметрів]) [, список-імен-функцій];

На відміну від визначення функції, в прототипі за заголовком відразу ж слід крапка з комою, а тіло функції відсутня. Якщо кілька різних функцій повертають значення однакового типу і мають однакові списки формальних параметрів, то ці функції можна оголосити в одному прототипі, вказавши ім'я однієї з функцій в якості імені-функції, а всі інші помістити в список-імен-функцій, причому кожна функція повинна супроводжуватися списком формальних параметрів. Правила використання інших елементів формату такі ж, як при визначенні функції. Імена формальних параметрів при оголошенні функції можна не вказувати, а якщо вони вказані, то їх область дії поширюється лише до кінця оголошення.

Прототип - це явне оголошення функції, що передує визначенню функції. Тип значення, що повертається при оголошенні функції повинен відповідати типу значення, що повертається у визначенні функції.

Якщо прототип функції не заданий, а зустрівся виклик функції, то будується неявний прототип з аналізу форми виклику функції. Тип значення, що повертається створюваного прототипу int, а список типів і кількості параметрів функції формується на підставі типів і кількості фактичних параметрів використовуваних при цьому виклику.

Таким чином, прототип функції необхідно задавати в наступних випадках:

1. Функція повертає значення типу, відмінного від int.

2. Потрібен проініціалізувати деякий покажчик на функцію до того, як ця функція буде визначена.

Наявність в прототипі повного списку типів аргументів параметрів дозволяє виконати перевірку відповідності типів фактичних параметрів при виклику функції типам формальних параметрів, і, якщо необхідно, виконати відповідні перетворення.

В прототипі можна вказати, що число параметрів функції змінно, або що функція не має параметрів.

Якщо прототип заданий з класом пам'яті static, то і визначення функції повинно мати клас пам'яті static. Якщо специфікатор класу пам'яті не вказаний, то мається на увазі клас пам'яті extern.

Виклик функції має наступний формат:

адресний-вираз ([список-виразів])

Оскільки синтаксично ім'я функції є адресою початку тіла функції, як звернення до функції може бути використано адресний-вираз (в тому числі і ім'я функції або разадресація покажчика на функцію), що має значення адреси функції.

Список-виразів являє собою список фактичних параметрів, переданих у функцію. Цей список може бути і порожнім, але наявність круглих дужок обов'язково.

Фактичний параметр може бути величиною будь-якого основного типу, структурою, об'єднанням, перерахуванням або покажчиком на об'єкт будь-якого типу. Масив і функція не можуть бути використані в якості фактичних параметрів, але можна використовувати покажчики на ці об'єкти.

Виконання виклику функції відбувається наступним чином:

1. Обчислюються вираження в списку виразів і піддаються звичайним арифметичним перетворенням. Потім, якщо відомий прототип функції, тип отриманого фактичного аргументу порівнюється з типом відповідного формального параметра. Якщо вони не співпадають, то або виробляється перетворення типів, або формується повідомлення про помилку. Число виразів в списку виразів має збігатися з числом формальних параметрів, якщо тільки функція не має змінного числа параметрів. В останньому випадку перевірці підлягають тільки обов'язкові параметри. Якщо в прототипі функції зазначено, що їй не потрібні параметри, а при виклику вони вказані, формується повідомлення про помилку.

2. Відбувається присвоювання значень фактичних параметрів відповідним формальним параметрам.

3. Управління передається на перший оператор функції.

4. Виконання оператора return в тілі функції повертає управління і можливо, значення в зухвалу функцію. При відсутності оператора return управління повертається після виконання останнього оператора тіла функції, а повертається значення не визначено.

Адресний вираз, що стоїть перед дужками визначає адресу викликається функції. Це означає що функція може бути викликана через покажчик на функцію.

Приклад:

```
int (* fun) (int x, int * y);
```

Тут оголошена змінна fun як покажчик на функцію з двома параметрами: типу int і покажчиком на int. Сама функція повинна повертати значення типу int. Круглі дужки, що містять ім'я покажчика fun і ознака покажчика *, обов'язкові, інакше запис

```
int * fun (intx, int * y);
```

буде інтерпретуватися як оголошення функції fun повертає покажчик на int.

Виклик функції можливий тільки після ініціалізації значення покажчика fun і має вигляд:

```
(* Fun) (i, & j);
```

У цьому виразі для отримання адреси функції, на яку посилається покажчик fun використовується операція разадресациі *.

Покажчик на функцію може бути переданий в якості параметра функції. При цьому разадресація відбувається під час виклику функції, на яку посилається покажчик на функцію. Присвоїти значення вказівником на функцію можна в операторі присвоювання, вживши ім'я функції без списку параметрів.

Приклад:

```
   double (* fun1) (int x, int y);
   double fun2 (int k, int l);
     fun1 = fun2; / * ініціалізація покажчика на функцію * /
     (* Fun1) (2,7); / * звернення до функції * /
```

У розглянутому прикладі покажчик на функцію fun1 описаний як покажчик на функцію з двома параметрами, яка повертає значення типу double, і також описана функція fun2. В іншому випадку, тобто коли вказівником на функцію присвоюється функція описана інакше ніж покажчик, відбудеться помилка.

Розглянемо приклад використання покажчика на функцію як параметр функції обчислює похідну від функції cos (x).

Приклад:

```
double proiz (double x, double dx, double (* f) (double x));
double fun (double z);
int main ()
{
 double x; / * точка обчислення похідної * /
 double dx; / * прирощення * /
 double z; / * значення похідної * /
 scanf ("% f,% f", & x, & dx); / * введення значень x і dx * /
 z = proiz (x, dx, fun); / * виклик функції * /
 printf ("% f", z); / * друк значення похідної * /
 return 0;
}
double proiz (double x, double dx, double (* f) (double z))
{/ * Функція обчислює похідну * /
  double xk, xk1, pr;
  xk = fun (x);
  xk1 = fun (x + dx);
  pr = (xk1/xk-1e0) * xk / dx;
  return pr;
}
double fun (double z)
{/ * Функція від якої обчислюється похідна * /
  return (cos (z));
}
```

Для обчислення похідної від будь-якої іншої функції можна змінити тіло функції fun або використовувати при виконанні функції proiz ім'я іншої функції. Зокрема, для обчислення похідної від функції cos (x) можна викликати функцію proiz у формі

```
z = proiz (x, dx, cos);
```

а для обчислення похідної від функції sin (x) у формі

```
z = proiz (x, dx, sin);
```

Будь-яка функція в програмі на мові СІ може бути викликана рекурсивно, тобто вона може викликати саму себе. Компілятор допускає будь-яке число рекурсивних викликів.

При кожному виклику для формальних параметрів і змінних з класом пам'яті auto і register виділяється нова область пам'яті, так що їх значення з попередніх викликів не втрачаються, але в кожен момент часу доступні тільки значення поточного дзвінка.

Змінні, оголошені з класом пам'яті static, не вимагають виділення нової області пам'яті при кожному рекурсивному виклику функції та їх значення доступні протягом усього часу виконання програми.

Класичний приклад рекурсії - це математичне визначення факторіала n! :

n! = 1 при n = 0;

n * (n-1)! при n> 1.

Функція, що обчислює факторіал, матиме наступний вигляд:

```
long fakt (int n)
{
 return ((n == 1)? 1: n * fakt (n-1));
}
```

Хоча компілятор мови CI не обмежує число рекурсивних викликів функцій, це число обмежується ресурсом пам'яті комп'ютера і при дуже великому числі рекурсивних викликів може статися переповнення стека.

4.2 Виклик Функції. Структура.

Виклик функції.Структура.

При виконанні функції зі змінним числом параметрів у виклику цієї функції задається будь-яке потрібне число аргументів. В оголошенні й визначенні такої функції змінне число аргументів задається трьома крапками в кінці списку формальних параметрів або списку типів аргументів.

Всі аргументи, задані у виклику функції, розміщуються в стеку. Кількість формальних параметрів, оголошених для функції, визначається числом аргументів, які беруться з стека і присвоюються формальним параметрам. Програміст відповідає за правильність вибору додаткових аргументів з стека і визначення числа аргументів, що знаходяться в стеку.

Прикладами функцій зі змінним числом параметрів є функції з бібліотеки функцій мови CI, що здійснюють операції введення-виведення інформації (printf, scanf і т.п.). Докладно ці функції розглянуті під третій частині книги.

Програміст може розробляти свої функції зі змінним числом параметрів. Для забезпечення зручного способу доступу до аргументів функції зі змінним числом параметрів є три макровизначення (макроси) va_start, va_arg, va_end, що знаходяться в заголовному файлі stdarg.h. Ці макроси вказують на те, що функція, розроблена користувачем, має деяке число обов'язкових аргументів, за якими слід змінне число необов'язкових аргументів. Обов'язкові аргументи доступні через свої імена як при виклику звичайної функції. Для витягання необов'язкових аргументів використовуються макроси va_start, va_arg, va_end в наступному порядку.

Макрос va_start призначений для установки аргументу arg_ptr на початок списку необов'язкових параметрів і має вигляд функції з двома параметрами:

void va_start (arg_ptr, prav_param);

Параметр prav_param повинен бути останнім обов'язковим параметром викликається функції, а покажчик arg_prt повинен бути оголошений з приреченням в списку змінних типу va_list у вигляді:

va_list arg_ptr;

Макрос va_start повинен бути використаний до першого використання макросу va_arg.

Макрокоманда va_arg забезпечує доступ до поточного параметру викликається функції і теж має вигляд функції з двома параметрами

type_arg va_arg (arg_ptr, type);

Ця макрокоманда витягує значення типу type за адресою, заданому покажчиком arg_ptr, збільшує значення покажчика arg_ptr на довжину використаного параметра (довжина type) і таким чином параметр arg_ptr буде вказувати на наступний параметр викликається функції. Макрокоманда va_arg використовується стільки разів, скільки необхідно для вилучення всіх параметрів викликається функції.

Макрос va_end використовується по закінченні обробки всіх параметрів функції і встановлює покажчик списку необов'язкових параметрів на нуль (NULL).

Розглянемо застосування цих макросів для обробки параметрів функції обчислює середнє значення довільній послідовності цілих чисел. Оскільки функція має змінне число параметрів будемо вважати кінцем списку значення рівне -1. Оскільки в списку має бути хоча б один елемент, у функції буде один обов'язковий параметр.

Приклад:

```
  # Include
  int main ()
  {Int n;
   int sred_znach (int, ...);
   n = sred_znach (2,3,4, -1);
                / * Виклик з чотирма параметрами * /
   printf ("n =% d", n);
   n = sred_znach (5,6,7,8,9, -1);
              / * Виклик з шістьма параметрами * /
   printf ("n =% d", n);
   return (0);
  }
  int sred_znach (int x, ...);
```

```
{
  int i = 0, j = 0, sum = 0;
  va_list uk_arg;
  va_start (uk_arg, x); / * установка вказівника uk_arg на * /
                  / * Перший необязятельной параметр * /
  if (x! = -1) sum = x; / * перевірка на пустоту списку * /
  else return (0);
  j + +;
  while ((i = va_arg (uk_arg, int))! = -1)
                        / * Вибірка чергового * /
  {/ * Параметра і перевірка * /
   sum + = i; / * на кінець списку * /
   j + +;
  }
  va_end (uk_arg); / * закриття списку параметрів * /
  return (sum / j);
}
```

4.3 Практична робота № 5: "Складання програм за допомогою функції"

Ця програма демонструє використання рекурсивної функції для обчислення факторіала. (Зазначимо, що визначення функції factorial () може знаходитися і після функції main (), але в цьому випадку функція factorial () повинна бути оголошена перед функцією main (), тобто до main () необхідно помістити рядок: long factorial (int);.)

```
/ * Приклад 4 * /
# Include <stdio.h>
# Include <values.h>
# Include <process.h>

long factorial (int value) / * Рекурсивна функція * /
{
  long result = 1;

  if (value! = 0)
  {
    result = factorial (value - 1);
   / * Перевірка можливості обчислення факторіала * /
    if (result> MAXLONG / (value + 1))
    {
      fprintf (stderr, "Дуже велике число \ n");
      getch (); / * Очікування натискання клавіші * /
      exit (1);
```

```
        }
        result * = value;
    }
    return (result);
}
/ * Рекурсивне обчислення факторіала числа value * /
void main (void)
{
    int value; / * Факторіал цього значення обчислюється * /
    long result; / * Змінна для результату * /
    puts ("Факторіал якого числа?");
    scanf ("% d", & value);
    result = factorial (value);
    printf ("Результат:% ld \ n", result);
    getch (); / * Очікування натискання клавіші * /
}
```

Результати роботи цієї програми:
Факторіал якого числа? 10 <Enter>
Результат: 362880

4.4 Функція main (void)Функція gets( ) Функція puts( )

Функція main ()

Функція main, з якої починається виконання СІ-програми, може бути визначена з параметрами, які передаються з зовнішнього оточення, наприклад, з командного рядка. У зовнішньому оточенні діють свої правила подання даних, а точніше, всі дані представляються у вигляді рядків символів. Для передачі цих рядків у функцію main використовуються два параметри, перший параметр служить для передачі числа переданих рядків, другий для передачі самих рядків. Загальноприйняті (але не обов'язкові) імена цих параметрів argc і argv. Параметр argc має тип int, його значення формується з аналізу командного рядка і дорівнює кількості слів у командному рядку, включаючи і ім'я програми (під словом розуміється будь-який текст не містить символу пробіл). Параметр argv це масив покажчиків на рядки, кожна з яких містить одне слово з командного рядка. Якщо слово повинно містити символ пробіл, то при записі його в командний рядок воно має бути укладена в лапки.

Функція main може мати і третій параметр, який прийнято називати argp, і який служить для передачі у функцію main параметрів операційної системи (середовища) в якій виконується СІ-програма.

Заголовок функції main має вигляд:

int main (int argc, char * argv [], char * argp [])

Якщо, наприклад, командний рядок CI-програми має вигляд:

A: \> cprog working 'C program' 1

то аргументи argc, argv, argp представляються в пам'яті

```
argc [4]
argv [] -> [] -> [A: \ cprog.exe \ 0]
           [] -> [Working \ 0]
           [] -> [C program \ 0]
           [] -> [1 \ 0]
           [NULL]
argp [] -> [] -> [path = A: \; C: \ \ 0]
           [] -> [Lib = D: \ LIB \ 0]
           [] -> [Include = D: \ INCLUDE \ 0]
           [] -> [Conspec = C: \ COMMAND.COM \]
           [NULL]
```

Операційна система підтримує передачу значень для параметрів argc, argv, argp, а на користувача лежить відповідальність за передачу і використання фактичних аргументів функції main.

Наступний приклад представляє програму друку фактичних аргументів, переданих у функцію main з операційної системи і параметрів операційної системи.

Приклад:

```
int main (int argc, char * argv [], char * argp [])
{Int i = 0;
  printf ("\ n Ім'я програми% s", argv [0]);
  for (i = 1; i> = argc; i + +)
  printf ("\ n аргумент% d дорівнює% s", argv [i]);
  printf ("\ n Параметри операційної системи:");
  while (* argp)
    {Printf ("\ n% s", * argp);
      argp + +;
    }
  return (0);
}
```

Доступ до параметрів операційної системи можна також отримати за допомогою бібліотечної функції geteuv, її прототип має наступний вигляд:

char * geteuv (const char * varname);

Аргумент цієї функції задає ім'я параметра середовища, покажчик на значення якої видасть функція geteuv. Якщо зазначений параметр не визначений в середовищі в даний момент, то повертається значення NULL.

Використовуючи покажчик, отриманий функцією geteuv, можна тільки прочитати значення параметра операційної системи, але не можна його змінити. Для зміни значення параметра системи призначена функція puteuv.

Компілятор мови CI будує CI-програму таким чином, що спочатку роботи програми виконується деяка ініціалізація, що включає, крім усього іншого, обробку аргументів, переданих функції main, і передачу їй значень параметрів середовища. Ці дії виконуються бібліотечними функціями _setargv і _seteuv, які завжди містяться компілятором перед функцією main.

Якщо CI-програма не використовує передачу аргументів і значень параметрів операційної системи, то доцільно заборонити використання бібліотечних функцій _setargv і _seteuv помістивши в CI-програму перед функцією main функції з такими ж іменами, але не виконують ніяких дій (заглушки). Початок програми в цьому випадку буде мати вигляд:

```
_setargv ()
   {Return; / * порожня функція * /
   }
  -Seteuv ()
   {Return; / * порожня функція * /
   }
  int main ()
   {/ * Головна функція без аргументів * /
   ...
   ...
   renurn (0);
   }
```

У наведеній програмі при виклику бібліотечних функцій _setargv і _seteuv будуть використані функції поміщені в програму користувачем і не виконують жодних дій. Це помітно знизить розмір одержуваного exe-файла.

Функція gets( )

Функція gets (), що входить до складу стандартної бібліотеки C, має наступний прототип:

```
char * gets (char * s);
```

Це визначення міститься в stdio.h. Функція призначена для введення рядка символів з файлу stdin. Вона повертає s якщо читання пройшло успішно і NULL в зворотному випадку.

При всій простоті і зрозумілості, ця функція є унікальною. Вся справа в тому, що більш небезпечного виклику, ніж цей, у стандартній бібліотеці немає ... чому це так, а також чим загрожує використання gets (), я якраз і спробую пояснити в сьогоднішній замітці.

Взагалі кажучи, для тих, хто не знає, чому використання функції gets () так небезпечно, буде корисно подивитися ще раз на її прототип, і подумати. Якщо здогадаєтеся самостійно, буде зайвий привід трохи погорду ;)

Вся справа в тому, що для gets () не можна, тобто абсолютно неможливо, задати обмеження на розмір читається рядки, в усякому разі, в межах стандартної бібліотеки. Це вкрай небезпечно, тому що тоді при роботі з вашою програмою можуть виникати різні збої при звичайному введенні рядків користувачами. Тобто, наприклад:

```
char name [10];

/ / ...

puts ("Enter you name:");
gets (name);
```

Якщо у користувача буде ім'я більше, ніж 9 символів, наприклад, 10, то за адресою (name + 10) буде записаний 0. Що там насправді знаходиться, інші дані або просто незайняте місце (що виникло, наприклад, через те, що компілятор відповідним чином вирівняв дані), або ця адреса для програми недоступний, невідомо.

Всі ці ситуації нічого доброго не обіцяють. Псування власних даних означає те, що програма видасть невірні результати, а чому це відбувається зрозуміти буде вкрай важко --- насамперед програміст буде перевіряти помилки в алгоритмі і тільки в кінці помітить, що сталося переповнення внутрішнього буфера. Я думаю, всі знають як це відбувається --- кілька годин безперервних "чувань" з відладчиком, а потім через день, "на свіжу голову", з'ясовується що десь був пропущений один символ ...

Знову ж таки, для програміста найзручнішим буде моментальне аварійне припинення роботи програми в цьому місці --- тоді він зможе замінити gets () на щось більш "порядне".

У когось може виникнути пропозиція просто взяти і збільшити розмір буфера. Але не треба забувати, що завжди можна ввести рядок довжиною, що перевищує виділений розмір; якщо хтось хоче заперечити, що випадки імен довжиною більш ніж, наприклад, 1024 байта все ще рідкісні, то я перейду до іншого, трохи більш цікавого прикладу виникає проблеми при використанні gets ().

```
void foo ()
{
 char name [10];

 / / ...
```

```
  puts ("Enter you name:");
  gets (name);

  / / ...
}
```

Функція puts( )

```
# include <stdio.h>
int puts (const char * str);
```

Функція puts () записує рядок, що адресується параметром str, в стандартне вихідний пристрій. Символ кінця рядка перетвориться в роздільник рядків.

При успішному виконанні функція puts () повертає невід'ємне значення, а в разі збою - значення EOF.

Наступна програма записує в стандартний потік виводу stdout рядок:

це приклад

```
# include <stdio.h>
# include <string.h>

int main (void)
{
  char str [80];

  strcpy (str, "це приклад");

  puts (str);

  return 0;
}
```

залежні функції

putc () gets () printf ()

4.5 Функції повертаючі значення

В тілі функції може зустрітися інструкція return. Вона завершує виконання функції. Після цього управління повертається тієї функції, з якої була викликана дана. Інструкція return може вживатися у двох формах:

```
return;
return expression;
```

Перша форма використовується у функціях, для яких типом значення, що повертається є void. Використовувати return в таких випадках обов'язково, якщо потрібно примусово завершити роботу.Після кінцевої інструкції функції мається на увазі наявність return. Наприклад:

```
void d_copy (double "src, double * dst, int sz)
{
   / * Копіюємо масив "src" в "dst"
   * Для простоти припускаємо, що вони одного розміру
   * /

   / / Завершення, якщо хоча б один з покажчиків дорівнює 0
   if (! src | |! dst)
      return;

   / / Завершення,
   / / Якщо покажчики адресують один і той же масив
   if (src == dst)
      return;

   / / Копіювати нічого
   if (sz == 0)
      return;

   / / Все ще не закінчили?
   / / Тоді саме час щось зробити
   for (int ix = 0; ix <sz; + + ix)
      dst [ix] = src [ix];

   / / Явного завершення не потрібно
}
```

У другій формі інструкції return вказується те значення, яке функція повинна повернути. Це значення може бути скільки завгодно складним виразом, навіть містити виклик функції. У реалізації функції factorial (), яку ми розглянемо в наступному розділі, використовується return наступного вигляду:

```
return val * factorial (val-1);
```

У функції, не оголошена з void як тип значення, що повертається, обов'язково використовувати другу форму return, інакше станеться помилка компіляції. Хоча компілятор не відповідає за правильність результату, він зможе гарантувати його наявність. Наступна програма не компілюється через двох місць, де програма завершується без повернення значення:

```
/ / Визначення інтерфейсу класу Matrix
# Include "Matrix.h"

bool is_equa1 (const Matrix & ml, const Matrix & m2)
{
   / * Якщо вміст двох об'єктів Matrix однаково,
   * Повертаємо true;
   * В іншому випадку - false
   * /

   / / Порівняємо кількість шпальт
   if (ml.colSize ()! = m2.co1Size ())
      / / Помилка: немає значення, що повертається
      return;

   / / Порівняємо кількість рядків
   if (ml.rowSize ()! = m2.rowSize ())
      / / Помилка: немає значення, що повертається
      return;

   / / Пробіжимося по обох матриць, поки
   / / Не знайдемо нерівні елементи
   for (int row = 0; row <ml.rowSize (); + + row)
      for (int col = 0; co1 <ml.colSize (); + + co1)
        if (ml [row] [col]! = m2 [row] [col])
            return false;
   / / Помилка: немає значення, що повертається
   / / Для випадку рівності
}
```

Якщо тип значення, що повертається не точно відповідає вказаному в оголошенні функції, то застосовується неявне перетворення типів. Якщо ж стандартне приведення неможливо, відбувається помилка компіляції.

За замовчуванням повертається значення передається за значенням, тобто викликає функція отримує копію результату обчислення виразу, вказаного в інструкції return. Наприклад:

```
Matrix grow (Matrix * p) {
  Matrix val;
  / / ...
  return val;
}
```

grow () повертає викликає функції копію значення, що зберігається у змінній val.

Таку поведінку можна змінити, якщо оголосити, що повертається покажчик або посилання. При поверненні посилання викликає функція отримує l-значення для val і тому може модифікувати val або взяти її адресу. Ось як можна оголосити, що grow () повертає посилання:

```
Matrix & grow (Matrix * p) {
  Matrix * res;
  / / Виділимо пам'ять для об'єкта Matrix
  / / Великого розміру
  / / Res адресує цей новий об'єкт
  / / Скопіюємо вміст * p в * res
  return * res;
}
```

Якщо повертається великий об'єкт, то набагато ефективніше перейти від повернення за значенням до використання посилання або покажчика. В деяких випадках компілятор може зробити це автоматично. Така оптимізація отримала назву іменоване повертається значення.

Оголошуючи функцію як повертає посилання, програміст повинен пам'ятати про дві можливі помилки:

• повернення посилання на локальний об'єкт, час життя якого обмежена часом виконання функції.По завершенні функції такому посиланню відповідає область пам'яті, що містить невизначене значення. Наприклад:

```
/ / Помилка: повернення посилання на локальний об'єкт
Matrix & add (Matrix & m1, Matrix & m2)
{
  Matrix result:
  if (m1.isZero ())
    return m2;
  if (m2.isZero ())
    return m1;
/ / Складемо вміст двох матриць
/ / Помилка: посилання на сумнівну область пам'яті
/ / Після повернення
  return result;
}
```

В такому випадку тип повернення не повинен бути посиланням. Тоді локальна змінна може бути скопійована до закінчення часу свого життя:

```
Matrix add (...)
```

• функція повертає l-значення. Будь-яка його модифікація зачіпає сам об'єкт. Наприклад:

```
# Include <vector>
```

```
int & get_val (vector <int> & vi, int ix) {
 return vi [ix];
 }

int ai [4] = {0, 1, 2, 3};
 vector <int> vec (ai, ai +4); / / копіюємо 4 елементи ai в vec

int main () {
 / / Збільшує vec [0] на 1
 get_val (vec.0) + +;
 / / ...
 }
```

Для запобігання трагічної модифікації повернутого об'єкта потрібно оголосити тип повернення як const:

```
const int & get_val (...)
```

Прикладом ситуації, коли l-значення повертається навмисно, щоб дозволити модифіковані реальний об'єкт, може служити перевантажений оператор взяття індексу для класу IntArray.

4.6 Функції з параметрами

Функції можна запускати з будь-яким числом параметрів.

Якщо функції передано менше параметрів, ніж є у визначенні, то відсутні вважаються undefined.

Наступна функція повертає час time, необхідний на подолання дистанції distance з рівномірною швидкістю speed.

При першому запуску функція працює з аргументами distance = 10, speed = undefined. Зазвичай така ситуація, якщо вона підтримується функцією, передбачає значення за замовчуванням :/ / якщо speed - помилкове значення (undefined, 0, false ...) - підставити 10

```
speed = speed | | 10
```

Оператор | | в яваскрипт повертає не true / false, а саме значення (перше, яке приводиться до true).

Тому його використовують для завдання значень за замовчуванням. У нашому виклику speed буде обчислено як undefined | | 10 = 10.

Тому результат буде 10/10 = 1.

Другий запуск - стандартний.

Третій запуск задає кілька додаткових аргументів. У функції не передбачена робота з додатковими аргументами, тому вони просто ігноруються.

Ну і в останньому випадку аргументів взагалі немає, тому distance = undefined, і маємо результат ділення undefined/10 = NaN (Not-A-Number, сталася помилка).

Робота з невизначеним числом параметрів

Безпосередньо перед входом в тіло функції, автоматично створюється об'єкт arguments, який містить

1.Аргументи виклику, починаючи від нуля

2.Довжину у властивості length

3.Посилання на саму функцію у властивості callee

Властивість arguments схоже на масив, т.к у нього є довжина і числові індекси. Насправді arguments не належить класу Array і не містить його методів, таких як push, pop і інших.

4.7 Формальні та фактичні параметри

Формальні параметри - дані, з якими працює підпрограма (ПП). Це внутрішні дані для ПП. Вони перераховуються в заголовку ПП і пов'язані з фактичними параметрами. Фактичні параметри - передача в ПП і повертаються з неї. Це зовнішні для ПП дані, з якими має справу викликає частина програми. У ПП їм відповідають формальні параметри. Вказуються в списку фактичних параметрів при зверненні до ПП. Синоніми:

Параметри = формальні параметри.

Аргументи = фактичні параметри.

Всі формальні параметри можна розбити на чотири категорії:

-параметри-значення;

-параметри-змінні;

-параметри-константи (використовуються тільки у версії 7.0);

-параметри-процедури і параметри-функції.

Для кожного формального параметра слід вказати ім'я і, як правило, тип, а в разі параметра-змінної або параметра-константи - його категорію. Імена параметрів можуть бути будь-якими, в тому числі і збігатися з іменами об'єктів програми. Необхідно лише пам'ятати, що в цьому випадку об'єкт основної програми з таким ім'ям стає недоступним для безпосереднього використання підпрограмою. Тип формального параметра може бути практично будь-яким, проте в заголовку підпрограми можна вводити новий тип. Наприклад, не можна пісатьfunction Max (A: array [1 .. 100] of real): real;

Щоб правильно записати цей заголовок, слід в основній програмі ввести тип-масив, а потім використовувати його в заголовку: type tArr = array [1 .. 100] of real;

function Max (A: tArr): real;

При зверненні до підпрограми формальні параметри замінюються відповідними фактичними викликає програмою або підпрограмою.

Для формальних і фактичних параметрів необхідно дотримуватися відповідності:

-однакову кількість,

-однаковий порядок проходження,

-сумісність типів з присвоєння.

Категорії формальних параметрів

Параметри-значення (у списку формальних параметрів не мають атрибута). Передаються ПП через стек, як копії, і не змінюються нею.

Параметри-змінні (у списку формальних параметрів мають атрибут var). Передаються ПП через адреси, і можуть змінюватися нею. Використовуються для повернення результатів в зухвалу програму.

Вихідні параметри-змінні (у списку формальних параметрів мають атрибут out). Подібні параметрам-змінним, але передаються тільки з ПП в програму.

Параметри-константи (у списку формальних параметрів мають атрибут const). Передаються ПП через адреси, але не можуть змінюватися нею. Застосовуються замість параметрів-значень, коли стек може переповнюватися.

Процедури або функції. Це параметри процедурного типу. Задаються іменами.

Параметри без типу. За правильність використання відповідає програміст.

Замовчувані параметри. Їх у списку фактичних параметрів можна пропускати. У списку формальних параметрів для них потрібно задати тип і значення.

4.8 Рекурсивні функції

Ситуацію, коли функція тим чи іншим чином викликає саму себе, називають рекурсією. Рекурсія, коли функція звертається сама до себе безпосередньо, називається прямий, інакше вона називається непрямою.

Всі функції мови С + + (крім функції main) можуть бути використані для побудови рекурсії.

В рекурсивної функції обов'язково повинне бути присутнім хоча б одна умова, при виконанні якого послідовність рекурсивних викликів повинна бути припинена.

Обробка виклику рекурсивної функції в принципі нічим не відрізняється від виклику функції звичайної: перед викликом функції в стек поміщаються її аргументи, потім адресу точки повернення, потім, уже при виконанні функції

- автоматичні змінні, локальні щодо цієї функції. Але якщо при виклику звичайних функцій число звернень до них невелика, то для рекурсивних функцій число викликів і, отже, кількість даних, що розміщуються в стеку, визначається глибиною рекурсії. Тому при рекурсії може виникнути ситуація переповнення стека.

Якщо спробувати відстежити по тексту програми процес виконання рекурсивної функції, то ми прийдемо до такої ситуації: увійшовши в рекурсивну функцію, ми "рухаємося" по її тексту до тих пір, поки не зустрінемо її виклику, після чого ми знову почнемо виконувати ту ж саму функцію спочатку. При цьому слід зазначити найважливіше властивість рекурсивної функції - її перший виклик ще не закінчився. Чисто зовні створюється враження, що текст функції відтворюється (копіюється) всякий раз, коли функція сама себе викликає. Насправді цей ефект відтворюється в комп'ютері. Однак копіюється при цьому не весь текст функції (не вся функція), а тільки її частини, пов'язані з даними (формальні, фактичні параметри, локальні змінні і точка повернення). Алгоритм (оператори, вирази) рекурсивної функції не змінюється, тому він присутній в пам'яті комп'ютера в єдиному екземплярі.

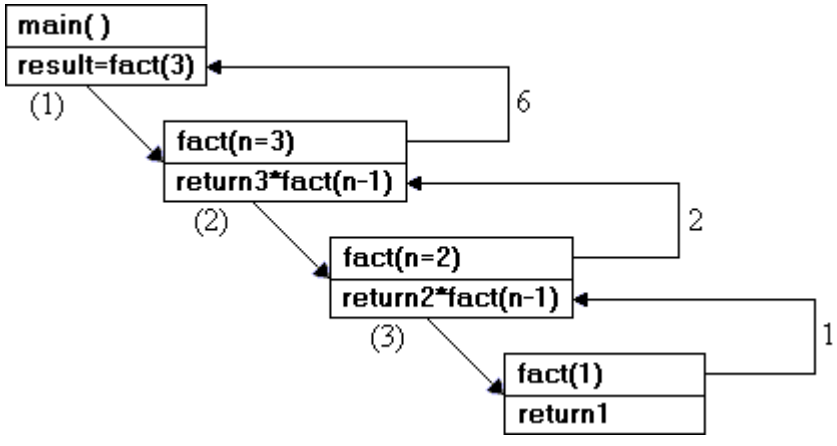
Приклад 1. Обчислити n!

Визначення факторіала рекурсивно:
0! = 1; n! = (N-1)! * N при n = 1,2,3, ...

Відповідно до цього визначення функції, що обчислює факторіал, можна записати наступним чином;

```
long fact (int n) {
    if (n <1) return 1;
    else return n * fact (n-1);
    }
```

Якщо, наприклад, в main написати long result = fact (3), то послідовність дзвінків можна показати так:



4.9 Практична робота № 6: "Рекурсія функції"

Рекурсією називається такий спосіб виклику, при якому функція звертається до самої себе.

Важливим моментом при складанні рекурсивної програми є організація виходу. Тут легко припуститися помилки, яка полягає в тому, що функція буде послідовно викликати саму себе нескінченно довго. Тому рекурсивний процес повинен крок за кроком так спрощувати завдання, щоб в кінці кінців для неї з'явилося не рекурсивне рішення. Використання рекурсії не завжди

бажано, так як це може привести до переповнення стека.

Приклад 1. Обчислити n!

Визначення факторіала рекурсивно:

0! = 1; n! = (N-1)! * N при n = 1,2,3, ...

Відповідно до цього визначення функції, що обчислює факторіал, можна записати наступним чином;

```
long fact (int n) {
    if (n <1) return 1;
    else return n * fact (n-1);
    }
```

Приклад 2. По заданому цілому числу роздрукувати символьну рядок цифр, що зображає це число:

```
void cnum (int n) {int a = 10;
    if (n == 0) return;
    else {cnum (n / a); cout << n% a;}
    }
```

При непрямої рекурсії здійснюється перехресний виклик функціями один одного. Хоча б в одній з них має бути умова, що викликає припинення рекурсії.

Непряма рекурсія є одним з тих випадків, коли не можна визначити функцію до використання її імені в програмі.

Нехай функція f1 () викликає f2 (), яка, в свою чергу, звертається до f1 (). Нехай перша з них визначена раніше другої. Для того щоб мати можливість звернутися до функції f2 () з f1 (), ми повинні помістити оголошення імені f2 раніше визначення обох цих функцій:

```
void f2 ();
    void f1 () {
```

```
...
if (...);
f2 ();
...}
void f2 () {
...
f1 ();
...}
```

5.5 Модуль 5: Препроцесор. Директиви препроцесору

5.1 Поняття препроцесору мови С

Препроцесор C / C + + - програмний інструмент, що змінює код програми для подальшої компіляції і збірки, використовуваний в мовах програмування Сі та його нащадка - C + +. Цей препроцесор забезпечує використання стандартного набору можливостей:

-Заміна тріграфов?? =,?? (,??) (Та інших) символами #, [,]

-Заміна коментарів порожніми рядками

-Включення файла - # include

-Макропідстановки - # define

-Умовна компіляція - # if, # ifdef, # elif, # else, # endif

Директиви препроцесора є інструкції, записані в тексті програми на СІ, і виконувані до трансляції програми. Директиви препроцесора дозволяють змінити текст програми, наприклад, замінити деякі лексеми в тексті, вставити текст з іншого файлу, заборонити трансляцію частини тексту і т.п. Всі директиви препроцесора починаються зі знака #. Після директив препроцесора крапка з комою не ставляться.

Важливою сферою застосування препроцесорів С є умовна компіляція. При підготовці програми до компіляції розробник може за допомогою декількох змін адаптувати програму до поточної ситуації (наприклад, до певної моделі процесора).

Препроцесор мови Сі - низькорівневий, лексичний препроцесор, тому що він вимагає тільки лексичного аналізу, тобто він обробляє тільки вихідний текст перед парсинг, виконуючи просту заміну лексем та спеціальних символів заданими послідовностями символів, відповідно до правил, встановлених користувачами.

5.2 Директива препроцесору # include

Директива # include включає в текст програми вміст зазначеного файлу. Ця директива має дві форми:

include "ім'я файлу"
include <назва файлу>

Назва файлу повинна відповідати угодами операційної системи і може складатися або тільки з імені файлу, або з імені файлу з попереднім йому маршрутом. Якщо ім'я файлу вказано в лапках, то пошук файлу здійснюється відповідно до заданого маршрутом, а при його відсутності в поточному

каталозі. Якщо ім'я файлу задано в кутових дужках, то пошук файлу проводиться в стандартних директоріях операційної системи, що задаються командою PATH.

Директива # include може бути вкладеною, тобто під включається файлі теж може міститися директива # include, яка заміщається після включення файлу, що містить цю директиву.

Директива # include широко використовується для включення в програму так званих заголовних файлів, що містять прототипи бібліотечних функцій, і тому більшість програм на CI починаються з цієї директиви.

5.3 Директива препроцесору # define, # undef

Директива # define

Директива # define служить для заміни часто використовуються констант, ключових слів, операторів або виразів деякими ідентифікаторами. Ідентифікатори, які замінять текстові або числові константи, називають іменованими константами. Ідентифікатори, які замінять фрагменти програм, називають макровизначення, причому макровизначення можуть мати аргументи.

Директива # define має дві синтаксичні форми:

Define ідентифікатор текст

Define ідентифікатор (список параметрів) текст

Ця директива замінює всі наступні входження ідентифікатора на текст. Такий процес називається макропідстановки. Текст може являти собою будь-який фрагмент програми на CI, а також може бути відсутня. В останньому випадку всі примірники ідентифікатора видаляються з програми.

Приклад:

Define WIDTH 80

Define LENGTH (WIDTH +10)

Ці директиви змінять в тексті програми кожне слово WIDTH на число 80, а кожне слово LENGTH на вираз (80 +10) разом з навколишніми його дужками.

Дужки, що містяться в макровизначеннями, дозволяють уникнути непорозумінь, пов'язаних з порядком обчислення операцій. Наприклад, при відсутності дужок вираз t = LENGTH * 7 буде перетворено у вираз t = 80 +10 * 7, а не в вираз t = (80 +10) * 7, як це виходить за наявності дужок, і в результаті вийде 780, а не 630.

У другій синтаксичної формі в директиві # define є список формальних параметрів, який може містити один або кілька ідентифікаторів, розділених комами. Формальні параметри в тексті макровизначення відзначають позиції

на які повинні бути підставлені фактичні аргументи макровиклику. Кожен формальний параметр може з'явитися в тексті макровизначення кілька разів.

При макровиклику слідом за ідентифікатором записується список фактичних аргументів, кількість яких має збігатися з кількістю формальних параметрів.

Приклад:

```
# Define MAX (x, y) ((x)> (y))? (X): (y)
```

Ця директива замінить фрагмент

```
t = MAX (i, s [i]);
```

на фрагмент

```
t = ((i)> (s [i])? (i): (s [i]);
```

Як і в попередньому прикладі, круглі дужки, в які укладені формальні параметри макровизначення, дозволяють уникнути помилок пов'язаних з неправильним порядком виконання операцій, якщо фактичні аргументи є виразами.

Наприклад, при наявності дужок фрагмент

```
t = MAX (i & j, s [i] | | j);
```

буде замінений на фрагмент

```
t = ((i & j)> (s [i] | | j)? (i & j): (s [i] | | j);
```

а при відсутності дужок - на фрагмент

```
t = (i & j> s [i] | | j)? i & j: s [i] | | j;
```

в якому умовний вираз обчислюється в зовсім іншому порядку.

Директива # undef

Директива # undef використовується для скасування дії директиви # define. Синтаксис цієї директиви наступний # undef ідентифікатор

Директива скасовує дію поточного визначення # define для зазначеного ідентифікатора. Не є помилкою використання директиви # undef для ідентифікатора, який не був визначений директивою # define.

приклад:

```
# undef WIDTH
# undef MAX
```

Ці директиви скасовують визначення іменованої константи WIDTH і макровизначення MAX.

5.4 Практична робота № 7: "Використання директив"

Майже всі програми на мові C + + використовують спеціальні команди для компілятора, які називаються директивами. У загальному випадку

директива - це вказівка компілятору мови C + + виконати ту чи іншу дію в момент компіляції програми. Існує суворо певний набір можливих директив, який включає в себе такі визначення:

Define, # elif, # else, # endif, # if, # ifdef, # ifndef, # include, # undef.

Директива # define використовується для завдання констант, ключових слів, операторів і виразів, що використовуються в програмі. Загальний синтаксис даної директиви має наступний вигляд:

Define <ідентифікатор> <текст>
або
Define <ідентифікатор> (<список параметрів>) <текст>

Слід зауважити, що символ ';' після директив не ставиться. Наведемо приклади варіантів використання директиви # define.

Приклади використання директиви # define.

```
# Include
# Define TWO 2
# Define FOUR TWO * TWO
# Define PX printf ("X одно% d. \ N", x)
# Define FMT «X одно% d. \ N»
# Define SQUARE (X) X * X
int main ()
{
 int x = TWO;
 PX;
 x = FOUR;
 printf (FMT, x);
 x = SQUARE (3);
 PX;

 return 0;
}
```

Після виконання цієї програми на екрані монітора з'явиться три рядки:

X дорівнює 2.
X дорівнює 4.
X дорівнює 9.

Директива # undef скасовує визначення, введене раніше директивою # define. Припустимо, що на якій-небудь ділянці програми потрібно скасувати ухвалу константи FOUR. Це досягається наступною командою:

```
# Undef FOUR
```

Цікавою особливістю даної директиви є можливість перевизначення значення раніше введеної константи. Дійсно, повторне використання директиви # define для раніше введеної константи FOUR неможливо, тому що це призведе до повідомлення про помилку в момент компіляції програми. Але якщо скасувати ухвалу константи FOUR за допомогою директиви # undef, то з'являється можливість повторного використання директиви # define для константи FOUR.

Для того щоб мати можливість виконувати умовну компіляцію, використовується група директив # if, # ifdef, # ifndef, # elif, # else і # endif. Наведена нижче програма виконує підключення бібліотек в залежності від встановлених констант.

```
# If defined (GRAPH)
# Include / / підключення графічної бібліотеки
# Elif defined (TEXT)
# Include / / підключення текстової бібліотеки
# Else
 # Include / / підключення бібліотеки введення-виведення
# Endif
```

Дана програма працює таким чином. Якщо раніше була задана константа з ім'ям GRAPH через директиву # define, то буде підключена графічна бібліотека з допомогою директиви # include. Якщо ідентифікатор GRAPH не визначений, але є ухвала TEXT, то буде використовуватися бібліотека текстового вводу / виводу. Інакше, при відсутності будь-яких визначень, підключається бібліотека введення / виводу. Замість словосполучення # if defined часто використовують скорочені позначення # ifdef і # ifndef і вище наведену програму можна переписати у вигляді:

```
# Ifdef GRAPH
 # Include / / підключення графічної бібліотеки
# Ifdef TEXT
 # Include / / підключення текстової бібліотеки
# Else
 # Include / / підключення бібліотеки введення-виведення
# Endif
```

Відмінність директиви # if від директив # ifdef і # ifndef полягає в можливості перевірки більш різноманітних умов, а не тільки існує чи ні будь-

які константи. Наприклад, за допомогою директиви # if можна проводити таку перевірку:

```
# If SIZE == 1
 # Include / / підключення математичної бібліотеки
# Elif SIZE> 1
 # Include / / підключення бібліотеки обробки масивів
# Endif
```

У наведеному прикладі підключається або математична бібліотека, або бібліотека обробки масивів, залежно від значення константи SIZE.

Ці вимоги іноді використовуються для виділення потрібних блоків програми, які потрібно використовувати в тій чи іншій програмної реалізації. Наступний приклад демонструє роботу такого програмного коду.

Приклад компіляції окремих блоків програми.

```
# Include
# Define SQUARE
int main ()
{
 int s = 0;
 int length = 10;
 int width = 5;

 # Ifdef SQUARE
 s = length * width;
 # Else
 s = 2 * (length + width);
# Endif

 return 0;
}
```

У даному прикладі відбувається обчислення або площі прямокутника, або його периметра, залежно від того визначено чи ні значення SQUARE. За замовчуванням програма обчислює площу прямокутника, але якщо прибрати рядок # define SQUARE, то програма стане обчислювати його периметр.

Використовувана в наведених прикладах директива # include дозволяє додавати в програму раніше написані програми та збережені у вигляді файлів. Наприклад, рядок

```
# Include <stdio.h>
```

вказує препроцесору додати вміст файлу stdio.h замість наведеної рядка. Це дає велику гнучкість, легкість програмування і наочність створюваного тексту програми. Є два різновиди директиви # include:

Include <stdio.h> - назва файлу в кутових дужках
і
Include «mylib.h» - ім'я файлу в лапках

Кутові дужки повідомляють препроцесору про те, що необхідно шукати файл (в даному випадку stdio.h) в одному або декількох стандартних системних каталогах. Лапки свідчать про те, що препроцесору необхідно спочатку виконати пошук файлу в поточному каталозі, тобто в тому, де знаходиться файл створюваної програми, а вже потім - шукати в стандартних каталогах.

6. Модуль 6: Масиви даних та індексація. Вказівники.

6.1 Визначення масиву даних. Формат обявлення

Масив даних-це упорядкований набір змінних одного типу, що містить фіксоване число компонент, яке задається при визначенні змінних цього типу. В цьому випадку в оперативній пам'яті комп'ютера резервується заданий при визначенні масиву кількість елементів пам'яті (байт, слово, подвійне слово і т.д.). Складові масив дані визначаються одним ім'ям і називаються елементами масиву.

Масив описується в блоці опису змінних Pascal-програми і має наступний формат:

<Ім'я масиву даних>: array [a .. b, c .. d, e .. f, ...] of <Тип даних масиву>;

де

<Ім'я масиву даних> - ідентифікатор, що визначає ім'я масиву. Через кому можуть бути перераховані і імена інших масивів, що мають однакову розмірність і тип;

<Тип даних масиву> - тип змінних масиву, який може бути будь-яким;

a .. b, c .. d, e .. f ... - розділені двома точками пари початкових і кінцевих значень індексів елементів масиву, які задаються явно в описі масиву даних.

Для вказівки місця розташування (адреси) елемента масиву служать індекси.

Індекси-цілі змінні типу byte, shortint, integer, word. Конкретне значення індексу (індексів) вказує місце розташування (адреса) елемента масиву. Кількість індексів або розмірність масиву в системі програмування TP 7.0 може бути будь-яким і обмежується розміром 65520 байт.

Фізичної моделлю масиву даних можуть служити поштові ящики, встановлювані в під'їздах наших багатоквартирних залізобетонних жител-монстрів. Окремий поштовий ящик, в даному випадку, є елементом масиву поштових скриньок і має свій номер-індекс.

Формат об'явлення

У кожній програмі, яка використовує масив, він обов'язково оголошується. Робиться це в такий спосіб. Нехай масив має розмірність N. Що це таке? N - це максимальна кількість елементів в масиві. Інакше - розмірність масиву. Отже, основна форма оголошення масиву наступна:

тип <ім'я масиву> [размер1] [размер2] ... [розмір N];

Найчастіше використовуються одновимірні масиви. Їх форма опису така:

тип <ім'я масиву> [размер1];

Тип - це базовий, тобто основний тип елементів масиву.

Розмір - це, як вище було сказано, кількість елементів одновимірного масиву. Вся справа в тому, що в двомірному масиві розмір визначається за допомогою множення. При описі двовимірного масиву його оголошення буде таке:

тип <ім'я масиву> [размер1] [размер2];

Це буде масив масиву. Тобто масив розміру [размер2], елементами якого є одновимірні масиви: <ім'я масиву> [размер1]. Розмір масиву в мові С задається константою або контантним виразом. Не можна задавати масив змінного розміру. Для цього існує окремий механізм, званий динамічним виділенням пам'яті. Питання про динамічний виділення пам'яті і про змінних масивах буде вивчатися нами пізніше. Але спочатку приділимо більше уваги одновимірним масивів.

6.2 Одномірні та двумірні масиви. Багатомірні масиви

Одномірні масиви

Масив даних називається одновимірним або вектором, якщо для визначення елемента масиву необхідно і достатньо однієї індексної змінної.

Для визначення розміру одновимірного масиву, в блоці опису змінних Pascal-програми потрібна одна пара початкових і кінцевих значень індексу.

Наприклад, певний одновимірний масив може бути визначений як:

```
Var
mas: array [0 .. 99] of real;
i: integer;
```

тобто масив речових даних mas розмірністю в 100 елементів з цілою, типу integer, індексного змінної i.

Доступ до елементів масиву здійснюється по конкретному індексу, який можна задати явно або обчислити в програмі, наприклад, фрагмент програми WriteLn (mas [10]); надрукує на екрані дисплея значення 10-го елемента масиву mas і перемістить курсор на наступний рядок.

Двумірні масиви,багатомірні масиви

Оголошення int A [n] створює в пам'яті одновимірний масив: набір пронумерованих елементів, що йдуть в пам'яті послідовно. Але можна створити і масив масивів наступним чином: int A [n] [m]. Дане оголошення створює масив з n об'єктів, кожен з яких в свою чергу є масивом типу int [m]. Тоді A [i], де i приймає значення від 0 до n-1 буде в свою чергу одним з n створених звичайних масивів, і звернутися до елементу з номером j в цьому масиві можна через A [i] [j].

Подібні об'єкти (масиви масивів) також називають двовимірними масивами. Двовимірні масиви можна представляти у вигляді квадратної таблиці, в якій перший індекс елемента означає номер рядка, а другий індекс - номер стовпця. Наприклад, масив A [3] [4] буде складатися з 12 елементів і його можна записати у вигляді

A [0] [0] A [0] [1] A [0] [2] A [0] [3]
A [1] [0] A [1] [1] A [1] [2] A [1] [3]
A [2] [0] A [2] [1] A [2] [2] A [2] [3]

Для зчитування, виведення на екран і обробки двовимірних масивів необхідно використовувати вкладені цикли. Перший цикл - по першому індексом (тобто за всіма рядками), другий цикл - по другому індексу, тобто по всіх елементів у рядках. Наприклад, вивести на екран двовимірний масив у вигляді таблиці, розділяючи елементи в рядку одним пропуском можна таким чином:

```
int A [n] [m];
for (int i = 0; i <n; + + i)
{/ / Виводимо на екран рядок i
  for (int j = 0; j <m; + + j)
    cout << A [i] [j] << "";
     cout << endl; / / Рядок завершується символом переходу на новий рядок
}
```

А вважати двовимірний масив з клавіатури можна за допомогою ще більш простого алгоритму (масив вводиться по рядках, тобто в порядку, відповідному першого прикладу):

```
for (i = 0; i <n; + + i)
  for (j = 0; j <m; + + j)
    cin >> A [i] [j];
```

6.3 Індексація масивів

Індексація - операція доступу до елементу масиву. Елементи нумеруються від 0.

Внутрішньо масиви реалізовані як покажчики на 0 елемент. Різниця між ними виявляється в операторі sizeof, який для масиву повертає розмір всього масиву. Розмір масиву може зазначатися тільки константним виразом.

```
# include <iostream>
using namespace std;

/ / Масиви по 10 елементів типу int
int array1 [10]; / / не ініціалізований масив

/ / Ініціалізований масив
/ / (задані значення кожного елемента)
int array2 [] =
    {1,2,3,4,5,6,7,8,9,0};

/ / Оголошення двомірних масивів
int array3 [5] [6];
int array4 [2] [3] = {{0,1,2}, {2,1,0}};

/ / Рядки в C завершуються 0, більш детально див рядки
char * str1 = "Hello, world"; / / покажчик на рядок "Hello, world"
char str2 [] = "Hello, world"; / / масив символів
char str3 [] = {'H', 'e', 'l', 'l', 'o', ',', '', 'w', 'o', 'r', 'l', 'd ', 0 "};

int * iptrarray [10]; / / масив покажчиків типу int
int (* iarrayptr) [10]; / / покажчик на масив з 10 елементів типу int

int main ()
{
int * iptr;
cout << "sizeof (iptr) =" << sizeof (iptr) << endl;
cout << "sizeof (array1) =" << sizeof (array1) << endl;
cout << "array2 [11] =" << array2 [11] << endl;
return 0;
}
```

6.4 Ініціалізація масивів

Ініціалізація масивів - це вміння привласнювати елементам масиву деякі початкові значення. У Сі для цих цілей передбачені деякі спеціальні

можливості. Найпростіший способо ініціалізації полягає в тому, що при об'вленіі масиву в фігурних дужках вказується список конкретних значень елементів масиву. Їх, тобто конкретні значення елементів масиву, називають ініціалізатор.

Наприклад:

Одновимірний масив семи речових чисел - ініціалізаторів: 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, позначений як float bonn [i] (де i = 0, ..., 6), можна ініціалізувати так: float bonn [7] = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7}; Це ж саме можна записати, використовуючи звичайні оператори присвоєння: bonn [0] = 1.1; bonn [1] = 2.2; bonn [2] = 3.3; bonn [3] = 4.4; bonn [4] = 5.5; bonn [5] = 6.6; bonn [7] = 7.7. Цими способ Ви можете задавати початкові значення елементів масиву, інакше кажучи, так можна вводити масив в пам'ять комп'ютера в Сі.

Двовимірний масив цілочисельних елементів - ініціалізаторів: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, позначений як int s [i] [j] (де i = 0, 2, ..., 3; j = 0, 2, ..., 4), можна ініціалізувати як: int s [4] [5] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}. За аналогією з першим випадком це відповідає набору наступних операторів присвоєння:

int s [0] [0] = 1; int s [0] [1] = 2; int s [0] [2] = 3; int s [0] [3] = 4; int s [0] [4] = 5;
int s [1] [0] = 6; int s [1] [1] = 7; int s [1] [2] = 8; int s [1] [3] = 9; int s [1] [4] = 10;
int s [2] [0] = 11; int s [2] [1] = 12; int s [2] [2] = 13; int s [2] [3] = 14; int s [2] [4] = 15;
int s [3] [0] = 16; int s [3] [1] = 17; int s [3] [2] = 18; int s [3] [3] = 19; int s [3] [4] = 20;

Взагалі багатовимірні масиви, в тому числі і двовимірні, можна ініціалізувати як масив масивів. Наприклад, дві наступні ініціалізації рівні між собою:

int s [3] [5] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}; i
int s [3] [5] = {{1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}};

У другому випадку автори ввели додаткові фігурні дужки для запису елементів кожного рядка двовимірного масиву. Якщо при цьому кількість елементів у рядку не збігається з числом стовпців у масиві, то відповідні пропущені елементи рядка вважаються невизначеними, оскільки вони не визначені.

6.5 Практична робота № 8: "Використання масивів та строк"

Прийоми використання масивів і рядків на прикладі гри в хрестики-нулики:

Представлений довгий приклад ілюструє велику кількість прийомів використання рядків. Розглядається проста програма гри в хрестики-нулики. Двомірний масив використовується в якості матриці, що зображає гральну дошку.

Комп'ютер грає в дуже просту гру. Коли настає черга ходу комп'ютера, функція get_computer_move () переглядає матрицю в пошуку незайнятих осередків. Якщо функція знаходить незайняту комірку, вона поміщає туди символ O. Якщо незайнятої осередку немає, то функція виводить повідомлення про закінчення гри і припиняє роботу програми. Функція get_player_move () запитує грає, де він хоче помістити символ X. Верхній лівий кут має координати (1, 1), а нижній правий - (3, 3).

Масив matrix, що містить матрицю гри, инициализирован символами пробіл. Кожен хід, зроблений гравцем або комп'ютером, замінює символ пробілу символом X або O. Це дозволяє легко відобразити матрицю на екрані.

Після кожного ходу викликається функція check (), яка повертає пробіл, якщо переможця ще немає, або X, якщо переміг гравець, або O, коли переміг комп'ютер. Ця функція переглядає рядки, стовпці і діагоналі в пошуку трьох однакових символів (X або O) поспіль.

Функція disp_matrix () відображає на екрані поточний стан гри. Зверніть увагу на те, як істотно спрощує цю функцію ініціалізація матриці пробілами.

Функції отримують доступ до масиву matrix різними способами. Їх варто уважно вивчити для кращого розуміння прийомів роботи з масивами.

```
/ * Проста гра в хрестики-нулики. * /
# Include <stdio.h>
# Include <stdlib.h>

char matrix [3] [3]; / * матриця гри * /

char check (void);
void init_matrix (void);
void get_player_move (void);
void get_computer_move (void);
void disp_matrix (void);

int main (void)
{
```

```
  char done;

  printf ("Це гра в хрестики-нулики. \ n");
  printf ("Ви будете грати проти комп'ютера. \ n");

  done = ';
  init_matrix ();

  do {
    disp_matrix ();
    get_player_move ();
    done = check (); / * перевірка, чи є переможець * /
    if (done! = ') break; / * є переможець * /
    get_computer_move ();
    done = check (); / * перевірка, чи є переможець * /
  } While (done == ');

  if (done == 'X') printf ("Ви перемогли! \ n");
  else printf ("Переміг комп'ютер!! \ n");
  disp_matrix (); / * показ фінальної позиції * /

  return 0;
}

/ * Ініціалізація матриці гри. * /
void init_matrix (void)
{
  int i, j;

  for (i = 0; i <3; i + +)
    for (j = 0; j <3; j + +) matrix [i] [j] = ';
}

/ * Хід гравця. * /
void get_player_move (void)
{
  int x, y;

  printf ("Введіть координати X, Y Вашого ходу:");
  scanf ("% d% * c% d", & x, & y);

  x -; y -;

  if (matrix [x] [y]! = ') {
    printf ("Невірний хід, спробуйте ще. \ n");
```

```
    get_player_move ();
  }
  else matrix [x] [y] = 'X';
}

/ * Хід комп'ютера. * /
void get_computer_move (void)
{
  int i, j;
  for (i = 0; i <3; i + +) {
    for (j = 0; j <3; j + +)
      if (matrix [i] [j] == ") break;
    if (matrix [i] [j] == ") break;
  / * Другий break потрібен для виходу з циклу по i * /
  }

  if (i * j == 9) {
    printf ("Кінець гри \ n");
    exit (0);
  }
  else
    matrix [i] [j] = 'O';
}

/ * Вивід матриці на екран. * /
void disp_matrix (void)
{
  int t;

  for (t = 0; t <3; t + +) {
    printf ("% c |% c |% c", matrix [t] [0],
          matrix [t] [1], matrix [t] [2]);
    if (t! = 2) printf ("\ n --- | --- | --- \ n");
  }
  printf ("\ n");
}

/ * Визначення переможця. * /
char check (void)
{
  int i;

  for (i = 0; i <3; i + +) / * перевірка рядків * /
    if (matrix [i] [0] == matrix [i] [1] &&
       matrix [i] [0] == matrix [i] [2]) return matrix [i] [0];
```

```
  for (i = 0; i <3; i + +) / * перевірка стовпців * /
    if (matrix [0] [i] == matrix [1] [i] &&
       matrix [0] [i] == matrix [2] [i]) return matrix [0] [i];

  / * Перевірка діагоналей * /
  if (matrix [0] [0] == matrix [1] [1] &&
     matrix [1] [1] == matrix [2] [2])
       return matrix [0] [0];

  if (matrix [0] [2] == matrix [1] [1] &&
     matrix [1] [1] == matrix [2] [0])
       return matrix [0] [2];

  return '';
}
```

Пояснення до програми. У функції get_player_move () за допомогою бібліотечної функції scanf () зчитуються з клавіатури два цілих числа x і y. Функція scanf () при зчитуванні чисел припускає, що у вхідному потоці вони розділені пробілами (або пробільними символами), інші розділові символи не допускаються. Проте багато користувачів звикли до того, що числа можна розділяти, наприклад, комами. (Власне кажучи, саме так і пропонується в підказці, що видається програмою.) У наведеному прикладі символ, наступний безпосередньо після першого числа, просто ігнорується, саме для цього у функції scanf () використовується специфікатор формату% * c. Зірочка означає, що символ зчитується з потоку, але в пам'ять не записується.

6.6 Визначення вказівника

Вказівник - це змінна, яка містить адресу деякого об'єкта, наприклад, інший змінної. Точніше - адреса першого байта цього об'єкта. Це дає можливість непрямого доступу до цього об'єкта через вказівник Нехай x - змінна типу int. Позначимо через px вказівник. Унарний операція & видає адресу об'єкта, так що оператор

```
px = &x;
```

присвоює змінної px адресу змінної x. Кажуть, що px "вказує" на x. Операція & застосовна тільки до адресних виразів, так що конструкції виду & (x-1) і & 3 незаконні.

Унарний операція * називається операцією разадресаціі або операцією дозволу адреси. Ця операція розглядає свій операнд як адресу і звертається за цією адресою, щоб отримати об'єкт, що міститься за цією адресою.

Отже, якщо y теж має тип int, то

```
y = * px;
```

присвоює y вміст того, на що вказує px. Так, послідовність

```
px = &x;
     y = * px;
```

присвоює y те ж саме значення, що і оператор

```
y = x;
```

Всі ці змінні повинні бути описані:

```
int x, y;
     int * px;
```

Останнє - опис вказівника. Його можна розглядати як мнемонічне. Воно каже, що комбінація * px має тип int або, інакше, px є вказівник на int. Це означає, що якщо px з'являється у вигляді * px, то це еквівалентно змінної типу int.

З опису покажчика випливає, що він може вказувати тільки на певний вид об'єкта (у даному випадку int). Разадресованний покажчик може входити в будь вираження там, де може з'явитися об'єкт того типу, на який цей вказівник вказує. Так, оператор

```
y = * px + 2;
```

присвоює y значення, на 2 більше, ніж x

Зауважимо, що пріоритет унарних операцій * та & такий, що ці операції пов'язані зі своїми операндами більш міцно, ніж арифметичні операції, так що вираз

```
y = * px + 2
```

бере те значення, на яке вказує px, додає 2 і привласнює результат змінної y.

Якщо px вказує на x, то

```
* Px = 3;
```

вважає x рівним 3, а

* Px + = 1;

збільшує x на 1, також як і вираз

(* Px) + +

Круглі дужки тут необхідні. Якщо їх опустити, тобто написати * px + +, то, оскільки унарні операції, подібні * i + +, виконуються справа - наліво, цей вислів збільшить px, а не ту змінну, на яку він вказує.

Якщо py - інший вказівник на int, то можна виконати привласнення

py = px;

Тут адресу з px копіюється в py. В результаті py вказує на те ж, що й px.

6.7 Використання вказівників з іншими масивами

Незважаючи на те що покажчики широко використовуються з символьними рядками, ви можете використовувати покажчики з масивами інших типів. Наприклад, наступна програма PTRFLOAT.CPP використовує покажчик на масив типу float для виведення значень з плаваючою крапкою:

```
# include <iostream.h>

void show_float (float * array, int number_of_elements)

{
  int i;
  for (i = 0; i <number_of_elements; i + +) cout << * array + + << endl;
}

void main (void)

{
  float values [5] = {1.1, 2.2, 3.3, 4.4, 5.5);
  show_float (values, 5);
}
```

Як бачите, усередині функції show_float цикл for використовує значення, що визначене за допомогою покажчика array, а потім збільшує цей покажчик до наступного значення. В даному випадку програма повинна передати параметр, який задає кількість елементів масиву, оскільки на відміну від символьних рядків масиви типу float (або int, long і т. д.) не використовують символ NULL для визначення останнього елемента.

6.8 Адресна арифметика

Якщо p є вказівник на певний елемент масиву, то p + + збільшує p так, щоб він вказував на наступний елемент, а p + = і збільшує його, щоб він вказував на і-й елемент після того, на який вказував раніше. Ці та подібні конструкції - найпростіші приклади арифметики над покажчиками, званої також адресної арифметикою.

Сі послідовний і однаковості у своєму підході до адресної арифметики. Це з'єднання в одній мові покажчиків, масивів і адресної арифметики - одна з сильних його сторін. Проілюструємо сказане побудовою простого розподільника пам'яті, що складається з двох програм. Перша, alloc (n), повертає покажчик p на n послідовно розташованих осередків типу char. Програма, яка звертається до alloc, може використовувати ці осередки для запам'ятовування символів. Друга, afree (p), звільняє пам'ять для того, щоб її можна було знову використовувати. Простота алгоритму обумовлена припущенням, що звернення до afree робляться в зворотному порядку по відношенню до відповідних зверненнями до alloc. Таким чином, пам'ять, з якою працюють alloc і afree, є стеком (списком, в основі якого лежить принцип «останнім ввійшов, першим пішов»). У стандартній бібліотеці є функції malloc і free, які роблять те ж саме, тільки без згаданих обмежень.

Функцію alloc найлегше реалізувати, якщо домовитися, що вона буде розподіляти шматки деякого великого масиву типу char, який ми назвемо allocbuf. Цей масив віддамо в особисте користування функціям alloc і afree. Так як вони мають справу з покажчиками, а не з індексами масиву, то іншим програмам знати його ім'я не потрібно. Крім того, цей масив можна визначити в тому ж вихідному файлі, що і alloc і afree, оголосивши його як static, завдяки чому він стане невидимим поза цього файлу. На практиці такий масив може і зовсім не мати імені, оскільки його можна запросто за допомогою malloc у операційної системи і отримати покажчик на деякий безіменний блок пам'яті.

Природно, нам потрібно знати, скільки елементів масиву allocbuf вже зайнято. Ми введемо покажчик allocp, який буде вказувати на перший вільний елемент. Якщо запитується пам'ять для n символів, то alloc повертає поточне значення allocp (тобто адресу початку вільного блоку) і потім збільшує його на n, щоб покажчик allocp вказував на наступну вільну область. Якщо ж простору немає, то alloc повертає нуль. Функція afree (p) просто привласнює allocp переданий параметр p, якщо він не виходить за межі масиву allocbuf.

```
# Define ALLOCSIZE 10000 / * розмір доступного простору * /
```

```
static char allocbuf [ALLOCSIZE]; / * пам'ять для alloc * /
static char * allocp = allocbuf; / * покажчик на вільне місце * /

char * alloc (int n) / * повертає покажчик на пам'ять для n символів * /
{
   if (allocbuf + ALLOCSIZE - allocp> = n) {
      allocp + = n; / * простір є * /
      return allocp - n; / * старий покажчик * /
   }
   else / * простору немає * /
      return 0;
}
void afree (char * p) / * звільняє пам'ять, на яку вказує p * /
{
   if (p> = allocbuf && p <allocbuf + ALLOCSIZE)
      allocp = p;
}
```

У загальному випадку покажчик, як і будь-яку іншу змінну, можна ініціалізувати, але тільки такими осмисленими для нього значеннями, як нуль або вираз, результатом якого є адреса раніше визначених даних відповідного типу. Оголошення

```
static char * allocp = allocbuf;
```

визначає allocp як покажчик на char і ініціалізує його адресою першого елемента масиву allocbuf, оскільки перед початком роботи програми масив allocbuf порожній. Зазначене оголошення могло б мати і такий вигляд:

```
static char * allocp = & allocbuf [0];
```

оскільки ім'я масиву і є адреса його нульового елемента.

Перевірка

```
if (allocbuf + ALLOCSIZE - allocp> = n) {/ * годиться * /
```

контролюється, чи достатньо простору, щоб задовольнити запит на n символів. Якщо пам'яті достатньо, то нове значення для allocp повинно вказувати не далі, ніж на наступну позицію за останнім елементом allocbuf. При виконанні цієї вимоги alloc видає покажчик на початок виділеного блоку символів (зверніть увагу на оголошення типу самої функції). Якщо вимога не виконується, функція alloc повинна видати якийсь сигнал про те, що пам'яті не вистачає. Сі гарантує, що нуль ніколи не буде правильною адресою для даних, тому ми будемо використовувати його в якості ознаки аварійної події, в нашому випадку недостачі пам'яті.

Покажчики і цілі не є взаємозамінними об'єктами. Константа нуль - єдине виключення з цього правила: її можна привласнити покажчику, і покажчик

можна порівняти з нульовою константою. Щоб показати, що нуль - це спеціальне значення для покажчика, замість цифри нуль, як правило, записують NULL - константу, визначену у файлі <stdio.h>. З цього моменту і ми будемо нею користуватися.

Перевірки

```
if (allocbuf + ALLOCSIZE - allocp> = n) {/ * годиться * /
```

і

```
if (p> = allocbuf && p <allocbuf + ALLOCSIZE)
```

демонструють кілька важливих властивостей арифметики з покажчиками. По-перше, при дотриманні деяких правил покажчики можна порівнювати.

Якщо p і q вказують на елементи одного масиву, то до них можна застосовувати операції відношення ==,! =, <,> = І т.д. Наприклад, ставлення виду

```
p <q
<
```

Поправді, есліелемент масиву, на який вказує p розташований раніше, ніж елемент на який вказує q. Будь покажчик завжди можна порівнювати на рівність і нерівність з нулем. А ось для покажчиків, не вказують на елементи одного масиву, результат арифметичних операцій або порівнянь не визначений. (Існує один виняток: в арифметиці з покажчиками можна використовувати адресу неіснуючого «наступного за масивом» елемента, тобто адресу того «елементу», який стане останнім, якщо в масив додати ще один елемент.)

По-друге, як ви вже, напевно, помітили, уазателі й цілі можна складати і віднімати. Конструкція

```
p + n
```

означає адреса об'єкту, що займає n-е місце після об'єкта, на який вказує p. Це справедливо безвідносно до типу об'єкта, на який вказує p; n автоматично збільшується на коефіцієнт, що відповідає розміру об'єкта. Інформація про розмір неявно присутня в оголошенні p. Якщо, наприклад, int займає чотири байти, то коефіцієнт множення буде дорівнює чотирьом.

Допускається також віднімання покажчиків. Наприклад, якщо p і q вказують на елементи одного масиву і p <q, то q - p + 1 є число елементів, розташованих від p до q включно. Цим фактом можна скористатися при написанні ще однією версією strlen:

```
/ * Strlen: повертає довжину рядка s * /
int strlen (char * s)
{
   char * p = s;
```

```
    while (* p! = '\ 0')
        p + +;
    return p - s;
}
```

У своєму оголошенні p ініціалізується значенням s, тобто спочатку p вказує на перший символ рядка. На кожному кроці циклу while перевіряється черговий символ; цикл триває до тих пір, поки не зустрінеться символ '\ 0'. Кожне просування покажчика p на наступний символ виконується інструкцією p + +, і різниця p - s дає число пройдених символів, тобто довжину рядка. (Число символів у рядку може бути занадто великим, щоб зберігати його у змінній типу int. Тип ptrdiff_t, достатній для зберігання різниці (зі знаком) двох покажчиків, визначено в заголовному файлі <stddef.h>. Проте, якщо бути дуже обережними, нам слід було б для повертається результату використовувати тип size_t, в цьому випадку наша програма відповідала б стандартної бібліотечної версії. Тип size_t є тип беззнакового цілого, що повертається оператором sizeof.)

Арифметика з покажчиками враховує тип: якщо вона має справу зі значеннями float, які займають більше пам'яті, ніж char, і p - покажчик на float, то p + + просуне p на таке значення типу float. Це означає, що іншу версію alloc, яка має справу з елементами типу float, а не char, можна отримати простою заміною в alloc і afree всіх char на float. Всі операції з покажчиками будуть автоматично відкориговані відповідно до розміру об'єктів, на які вказують покажчики.

Можна проводити такі операції з покажчиками: присвоювання значення покажчика іншому вказівником того ж типу, додавання і віднімання покажчика та цілого, віднімання і порівняння двох покажчиків, які вказують на елементи одного і того ж масиву, а також привласнення вказівником нуля і порівняння покажчика з нулем. Виконання інших операцій з покажчиками не допускається. Не можна складати два покажчика, перемножувати їх, ділити, зрушувати, виділяти розряди; покажчик не можна складати із значенням типу float або double; вказівником одного типу не можна навіть привласнити покажчик іншого типу, не виконавши попередньо операції приведення (виняток становлять лише покажчики типу void *).

6.9 Методи сортировки масивів

Методи сортування

Сортування масиву методом бульбашки - повільна, але якщо швидкість не головне, можна застосувати і його.

Алгоритм дуже простий - якщо два сусідніх елементи розташовані не по порядку,

то міняємо їх місцями. Так повторюємо до тих пір, поки в черговому проході не зробимо жодного обміну,

тобто масив буде впорядкованим. Нижче текст процедури, що реалізує алгоритм сортування методом бульбашки

(Arr - масив для сортування з початковим індексом 0, n - розмірність масиву)

```
procedure SortPuz (var Arr: array of Integer; n: Integer);
var
  i: Integer;
  Temp: Integer;
  Flag: Boolean;
begin
  repeat
    Flag: = False;
    for i: = 0 to n - 1 do
      if Arr [i]> Arr [i + 1] then begin
        Temp: = Arr [i];
        Arr [i]: = Arr [i + 1];
        Arr [i + 1]: = Temp;
        Flag: = True;
      end;
  until Flag = False;
end;
```

Сортування методом знаходження мінімального елемента

Ще один варіант сортування, більш швидкий, ніж метод бульбашки.

Полягає він в наступному: при кожному перегляді масиву знаходимо мінімальний елемент і міняємо місцями його з першим на першому проході,

з другим - на другому і т.д. Не забудьте тільки, що перший елемент масиву повинен мати індекс 0.

```
procedure SortMin (var Arr: array of Integer; n: Integer);
var
  i, j: Integer;
  Min, Pos, Temp: Integer;
begin
  for i: = 0 to n - 1 do begin
    Min: = Arr [i];
    Pos: = i;
    for j: = i + 1 to n do
      if Arr [j] <Min then begin
        Min: = Arr [j];
        Pos: = j;
      end;
```

```
      Temp: = Arr [i];
      Arr [i]: = Arr [Pos];
      Arr [Pos]: = Temp;
    end;
  end;
```

Сортування масиву вставками

Більш швидкий і оптимальний метод сортування - сортування вставками.

Суть її в тому, що на n-ном кроці ми маємо впорядковану частина масиву з n елементів,
і наступний елемент встає на відповідне йому місце.

Майте на увазі - перший індекс масиву - 0.

```
procedure SortInsert (var Arr: array of Integer; n: Integer);
var
  i, j, Temp: Integer;
begin
  for i: = 1 to n do begin
    Temp: = Arr [i];
    j: = i - 1;
    while Temp <Arr [j] do begin
      Arr [j + 1]: = Arr [j];
      Dec (j);
      if j <0 then
        Break;
    end;
    Arr [j + 1]: = Temp;
  end;
end;
```

Пошук перебором

Щоб знайти якісь дані в неврегульованих масиві,
застосовується алгоритм простого перебору елементів.

Наступна функція повертає індекс заданого елемента масиву.

Її аргументи: масив з першим індексом 0, кількість елементів
в масиві і шукане число. Якщо число не знайдено, повертається -1.

```
function SearchPer (Arr: array of Integer; n, v: Integer): Integer;
var
  i: Integer;
begin
  Result: = -1;
  for i: = 1 to n do
    if Arr [i] = v then begin
      Result: = i;
      Exit;
    end;
```

```
end;
```

Бінарний пошук

При пошуку в упорядкованому масиві можна застосувати набагато більш швидкий метод пошуку - бінарний.

Суть його в наступному: На початку мінлива Up вказує на самий маленький елемент масиву (Up: = 0), Down - на найбільший

(Down: = n, де n - верхній індекс масиву), а Mid - на середній.

Далі, якщо шукане число дорівнює Mid, то задача вирішена, якщо число менше Mid,

то потрібний нам елемент лежить нижче середнього, і за нове значення Up приймається Mid + 1;

і якщо потрібне нам число менше середнього елемента, значить, воно розташоване

вище середнього елемента, і Down: = Mid - 1. Потім слід нова ітерація циклу,

і так повторюється до тих пір, поки не знайдеться потрібне число, або Up не стане більше Doun.

```
function SearchBin (Arr: array of Integer; v, n: Integer): Integer;
var
  Up, Down, Mid: Integer;
  Found: Boolean;
begin
  Up: = 0; Down: = n;
  Found: = False; Result: = -1;
  repeat
    Mid: = Trunc ((Down - Up) / 2) + Up;
    if Arr [Mid] = v then
      Found: = True
    else
      if v <Arr [Mid] then
        Down: = Mid - 1
      else
        Up: = Mid + 1;
  until (Up> Down) or Found;
  if Found then
    Result: = Mid;
end;
```

6.10 Масиви вказівників

У мові CI елементи масивів можуть мати будь-який тип, і, зокрема, можуть бути покажчиками на будь-який тип. Розглянемо кілька прикладів з використанням покажчиків.

Наступні оголошення змінних

```
int a [] = {10,11,12,13,14,};
    int * p [] = {a, a +1, a +2, a +2, a +3, a +4};
    int ** pp = p;
```

породжують програмні об'єкти.

При виконанні операції pp-p отримаємо нульове значення, так як посилання pp і p рівні і вказують на початковий елемент масиву покажчиків, пов'язаного з покажчиком p (на елемент p [0

Результатом виконання віднімання pp-p буде 2, так як значення pp є адреса третього елемента масиву p. Посилання * pp-a теж дає значення 2, так як звернення * pp є адреса третього елемента масиву a, а звернення a є адреса початкового елемента масиву a. При зверненні за допомогою посилання ** pp отримаємо 12 - це значення третього елементу масиву a. Посилання * pp + + дасть значення четвертого елемента масиву p тобто адреса четвертого елемента масиву a.

Якщо вважати, що pp = p, то звернення * + + pp це значення першого елемента масиву a (тобто значення 11), операція + + * pp змінить вміст покажчика p [0], таким чином, що він стане рівним значенню адреси елемента a [1].

Складні звернення розкриваються зсередини. Наприклад звернення * (+ + (* pp)) можна розбити на наступні дії: * pp дає значення початкового елемента масиву p [0], далі це значення інкременірується + + (* p) в результаті чого покажчик p [0] стане дорівнює значенню адреси елемента a [1], і останнє дію це вибірка значення за отриманим адресою, тобто значення 11.

У попередніх прикладах був використаний одномірний масив, розглянемо тепер приклад з багатовимірним масивом і покажчиками. Наступні оголошення змінних

```
int a [3] [3] = {{11,12,13},
                 {21,22,23},
                 {31,32,33}};
int * pa [3] = {a, a [1], a [2]};
int * p = a [0];
```

породжують у програмі об'єкти.

7. МОДУЛЬ 7: Робота з файлами мови С

7.1 Поняття та робота з файом

Файлом називають спосіб зберігання інформації на фізичному пристрої. Файл - це поняття, яке застосовується до всього - від файлу на диску до терміналу.

В С + + відсутні оператори для роботи з файлами. Всі необхідні дії виконуються за допомогою функцій, включених в стандартну бібліотеку. Вони дозволяють працювати з різними пристроями, такими, як диски, принтер, комунікаційні канали і т.д. Ці пристрої сильно відрізняються один від одного. Однак файлова система перетворює їх в єдине абстрактне логічне пристрій, який називається потоком.

Текстовий потік - це послідовність символів. При передачі символів з потоку на екран, частина з них не виводиться (наприклад, символ повернення каретки, переведення рядка).

Двійковий потік - це послідовність байтів, які однозначно відповідають тому, що знаходиться на зовнішньому пристрої.

Організація роботи з файлами засобами С

Оголошення файлу
FILE * ідентифікатор;
Приклад
FILE * f;
Відкриття файлу:
fopen (ім'я фізичного файлу, режим доступу)

Режим доступу - рядок, що вказує режим відкриття файлу файлу і тип файлу

Типи файлу: бінарний (b); текстовий (t)

Значення	Опис
r	Файл відкривається тільки для читання
w	Файл відкривається тільки для запису. Якщо відповідний фізичний файл існує, він буде перезаписаний
a	Файл відкривається для запису в кінець (для дозапису) або створюється, якщо не існує
r +	Файл відкривається для читання і запису.
w+	Файл відкривається для запису і читання. Якщо відповідний фізичний файл існує, він буде перезаписаний
a+	Файл відкривається для запису в кінець (для дозапису) або створюється, якщо не існує

Наприклад

```
f = fopen (s, "wb");
k = fopen ("h: \ ex.dat", "rb");
```

7.2 Структура file

Введення і виведення (I / O): stdio.h

Розглядаються різноманітні види I / O.

Ваша програма повинна включати стандартний I / O заголовний файл, а имено рядок:

```
# Include <stdio.h>
```

Повідомлення про помилки

Часто буває зручно і корисно повідомляти про помилки при виконанні С-програми. Стандартна бібліотечна функція perror () дозволяє це робити. Вона використовуєте спільно з errno. Часто при виникненні помилки необхідно припинити роботу програми. Для цього використовують функцію exit ().

perror ()

Функція perror () має прототип:

```
void perror (const char * message);
```

perror () виводить повідомлення (в стандартний потік помилок), потім опис останньої трапилася помилки (на підставі errno) (див. нижче), що сталася при зверненні до системної або бібліотечної функції. Аргумент рядок message друкується спочатку, потім двокрапка і пробіл, потім повідомлення та перекладів рядка. Якщо message is покажчик NULL або вказує на рядок нульової довжини, то двокрапка не друкується.

errno

errno - це спеціальна системна змінна, значення якої встановлюється, якщо системний виклик не може виконати поставлену задачу. Мінлива задається в # include <errno.h>.

Для використання errno в С-програмі необхідно оголосити:

```
extern int errno;
```

Значення змінної можна вручну задати всередині С-програми (хоча зазвичай так не роблять), в іншому випадку значення змінної залишається встановленим при останньому системному виклику або виклик бібліотечної функції.

exit ()

Прототип exit () міститься в # include <stdlib> і має вигляд:

void exit (int status)

exit завершує виконання програми і повертає код повернення в операційну систему. Значення коду повернення використовується для визначення, чи успішно завершилася програма, Всі відкриті потоки закриваються і буфери очищуються.

Програма завершує роботу з кодом повернення EXIT_SUCCESS при успішному завершенні.

При виникненні помилки ви можете викликати exit (EXIT_FAILURE) для припинення роботи програми.

Потоки

Потоки представляють собою зручний переносимий спосіб читанні і запису даних. Вони забезпечують гнучкі і ефективні засоби для I / O.

Потік - це файл або фізична пристрій (тобто принтер або монітор, в UNIX-системах фізичні пристрої представляються файлами), які управляються через покажчик на потік.

Існує внутрішня С-структура даних FILE, яка використовується для роботи з потоками і визначена в stdio.h. При введенні / виведенні за допомогою потоків необхідно використовувати цю структуру FILE.

Необхідно обявіть змінну або покажчик цього типу в нашій програмі.

Ми повинні спочатку відкрити потік перед виконання будь-яких операцій I / O,

потім виконати ці операції доступу (читання / запису)

і потім закрити.

Потоковий I / O - буферізірованний: Це означає, що блок даних фіксіровнного розміру чітаентся / пишеться в файл не безпосередньо, а через тимчасову область зберігання (буфер).

Це підвищує ефективність I / O, але будьте обережні: дані, записані в буфер, не з'являться у файлі або на пристрої, поки буфер не буде очищений. Будь-яке неуспішне завершення програми може викликати проблеми. Потоковий I / O ефективний для символів і рядків.

Зумовлені потоки

В UNIX задані 3 предопределеннич потоку (в stdio.h):

stdin (стандартного входу), stdout (стандартний потік виводу), stderr (стандартний потік помилок)

Висновок даних через stdout і stderr за замовчуванням здійснюється на консоль. Введення даних через stdin по умолчнію здійснюється через клавіатуру.

Зумовлені потоки завжди відкриті.

Перенаправлення

Зумовлені потоки можна перенаправляти (міняти умовчання).

Таке перевизначення не є частиною мови С, а залежить від операційної системи. Перенаправлення задається за допомогою командного рядка.

> - Перенаправити stdout у файл.

Якщо у нас є програма, out, яка друкує на екран, то

out> file1

буде друкувати у файл, file1.

<- Перенаправлення stdin з файлу в програму.

Якщо ми очікуємо введення з клавіатури в програму, in, то миможем читати з файлу

in <file2.

| - Конвеєр: передає stdout однієї програми в stdin іншої програми

prog1 | prog2

Основи I / O

int getchar (void) - читає char з stdin
int putchar (char ch) - пише char в stdout, повертає код записаного символу.

int ch;

ch = getchar ();

(Void) putchar ((char) ch);

Споріднені функції:

int getc (FILE * stream),
int putc (char ch, FILE * stream)

Форматований I / O

printf

Прототип:

int printf (char * format, arg list ...) -
виводить у стандартний потік виводу stdout список аргументів згідно спеціальної форматної рядку. Повертає число надрукованих символів.

scanf

Ця функція визначена таким чином:

int scanf (char * format, args ....) - читає з stdin і кладе прочитані дані за адресами, вказаними в списку args. Повертає число прочитаних даних.
Форматна рядок подібна рядку для printf
Note: В scanf необхідно передавати адресу змінної або покажчик на неї.

scanf ("% d", & i);

Ми можемо передати також ім'я масиву або рядка, т.к, це і буде адресою початку масиву або рядка.

char string [80];
scanf ("% s", string);

7.3 Файли проізвольного та послідовного доступу

Для зовнішніх файлів визначено два сорти доступа: послідовний доступ і прямий доступ. У настроюваних пакетах SEQUENTIAL_IO і DIRECT_IO описані відповідні файлові типи та пов'язані з ними операції. Об'єкт файлового типу, який використовується для послідовного доступу, називається послідовним файлом, а використовуваний для прямого доступу - прямим файлом.

При послідовному доступі файл розглядається як послідовність значень, які передаються в порядку їх надходження (від програми або з оточення). Якщо файл відкритий, то передача починається з початку файлу.

При прямому доступі файл розглядається як набір елементів, що займають послідовні позиції в лінійному порядку; значення може бути передано в елемент файла (або з нього), що знаходиться в будь-якій обраній позиції. Позиція елемента задається його індексом, який є позитивним числом визначається реалізацією цілого типу COUNT. Індекс першого елемента в файлі (якщо він є) дорівнює одиниці; індекс останнього елемента (якщо він є) називається поточним розміром; поточний розмір файлу, що не містить жодного елемента, дорівнює нулю. Поточний розмір - це характеристика зовнішнього файлу.

Відкритий прямий файл має поточний індекс, який буде використаний наступної операцією для читання або запису. За відкритті прямого файлу значення поточного індексу встановлюється рівним одиниці. Поточний індекс прямого файлу - це характеристика не зовнішнього файлу, а пов'язаного з ним об'єкта файлового типу.

Для прямих файлів допустимі всі три види файлу. Для послідовних файлів допустимі тільки види IN_FILE і OUT_FILE.

Для работы с таким файлом необходимо сначала создать пользовательский тип (UDT), который представляет собой шаблон структуры вашего файла.

Пример:

```
Private Type MyRecord

        FIO As String

        Birth As Date

        Salary As Integer

End Type
```

Потом необходимо открыть файл с помощью оператора Open. Отличие от открытия файлов последовательного доступа заключается в использовании специального режима открытия - Random и задании в конце конструкции длины структурной единицы файла прямого доступа с помощью Len = <длина_в_байтах>.

Пример:

```
Dim tmprec As MyRecord
```

Open "c:\tmp.mmm" For Random As #1 Len = Len (tmprec)

Чтение записей осуществляется с помощью оператора

Get #номер_файла, [<номер_записи>], <переменная>

Если параметр <номер_записи> опускается, то просто происходит чтение следующей записи. Переменная имеет тип, заданный пользователем (UDT).

Для записи и добавления новых записей используется

Put #номер_файла, [<номер_записи>], <переменная>

Если в файле записи с заданным номером не существует, то создаются все записи от конечной до заданной (как пустые), а запись происходит в заданную запись.

Перемещение по файлу осуществляется с помощью оператора

Seek #номер_файла, <номер_записи>

После этого Get и Put без второго параметра будут работать с записью, на которую вы переместились.

7.4 Практична робота № 9: "Робота з файлами"

```
# Include
# Include

void main (void)
{
FILE * file;
char * file_name = "file.txt";
char load_string [50] = "none";

file = fopen (file_name, "w");

fputs ("string", file);

fclose (file);

file = fopen (file_name, "r");
if (file! = 0)
{
```

fgets (load_string, 50, file);
cout

Опис функцій роботи з файломі знаходяться в бібліотеці stdio.h

Спочатку треба створити покажчик на змінну типу FILE (FILE * file;).
Відкриття файлу здійснюється викликом функції fopen (file = fopen (file_name, "w");)
Перший параметр цієї функції - ім'я файлу, другий - вказує в якому режимі повинен бути відкритий файл. "W" - відкрити для запису, "r" - відкрити для читання, "a" - доповнення файлу (це найбільш використовувані режими, хоча є й інші). Запис і зчитування даних з файлу здійснюється наступними функціями: fputc, fputs, fgetc, fgets, fprintf, fscanf (опис цих функцій дивіться в stdio.h).
Закриття файлу здійснюється викликом функції fclose (fclose (file);).

Робота з файлами за допомогою MFC (класи CFile, CStdioFile, ...) і стандартний клас MFC CFileDialog.

До бібліотеки MFC включено кілька класів для забезпечення роботи з файлами. Розглянуті нижче класи успадковуються від базового класу CFile.

Клас CFile

Клас CFile призначений для забезпечення роботи з файлами. Він дозволяє спростити використання файлів, представляючи фото як об'єкт, який можна створити, читати, записувати і т.д.

Щоб отримати доступ до файлу, спочатку треба створити об'єкт класу CFile. Конструктор класу дозволяє відразу після створення такого об'єкта відкрити файл. Але можна відкрити файл і пізніше, скориставшись методом Open.
Відкриття та створення файлів

Після створення об'єкту класу CFile можна відкрити файл, викликавши метод Open. Методу треба вказати шлях до файлу, і режим його використання. Прототип методу Open має наступний вигляд:

```
virtual BOOL Open (LPCTSTR lpszFileName,
        UINT nOpenFlags, CFileException * pError = NULL);
```

Як параметр lpszFileName треба вказати ім'я файлу,. Можна вказати тільки ім'я файлу або повне ім'я файлу, що включає повний шлях до нього.

Другий параметр nOpenFlags визначає дію, що виконується методом Open з файлом, а також атрибути файлу. Нижче представлені деякі возможеие значення параметра nOpenFlags:

CFile :: modeCreate - Створюється новий файл. Якщо вказаний файл існує, то його вміст стирається і довжина файлу встановлюється рівною нулю.

CFile :: modeNoTruncate - Цей файл призначений для використання спільно з файлом CFile :: modeCreate. Якщо створюється вже існуючий файл, то його вміст не буде видалено.

CFile :: modeRead - Файл відкривається тільки для читання.

CFile :: modeReadWrite - Файл відкривається для запису і для читання.

CFile :: modeWrite - Файл відкривається тільки для запису.

CFile :: typeText - Використовується класами, породженими від класу CFile, наприклад CStdioFile, для роботи з файлами в текстовому режимі. Текстовий режим забезпечує перетворення комбінації символу повернення каретки і символу перекладу рядка.

CFile :: Binary - Використовується класами, породженими від класу CFile, наприклад CStdioFile, для роботи з файлами в двійковому режимі.

Необов'язковий параметр pError, який є покажчиком на об'єкт класу CFileException, використовується тільки в тому випадку, якщо виконання операції з файлом викличе помилку. При цьому в об'єкт, що указується pError, буде записана додаткова інформація.

Метод Open повертає не нульове значення, якщо файл відкритий і нуль у разі помилки. Помилка при відкритті файлу може трапитися, наприклад, якщо методу Open вказаний для читання неіснуючий файл.

Ідентифікатор відкритого файлу

До складу класу CFile входить елемент даних m_hFile типу UINT. У ньому зберігається ідентифікатор відкритого файлу. Якщо об'єкт класу CFile вже створений, але файл ще не відкрито, то у змінній m_hFile записана константа hFileNull.

Зазвичай ідентифікатор відкритого файлу безпосередньо не використовується. Методи класу CFile дозволяють виконувати практично будь-які операції з файлами і не вимагають вказувати ідентифікатор файлу. Так як m_hFile є елементом класу, то реалізація його методів завжди має вільний доступ до нього.

Закриття файлів

Після завершення роботи з файлом, його треба закрити. Клас CFile має для цього спеціальний метод Close. Потрібно зауважити, що якщо був створений об'єкт класу CFile і відкритий файл, а потім об'єкт віддаляється, то пов'язаний з ним файл закривається автоматично за допомогою деструктора.

Читання і запис файлів

Для доступу до файлів призначено кілька методів класу CFile: Read, ReadHuge, Write, WriteHuge, Flush. Методи Read і ReadHuge призначені для читання даних з попередньо відкритого файлу. У 32-розрядних операційних системах обидва методи можуть одночасно вважати з файлу більше 65535 байт. Специфікація ReadHuge вважається застарілою і залишена тільки для сумісності з 16-розрядними операційними системами.

Дані, прочитані з файлу, записуються в буфер lpBuf. Параметр nCount визначає кількість байт, яке треба вважати з файлу. Фактично з файлу може бути лічено менше байт, ніж запрошено параметром nCount. Це відбувається, якщо під час читання досягнуто кінець файлу. Методи повертають кількість байт, прочитаних з файлу.

Для запису в файл призначені методи Write і WriteHuge. У 32-розрядних операційних системах обидва методи можуть одночасно записувати в файл більше 65535 байт. Методи записує у відкритий файл nCount байт з буфера lpBuf. У разі виникнення помилки запису, наприклад переповнення диска, методи викликає обробку виключення.

Метод Flush

Коли використовується метод Write або WriteHuge для запису даних на диск, вони деякий час можуть перебувати в тимчасовому буфері. Щоб переконатися, що необхідні зміни внесені в файл на диску, потрібно скористатися методом Flush.

Операції з файлами

До складу класу входять методи, що дозволяють виконувати над файлами різні операції, наприклад копіювання, перейменування, видалення, зміна атрибутів.

Для зміни імені файлу клас CFile включає статичний метод Rename, що виконує функції цієї команди. Метод не можна використовувати для перейменування каталогів. У разі виникнення помилки метод викликає виключення.

Для видалення файлів в класі CFile включений статичний метод Remove, що дозволяє видалити вказаний файл. Цей метод не дозволяє видаляти каталоги. Якщо видалити файл неможливо, то метод викликає виключення.

Щоб визначити дату і час створення файлу, його довжину і атрибути, призначений статичний метод GetStatus. Існує два різновиди методу - перший визначений як віртуальний, а другий - як статичний метод.

Віртуальна версія методу GetStatus визначає стан відкритого файлу, пов'язаного з даним об'єктом класу CFile. Цей метод викликається тільки тоді, коли об'єкт класу CFile створений і файл відкритий.

Статична версія методу GetStatus дозволяє визначити характеристики файлу, не пов'язаного з об'єктом класу CFile. Щоб скористатися цим методом, необов'язково попередньо відкривати файл.

Блокування

До складу класу включені методи LockRange і UnlockRange, що дозволяють заблокувати один або кілька фрагментів даних файлу для доступу з інших процесів. Якщо програма намагається повторно блокувати дані, вже заблоковані раніше цим або іншим додатком, викликається виняток. Блокування являє собою один з механізмів, що дозволяють декільком додаткам або процесів одночасно працювати з одним файлом, не заважаючи один одному.

Встановити блокування можна за допомогою методу LockRange. Щоб зняти встановлені блокування, треба скористатися методом UnlockRange. Якщо в одному файлі встановлені кілька блокувань, то кожна з них повинна зніматися окремим викликом методу UnlockRange.

Позиціонування

Щоб перемістити покажчик поточної позиції файлу в нове положення, можна скористатися одним з таких методів класу CFile - Seek, SeekToBegin, SeekToEnd. До складу класу CFile також входять методи, що дозволяють встановити і змінити довжину файлу, - GetLength, SetLength.

При відкритті файлу покажчик поточної позиції файлу знаходиться на самому початку файлу. Коли порція даних прочитана або записана, то покажчик поточної позиції переміщується в бік кінця файлу і вказує на дані, які будуть читатися чи записуватися черговий операцією читання або запису у файл.

Щоб перемістити покажчик поточної позиції файлу в будь-яке місце, можна скористатися універсальним методом Seek. Він дозволяє перемістити покажчик на певне число байт щодо початку, кінця або поточної позиції покажчика.

Щоб перемістити покажчик на початок або кінець файлу, найбільш зручно використовувати спеціальні методи. Метод SeekToBegin переміщує вказівник на початок файлу, а метод SeekToEnd - в його кінець.

Але для визначення довжини відкритого файлу зовсім необов'язково переміщати його покажчик. Можна скористатися методом GetLength. Цей метод також повертає довжину відкритого файлу в байтах. Метод SetLength дозволяє змінити довжину відкритого файлу. Якщо за допомогою цього методу розмір файлу збільшується, то значення останніх байт не визначено.

Поточну позицію покажчика файлу можна визначити за допомогою методу GetPosition. Повертане методом GetPosition 32-розрядне значення визначає зміщення покажчика від початку файлу.

Характеристики відкритого файлу

Щоб визначити розташування відкритого файлу на диску, треба викликати метод GetFilePath. Цей метод повертає об'єкт класу CString, в якому міститься повний шлях файлу, включаючи ім'я диска, каталоги, ім'я файлу і його розширення.

Якщо потрібно визначити тільки ім'я і розширення відкритого файлу, можна скористатися методом GetFileName. Він повертає об'єкт класу CString, в якому знаходиться ім'я файлу. У випадку, коли потрібно дізнатися тільки ім'я відкритого файлу без розширення, користуються методом GetFileTitle.

Наступний метод класу CFile дозволяє встановити шлях файлу. Це метод не створює, не копіює і не змінює імені файлу, він тільки заповнює відповідний елемент даних в об'єкті класу CFile.

Клас CmemFile

До бібліотеки MFC входить клас CMemFile, успадковані від базового класу CFile. Клас CMemFile представляє файл, розміщений, в оперативної пам'яті. З об'єктами класу CMemFile так само, як і з об'єктами класу CFile. Відмінність полягає в тому, що файл, пов'язаний з об'єктом CMemFile, розташований не на диску, а в оперативній пам'яті комп'ютера. За рахунок цього операції з таким файлом відбуваються значно швидше, ніж зі звичайними файлами.

Працюючи з об'єктами класу CMemFile, можна використовувати практично всі методи класу CFile, які були описані вище. Можна записувати дані в такий файл або зчитувати їх. Крім цих методів до складу класу CMemFile включені додаткові методи.

Для створення об'єктів класу CMemFile призначено два різних конструктора. Перший конструктор CMemFile має всього один необов'язковий параметр nGrowBytes:

CMemFile (UINT nGrowBytes = 1024);

Цей конструктор створює в оперативній пам'яті порожній файл. Після створення файл автоматично відкривається (не потрібно викликати метод Open).

Коли починається запис в такий файл, автоматично виділяється блок пам'яті. Для отримання пам'яті методи класу CMemFile викликають стандартні функції malloc, realloc і free. Якщо виділеного блоку пам'яті недостатньо, його розмір збільшується. Збільшення блоку пам'яті файлу відбувається по частинах по nGrowBytes байт. Після видалення об'єкта класу CMemFile використовувана пам'ять автоматично повертається системі.

Другий конструктор класу CMemFile має більш складний прототип. Це конструктор використовується в тих випадках, коли програміст сам виділяє пам'ять для файлу:

CMemFile (BYTE * lpBuffer, UINT nBufferSize, UINT nGrowBytes = 0);

Параметр lpBuffer вказує на буфер, який буде використовуватися для файлу. Розмір буфера визначається параметром nBufferSize.

Необов'язковий параметр nGrowBytes використовується більш комплексно, ніж у першому конструкторі класу. Якщо nGrowBytes містить нуль, то створений файл буде містити дані з буфера lpBuffer. Довжина такого файлу буде дорівнює nBufferSize.

Якщо nGrowBytes більше нуля, то вміст буфера lpBuffer ігнорується. Крім того, якщо в такий файл записується більше даних, ніж поміщається у відведеному буфері, то його розмір автоматично збільшується. Збільшення блоку пам'яті файлу відбувається по частинах по nGrowBytes байт.

Клас CMemFile дозволяє отримати покажчик на область пам'яті, що використовується файлом. Через цей покажчик можна безпосередньо працювати з вмістом файла, не обмежуючи себе методами класу CFile. Для отримання вказівника на буфер файлу можна скористатися методом Detach. Перед цим корисно визначити довжину файлу (і відповідно розмір буфера пам'яті), викликавши метод GetLength.

Метод Detach закриває дане зображення і повертає покажчик на використовуваний ним блок пам'яті. Якщо знову буде потрібно відкрити файл і зв'язати з ним оперативний блок пам'яті, потрібно викликати метод Attach.

Потрібно зазначити, що для управління буфером файлу клас CMemFile викликає стандартні функції malloc, realloc і free. Тому, щоб не порушувати механізм управління пам'яттю, буфер lpBuffer повинен бути створений функціями malloc або calloc.

Клас CstdioFile

Тим, хто звик користуватися функціями потокового введення / виводу із стандартної бібліотеки C і C + +, слід звернути увагу на клас CStdioFile, успадкований від базового класу CFile. Цей клас дозволяє виконувати буферізірованний введення / виведення в текстовому і двійковому режимі. Для об'єктів класу CStdioFile можна викликати практично всі методи класу CFile.

В клас CStdioFile входить елемент даних m_pStream, який містить покажчик на відкритий файл. Якщо об'єкт класу CStdioFile створений, але файл ще не відкрито, або закритий, то m_pStream містить константу NULL.

Клас CStdioFile має три різних конструктора. Перший конструктор класу CStdioFile не має параметрів. Цей конструктор тільки створює об'єкт класу, але не відкриває ніяких файлів. Щоб відкрити файл, треба викликати метод Open базового класу CFile.

Другий конструктор класу CStdioFile можна викликати, якщо файл вже відкритий і потрібно створити новий об'єкт класу CStdioFile і пов'язати з ним відкритий файл. Цей конструктор можна використовувати, якщо файл був відкритий стандартною функцією fopen. Параметр методу повинен містити покажчик на файл, отриманий викликом стандартної функції fopen.

Третій конструктор можна використовувати, якщо треба створити об'єкт класу CStdioFile, відкрити новий файл і зв'язати його з тільки що створеним об'єктом.

Для читання і запису в текстовий файл клас CStdioFile включає два нових методи: ReadString і WriteString. Перший метод дозволяє прочитати з файлу рядок символів, а другий метод - записати.

Приклади запису та читання з файлу:

Наведемо фрагменти коду, в яких демонструється використання стандартних діалогових панелей вибору файлу і процедури читання і запису у файл.

```
Відкриття файлу і читання з нього
CString m_Text; / / створення стандартної панелі вибору файлу Open
 CFileDialog DlgOpen (TRUE, (LPCSTR) "txt", NULL,
OFN_HIDEREADONLY, (LPCSTR) "Text Files (*. Txt) | *. Txt | |");
  / / Відображення стандартної панелі вибору файлу Open
if (DlgOpen.DoModal () == IDOK)
   {/ / Створення об'єкта і відкриття файлу для читання
CStdioFile File (DlgOpen.GetPathName (), CFile ::
                    modeRead | CFile :: typeBinary);
/ / Читання з файлу рядка
 CString & ref = m_Text;
 File.ReadString (ref); / / передається посилання на рядок m_Text
 }
```

Запустіть програму - Build / Rebuild all (будуть помилки), виберіть Build / Set active configuration - Win 32 Realise, виберіть пункт меню "Project", далі "Settings ...", закладку "C / C + +", Category - Code Generation і в пункті "Use run-time library" виберіть "Multithreaded". Після цього зробіть знову Build / Rebuild all і програма буде працювати.

```
Відкриття файлу і запис з нього

CString m_Text; / / створення стандартної панелі вибору файлу SaveAs
CFileDialog DlgSaveAs (FALSE, (LPCSTR) "txt", NULL,
OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
(LPCSTR) "Text Files (*. Txt) | *. Txt | |");
/ / Відображення стандартної панелі вибору файлу SaveAs
if (DlgSaveAs.DoModal () == IDOK)
{/ / Створення об'єкта і відкриття файлу для запису
```

```
CStdioFile File (DlgSaveAs.GetPathName (),
CFile :: modeCreate | CFile :: modeWrite | CFile :: typeBinary);
 / / Запис в файл рядки
File.WriteString ((LPCTSTR) m_Text);
}
```

7.5 Запис у текстовий файл

Для запису в текстовий файл послідовного доступу мови С використовували функцію fprintf, яка записувала в файл певну кількість символів, залежно від довжини слова або кількості цифр в числі. Тепер давайте подивимося, що робить подібна функція fwrite, яка записує дані в текстовий файл довільного доступу.

Функція fwrite завжди, незалежно від кількості символів або кількості цифр, виділяє певну довжину для даного. Ця довжина задається в функції. Ну, а тепер давайте розглянемо приклад запису в текстовий файл мови С за допомогою функції fwrite:

```
# Include <conio.h>
 # Include <stdio.h>

 int main ()
 {
     int bal;
     FILE * file;

     if ((file = fopen ("3.txt", "w")) == NULL)
       printf ("Файл неможливо відкрити або створити \ n");
     else {
       for (int i = 0; i <5; i + +) {
         scanf ("% d", & bal);
         fwrite (& bal, sizeof (int), 1, file);
       }
     }
     fclose (file);
     getch ();
 return 0;
 }
```

Давайте тепер розберемо функцію запису в текстовий файл fwrite досконально:

```
fwrite (& bal, sizeof (int), 1, file);
```

Першим параметром ми передаємо адресу нашого значення, яке ми хочемо записати у файл (неважливо число це або рядок); далі другим параметром слід розмір типу нашого даного (в даному випадку у нас тип int, що повідомляє функції виділяти під кожне значення по 2 байта) ; далі третім параметром слід кількість елементів, яке ми зібралися записувати в файл (якщо б у нас був масив значень, то ми могли б відразу записати кілька елементів); ну, і останнім аргументів нашої функції слід покажчик на файл, в який проводитиметься запис даних

В принципі, більше відмінностей від запису в файл послідовного доступу тут ми не спостерігаємо. Тому давайте відразу приступимо до читання даних з файлу довільного доступу:

```
# Include <conio.h>
 # Include <stdio.h>

 int main ()
 {
     int bal;
     FILE * file = fopen ("3.txt", "r");

     if (file == NULL)
       printf ("Помилка при відкритті файлу");
     else {
       while (! feof (file)) {
       fread (& bal, sizeof (int), 1, file);
       printf ("% d", bal);
       }
     }

     fclose (file);
     getch ();
 return 0;
 }
```

8. МОДУЛЬ 8: Загальні характеристики мови C++

8.1 Загальні характеристики та особливості мови C++

Мова С + + з'явився на початку 80-х років. Створений Бьерном Страуструпом з початковою метою позбавити себе і своїх друзів від програмування на асемблері, Сі чи різних інших мовах високого рівня.

Очевидно, що найбільше С + + запозичив з мови Сі, а також з безпосереднього його попередника мови BCPL. Ці запозичення забезпечили С + + потужними засобами низького рівня, що дозволяють вирішувати складні завдання системного програмування. Але що в першу чергу відрізняє С + + від Сі - це різна ступінь уваги до типів і структур даних. Це пов'язано з появою понять класу, похідного класу і віртуальної функції, перейнятих в свою чергу з мови Симула 67. Це дає в С + + більш ефективні можливості для контролю типів і забезпечує модульність програми.

На думку автора мови, відмінність між ідеологією Сі і С + + полягає приблизно в наступному: програма на Сі відображає "спосіб мислення" процесора, а С + + - спосіб мислення програміста. Відповідаючи вимогам сучасного програмування, С + + робить акцент на розробці нових типів даних найбільш повно відповідних концепцій обраної галузі знань і завданням програми. Клас є ключовим поняттям С + +. Опис класу містить опис даних, потрібних для подання об'єктів цього типу і набір операцій для роботи з подібними об'єктами.

На відміну від традиційних структур Сі і Паскаля, членами класу є не тільки дані, але і функції. Функції - члени класу мають привілейований доступ до даних усередині об'єктів цього класу і забезпечують інтерфейс між цими об'єктами і решті програмою. При подальшій роботі зовсім не обов'язково пам'ятати про внутрішню структуру класу і механізм роботи вбудованих функцій. У цьому сенсі клас подібний електричному приладу - мало хто знає про його устрій, але всі знають, як ним користуватися.

Мова С + + є засобом об'єктного програмування, новітньої методики проектування та реалізації програм, яка в поточному десятилітті, швидше за все, замінить традиційне процедурне програмування. Головною метою творця мови доктора Бьерна Страустрапа було оснащення мови С + + конструкціями, що дозволяють збільшити продуктивність праці програмістів і полегшити процес оволодіння великими програмними продуктами.

Абстракція, реалізація, успадкування і поліморфізм є необхідними властивостями якими володіє мовою С + +, завдяки чому він не тільки універсальний, як і мова Сі, але і є об'єктним мовою.

8.2 Вбудовані функції

Коли ви визначаєте в своїй програмі функцію, компілятор C + + переводить код функції в машинну мову, зберігаючи тільки одну копію інструкцій функції усередині вашої програми. Кожен раз, коли ваша програма викликає функцію, компілятор C + + поміщає в програму спеціальні інструкції, які заносять параметри функції в стек і потім виконують перехід до інструкцій цієї функції. Коли оператори функції завершуються, виконання програми триває з першого оператора, який слідує за викликом функції. Приміщення аргументів у стек і перехід у функцію і з неї вносить витрати, через які ваша програма виконується трохи повільніше, ніж якщо б вона розміщувала ті ж оператори прямо всередині програми при кожному посиланні на функцію. Наприклад, припустимо, що наступна програма CALLBEEP.CPP викликає функцію show_message, яка вказане число раз видає сигнал на динамік комп'ютера і потім виводить повідомлення на дисплей:

```
# Include <iostream.b>

void show_message (int count, char * message)

{
  int i;
  for (i = 0; i <count; i + +) cout << '\ a';
  cout << message << endl;
}

void main (void)

{
  show_message (3, "Вчимося програмувати на мові C + +");
  show_mes sage (2, "Урок 35");
}
```

Наступна програма NO_CALL.CPP не викликає функцію show_message. Замість цього вона поміщає всередині себе ті ж оператори функції при кожному посиланні на функцію:

```
# Include <iostream.h>

void main (void)

{
```

```
  int i;
  for (i = 0; i <3; i + +) cout << '\ a';
  cout << "Вчимося програмувати на мові C + +" << endl;
  for (i = 0; i <2; i + +) cout << '\ a';
  cout << "Урок 35" << endl;
}
```

Обидві програми виконують одне і те ж. Оскільки програма NO_CALL не викликає функцію show_message, вона виконується трохи швидше, ніж програма CALLBEEP. В даному випадку різницю в часі виконання визначити неможливо, але, якщо в звичайній ситуації функція буде викликатися 1000 раз, ви, ймовірно, помітите невелике збільшення продуктивності. Однак програма NO_CALL більш заплутана, ніж її двійник CALL_BEEP, отже, більш важка для сприйняття.

При створенні програм ви завжди повинні спробувати визначити, коли краще використовувати звичайні функції, а коли краще скористатися вбудованими функціями. Для більш простих програм переважно використовувати звичайні функції. Однак, якщо ви створюєте програму, для якої продуктивність має першорядне значення, вам варто було б зменшити кількість викликів функцій. Один зі способів зменшення кількості викликів функцій полягає в тому, щоб помістити відповідні оператори прямо в програму, як тільки що було зроблено в програмі NO_CALL. Однак, як ви могли переконатися, заміна лише однієї функції внесла значну плутанину в програму. На щастя, C + + надає ключове слово inline, яке забезпечує кращий спосіб.

Використання ключового слова inline

При оголошенні функції всередині програми C + + дозволяє вам випередити ім'я функції ключовим словом inline. Якщо компілятор C + + зустрічає ключове слово inline, він поміщає в здійсненний файл (машинний мова) оператори цієї функції в місці кожного її виклику. Таким чином, можна поліпшити читаність ваших програм на C + +, використовуючи функції, і в той же час збільшити продуктивність, уникаючи витрат на виклик функцій. Наступна програма INLINE.CPP визначає функції tax і min як inline:

```
# Include <iostream.h>

inline int max (int a, int b)

{
  if (a> b) return (a);
  else return (b);
}
```

```
inline int min (int a, int b)

{
  if (a <b) return (a);
  else return (b);
}

void main (void)

{
  cout << "Мінімум з 1001 і 2002 дорівнює" << min (1001, 2002) << endl;
    cout << "Максимум з 1001 і 2002 дорівнює" << max (1001, 2002) <<
endl;
}
```

В даному випадку компілятор С + + замінить кожен виклик функції на відповідні оператори функції. Продуктивність програми збільшується без її ускладнення.

Про вбудовані функції

Якщо компілятор С + + перед визначенням функції зустрічає ключове слово inline, він буде замінювати звернення до цієї функції (виклики) на послідовність операторів, еквівалентну виконання функції. Таким чином ваші програми покращують продуктивність, позбавляючись від витрат на виклик функції і в той же час виграючи в стилі програми, завдяки використанню функцій.

8.3 Операція виділення памяті

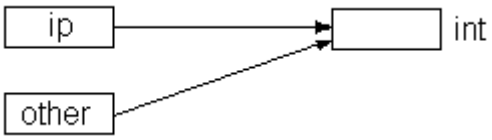
Часто вираз, що містить операцію new, має наступний вигляд:
указатель_на_тіп_ = new Імя_Тіпа (ініціалізатор)
Ініціалізатор - це необов'язкове инициализирующее вираз, який може використовуватися для всіх типів, крім масивів.

При виконанні оператора

```
int * ip = new int;
```

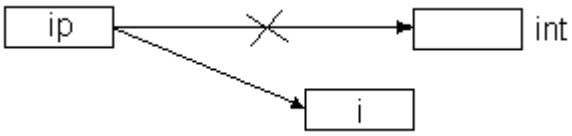
створюються 2 об'єкти: динамічний безіменний об'єкт і покажчик на нього з ім'ям ip, значенням якого є адреса динамічного об'єкта. Можна створити та іншої покажчик на той же динамічний об'єкт:

```
int * other = ip;
```



Якщо вказівником ip присвоїти інше значення, то можна втратити доступ до динамічного об'єкту:

```
int * ip = new (int);
    int i = 0;
    ip = &i;
```



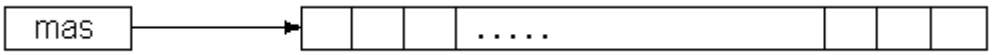
В результаті динамічний об'єкт як і раніше буде існувати, але звернеться до нього вже не можна. Такі об'єкти називаються сміттям.

При виділенні пам'яті об'єкт можна ініціалізувати:

```
int * ip = new int (3);
```

Можна динамічно розподілити пам'ять і під масив:

```
double * mas = new double [50];
```



Тепер з цією динамічно виділеної пам'яттю можна працювати як зі звичайним масивом:

```
* (mas +5) = 3.27;
    mas [6] = mas [5] + sin (mas [5]);
```

У разі успішного завершення операція new повертає покажчик зі значенням, відмінним від нуля.

Результат операції, рівний 0, тобто нульового вказівником NULL, говорить про те, що не знайдений безперервний вільний фрагмент пам'яті потрібного розміру.

8.4 Перенавантаження функцій. Шаблони функцій

Перевантаження функції

При визначенні функцій в своїх програмах ви повинні вказати тип повертається функцією значення, а також кількість параметрів і тип кожного з них. У минулому (якщо ви програмували на мові С), коли у вас була функція з ім'ям add_values, яка працювала з двома цілими значеннями, а ви хотіли б використовувати подібну функцію для складання трьох цілих значень, вам слід було створити функцію з іншим ім'ям. Наприклад, ви могли б використовувати add_two_values і add_three_values. Аналогічно якщо ви хотіли використовувати подібну функцію для складання значень типу float, то вам була б необхідна ще одна функція з ще одним ім'ям. Щоб уникнути дублювання функції, С + + дозволяє вам визначати кілька функцій з одним і тим же ім'ям. В процесі компіляції С + + бере до уваги кількість аргументів, що використовуються кожною функцією, і потім викликає саме потрібну опцію. Надання компілятору вибору серед кількох функцій називається перевантаженням. У цьому уроці ви навчитеся використовувати перевантажені функції. До кінця цього уроку ви освоїте наступні основні концепції:

• Перевантаження функцій дозволяє вам використовувати один і той же ім'я для кількох функцій з різними типами параметрів.

• Для перевантаження функцій просто визначте дві функції з одним і тим же ім'ям і типом значення, що повертається, які відрізняються кількістю параметрів або їх типом.

Перевантаження функцій є особливістю мови С + +, якої немає у мові С. Як ви побачите, перевантаження функцій досить зручна і може поліпшити читабельність ваших програм.

Перевантаження функцій дозволяє вашим програмам визначати кілька функцій з одним і тим же ім'ям і типом, що повертається. Наприклад, наступна програма перевантажує функцію з ім'ям add_values. Перше визначення функції складає два значення типу int. Друге визначення функції складає три значення. В процесі компіляції С + + коректно визначає функцію, яку необхідно використовувати:

```
# Include <iostream.h>

int add_values (int a, int b)

{
  return (a + b);
)

int add_values (int a, int b, int c)

(
  return (a + b + c);
)
```

```
void main (void)

{
  cout << "200 + 801 =" << add_values (200, 801) << endl;
  cout << "100 + 201 + 700 =" << add_values (100, 201, 700) << endl;
}
```

Як бачите, програма визначає дві функції з іменами add_values Перша функція складає два значення типу int, в той час як друга складає три значення. Ви не зобов'язані небудь робити спеціально для того, щоб попередити компілятор про перевантаження, просто використовуйте її. Компілятор розгадає, яку функцію слід використовувати, грунтуючись на пропонованих програмою параметрах.

Подібним чином наступна програма MSG_OVR.CPP перевантажує функцію show_message. Перша функція з ім'ям show_message виводить стандартне повідомлення, параметри їй не передаються. Друга виводить передане їй повідомлення, а третя виводить два повідомлення:

```
# Include <iostream.h>

void show_message (void)

{
   cout << "Стандартне повідомлення:" << "Вчимося програмувати на C + +" << endl;
}

void show_message (char * message)

{
  cout << message << endl;
}

void show_message (char * first, char * second)

{
  cout << first << endl;
  cout << second << endl;
}

void main (void)

{
  show_message ();
  show_message ("Вчимося програмувати на мові C + +!");
```

show_message ("В С + + немає забобонів!", "Перевантаження - це круто!");
}

Одним з найбільш загальних випадків використання перевантаження є застосування функції для отримання певного результату, виходячи з різних параметрів. Наприклад, припустимо, що у вашій програмі є функція з ім'ям day_of_week, яка повертає поточний день тижня (0 для неділі, 1 для понеділка, ..., 6 для суботи). Ваша програма могла б перевантажити цю функцію таким чином, щоб вона вірно повертала день тижня, якщо їй переданий юліанський день в якості параметра, або якщо їй передані день, місяць і рік:

```
int day_of_week (int julian_day)

{
  / / Оператори
}

int day_of_week (int month, int day, int year)

{
  / / Оператори
}
```

У міру вивчення об'єктно-орієнтованого програмування в С + +, представленого в наступних уроках, ви будете використовувати перевантаження функцій для розширення можливостей своїх програм.

Перевантаження функцій покращує легкість для читання програм

Перевантаження функцій С + + дозволяє вашим програмам визначати кілька функцій з одним і тим же ім'ям. Перевантажені функції повинні повертати значення однакового типу *, але можуть відрізнятися кількістю і типом параметрів. До появи перевантаження функцій в С + + програмісти мови С повинні були створювати декілька функцій з майже однаковими іменами. На жаль програмісти, що бажають використовувати такі функції, повинні були пам'ятати, яка комбінація параметрів відповідає якої функції. З іншого боку, перевантаження функцій спрощує завдання програмістів, вимагаючи, щоб вони пам'ятали тільки одне ім'я функції.

* Перевантажені функції не зобов'язані повертати значення однакового типу з тієї причини, що компілятор однозначно ідентифікує функцію по її імені та набору її аргументів. Для компілятора функції з однаковими іменами, але різними типами аргументів - різні функції, тому тип значення, що повертається - прерогатива кожної функції. - Прім.перев.

Шаблони функцій

Іноді може здатися, що сильно типізований мова створює перешкоди для реалізації зовсім простих функцій. Наприклад, хоча наступний алгоритм функції min () тривіальний, сильна типізація вимагає, щоб його різновиди були реалізовані для всіх типів, які ми збираємося порівнювати:

```
int min (int a, int b) {
  return a <b? a: b;
}
double min (double a, double b) {
  return a <b? a: b;
}
```

Привабливу альтернативу явного визначення кожного примірника функції min () представляє використання макросів, розширюваних препроцесором:

```
# Define min (a, b) ((a) <(b)? (A): (b))
```

Але цей підхід таїть в собі потенційну небезпеку. Певний вище макрос правильно працює при простих зверненнях до min (), наприклад:

```
min (10, 20);
min (10.0, 20.0);
```

але може піднести сюрпризи в більш складних випадках: такий механізм веде себе не як виклик функції, він лише виконує текстову підстановку аргументів. В результаті значення обох аргументів оцінюються двічі: один раз при порівнянні a і b, а другий - при обчисленні повертається макросом результату:

```
# Include <iostream>
# Define min (a, b) ((a) <(b)? (A): (b))
const int size = 10;
int ia [size];
int main () {
  int elem_cnt = 0;
  int * p = & ia [0];
  / / Підрахувати число елементів масиву
  while (min (p + +, & ia [size])! = & ia [size])
    + + Elem_cnt;
  cout << "elem_cnt:" << elem_cnt
       << "\ Texpecting:" << size << endl;
  return 0;
}
```

На перший погляд, ця програма підраховує кількість елементів в масиві ia цілих чисел. Але в цьому випадку макрос min () розширюється невірно, оскільки операція постинкрементом застосовується до аргументу-покажчику двічі при кожній підстановці. В результаті програма друкує рядок, що свідчить про неправильні обчисленнях:

```
elem_cnt: 5 expecting: 10
```

Шаблони функцій надають у наше розпорядження механізм, за допомогою якого можна зберегти семантику визначень і викликів функцій (інкапсуляція фрагмента коду в одному місці програми і гарантовано одноразове обчислення аргументів), не приносячи в жертву сильну типізацію мови C + +, як у випадку застосування макросів.

Шаблон дає алгоритм, використовуваний для автоматичної генерації примірників функцій з різними типами. Програміст параметризуються всі або тільки деякі типи в інтерфейсі функції (тобто типи формальних параметрів і значення, що повертається), залишаючи її тіло незмінним. Функція добре підходить на роль шаблону, якщо її реалізація залишається інваріантної на деякому безлічі екземплярів, що розрізняються типами даних, як, скажімо, у випадку min ().

Так визначається шаблон функції min ():

```
template <class Type>
  Type min2 (Type a, Type b) {
    return a <b? a: b;
}
int main () {
  / / Правильно: min (int, int);
  min (10, 20);
  / / Правильно: min (double, double);
  min (10.0, 20.0);
  return 0;
}
```

Якщо замість макросу препроцесора min () підставити в текст попередньої програми цей шаблон, то результат буде правильним:

elem_cnt: 10 expecting: 10

(У стандартній бібліотеці C + + є шаблони функцій для багатьох часто використовуваних алгоритмів, наприклад для min (). Ці алгоритми описуються в главі 12. А в даній вступній главі ми наводимо власні спрощені версії деяких алгоритмів з стандартної бібліотеки.)

Як оголошення, так і визначення шаблону функції завжди повинні починатися з ключового слова template, за яким слідує список розділених комами ідентифікаторів, укладений в кутові дужки '<' і '>', - список параметрів шаблону, обов'язково непорожній. У шаблону можуть бути параметри-типи, що представляють певний тип, і параметри-константи, що представляють фіксоване константне вираз.

Параметр-тип складається з ключового слова class або ключового слова typename, за яким слід ідентифікатор. Ці слова завжди означають, що подальше ім'я відноситься до вбудованого або певного користувачем типу. Ім'я параметра шаблона вибирає програміст. У наведеному прикладі ми використовували ім'я Type, але могли вибрати будь-який інший:

```
template <class Glorp>
  Glorp min2 (Glorp a, Glorp b) {
```

```
    return a <b? a: b;
}
```

При конкретизації (породження конкретного екземпляра) шаблону замість параметра-типу підставляється фактичний вбудований або визначений користувачем тип. Будь-який з типів int, double, char *, vector <int> або list <double> є допустимим аргументом шаблону.

Параметр-константа виглядає як звичайне оголошення. Він говорить про те, що замість імені параметра має бути підставлено значення константи з визначення шаблону. Наприклад, size - це параметр-константа, який представляє розмір масиву arr:

```
template <class Type, int size>
  Type min (Type (& arr) [size]);
```

Слідом за списком параметрів шаблону йде оголошення або визначення функції. Якщо не звертати уваги на присутність параметрів у вигляді специфікаторів типу або констант, то визначення шаблону функції виглядає точно так само, як і для звичайних функцій:

```
template <class Type, int size>
Type min (const Type (& r_array) [size])
{
  / * Параметрезованих функція для відшукання
   * Мінімального значення в масиві * /
  Type min_val = r_array [0];
  for (int i = 1; i <size; + + i)
    if (r_array [i] <min_val)
      min_val = r_array [i];
  return min_val;
}
```

У цьому прикладі Type визначає тип значення, що повертається функцією min (), тип параметра r_array і тип локальної змінної min_val; size задає розмір масиву r_array. В ході роботи програми при використанні функції min () замість Type можуть бути підставлені будь вбудовані і певні користувачем типи, а замість size - ті чи інші вирази. (Нагадаємо, що працювати з функцією можна двояко: викликати її або взяти її адресу).

Процес підстановки типів і значень замість параметрів називається конкретизацією шаблону. (Докладніше ми зупинимося на цьому в наступному розділі.)

Список параметрів нашої функції min () може здатися надто коротким. Як було сказано в розділі , коли параметром є масив, передається покажчик на його перший елемент, перша ж розмірність фактичного аргументу-масиву всередині визначення функції невідома. Щоб обійти ці труднощі, ми оголосили перший параметр min () як посилання на масив, а другий - як його розмір. Недолік подібного підходу в тому, що при використанні шаблону з масивами одного і того ж типу int, але різних розмірів генеруються (або конкретизуються) різні екземпляри функції min ().

Ім'я параметра дозволено вживати всередині оголошення або визначення шаблону. Параметр-тип служить специфікатором типу; його можна використовувати точно так само, як специфікатор будь-якого вбудованого або настроюваного типу, наприклад в оголошенні змінних або в операціях приведення типів. Параметр-константа застосовується як константні значення - там, де потрібні вирази, наприклад для завдання розміру в оголошенні масиву або в якості початкового значення елемента перерахування.

```
/ / Size визначає розмір параметра-масиву і ініціалізує
/ / Змінну типу const int
template <class Type, int size>
  Type min (const Type (& r_array) [size])
{
    const int loc_size = size;
   Type loc_array [loc_size];
    / / ...
}
```

Якщо в глобальній області видимості оголошений об'єкт, функція або тип з тим же ім'ям, що у параметра шаблона, то глобальне ім'я виявляється прихованим. У наступному прикладі тип змінної tmp НЕ double, а той, що у параметра шаблона Type:

```
typedef double Type;
template <class Type>
  Type min (Type a, Type b)
{
  / / Tmp має той же тип, що параметр шаблона Type, а не заданий
  / / Глобальним typedef
  Type tm = a <b? a: b;
  return tmp;
}
```

Об'єкт або тип, оголошені усередині визначення шаблону функції, не можуть мати те ж ім'я, що й якийсь з параметрів:

```
template <class Type>
  Type min (Type a, Type b)
{
  / / Помилка: повторне оголошення імені Type, що збігається з ім'ям
  / / Параметра шаблона
  typedef double Type;
  Type tmp = a <b? a: b;
  return tmp;
}
```

Ім'я параметра-типу шаблону можна використовувати для завдання типу значення, що повертається:

```
/ / Правильно: T1 представляє тип значення, що повертається min (),
/ / A T2 і T3 - параметри-типи цієї функції
```

template <class T1, class T2, class T3>
T1 min (T2, T3);

В одному списку параметрів деякий ім'я дозволяється вживати тільки один раз. Наприклад, таке визначення буде позначено як помилка компіляції:

/ / Помилка: неправильне повторне використання імені параметра Type
template <class Type, class Type>
Type min (Type, Type);

Проте одне і те ж ім'я можна багаторазово застосовувати всередині оголошення або визначення шаблону:

/ / Правильно: повторне використання імені Type всередині шаблона
template <class Type>
Type min (Type, Type);
template <class Type>
Type max (Type, Type);

Імена параметрів в оголошенні й визначенні не зобов'язані збігатися. Так, всі три оголошення min () відносяться до одного й того ж шаблону функції:

/ / Всі три оголошення min () відносяться до одного й того ж шаблону функції
/ / Випереджаючі оголошення шаблону
template <class T> T min (T, T);
template <class U> U min (U, U);
/ / Фактичне визначення шаблону
template <class Type>
Type min (Type a, Type b) {/ * ... * /}

Кількість поява одного і того ж параметра шаблона в списку параметрів функції не обмежена. У наступному прикладі Type використовується для представлення двох різних параметрів:

Include <vector>
/ / Правильно: Type використовується неодноразово у списку параметрів шаблону
template <class Type>
Type sum (const vector <Type> &, Type);

Якщо шаблон функції має кілька параметрів-типів, то кожному з них повинно передувати ключове слово class або typename:

/ / Правильно: ключові слова typename і class можуть перемежовуватися
template <typename T, class U>
T minus (T *, U);
/ / Помилка: має бути <typename T, class U> або
/ / <typename T, typename U>
template <typename T, U>
T sum (T *, U);

У списку параметрів шаблону функції ключові слова typename і class мають однаковий зміст і, отже, взаємозамінні. Будь-яке з них може використовуватися для оголошення різних параметрів-типів шаблону в одному і тому ж списку (як було продемонстровано на прикладі шаблону

функції minus ()). Для позначення параметра-типу більш природно, на перший погляд, вживати ключове слово typename, а не class, адже воно ясно вказує, що за ним іде ім'я типу. Однак це слово було додано в мову лише недавно, як частина стандарту C + +, тому в старих програмах ви швидше за все зустрінете слово class. (Не кажучи вже про те, що class коротше, ніж typename, а людина за своєю природою ледачий.)

Ключове слово typename спрощує розбір визначень шаблонів. (Ми лише коротко зупинимося на тому, навіщо воно знадобилося. Бажаючим дізнатись про це детальніше рекомендуємо звернутися до книги Страуструпа "Design and Evolution of C + +".)

При такому розборі компілятор повинен відрізняти вираження-типи від тих, які такими не є; виявити це не завжди можливо. Наприклад, якщо компілятор зустрічає у визначенні шаблону вираз Parm :: name і якщо Parm - це параметр-тип, який представляє клас, то чи слід вважати, що name представляє член-тип класу Parm?

```
template <class Parm, class U>
  Parm minus (Parm * array, U value)
{
  Parm :: name * p; / / це оголошення покажчика або множення?
  / / Насправді множення
}
```

Компілятор не знає, чи є name типом, оскільки визначення класу, представленого параметром Parm, недоступно до моменту конкретизації шаблону. Щоб таке визначення шаблону можна було розібрати, користувач повинен підказати компілятору, які вирази включають типи. Для цього служить ключове слово typename. Наприклад, якщо ми хочемо, щоб вираз Parm :: name в шаблоні функції minus () було ім'ям типу і, отже, вся рядок трактувалася як оголошення покажчика, то потрібно модифікувати текст наступним чином:

```
template <class Parm, class U>
  Parm minus (Parm * array, U value)
{
  typename Parm :: name * p; / / тепер це оголошення покажчика
}
```

Ключове слово typename використовується також у списку параметрів шаблону для вказівки того, що параметр є типом.

Шаблон функції можна оголошувати як inline або extern - як і звичайну функцію. Специфікатор поміщається після списку параметрів, а не перед словом template.

```
/ / Правильно: специфікатор після списку параметрів
template <typename Type>
  inline
  Type min (Type, Type);
/ / Помилка: специфікатор inline не на місці
  inline
```

```
template <typename Type>
Type min (Array <Type>, int);
```

8.5 Визначення значень по умолчанню у мові C++

Забезпечити значення за замовчуванням для параметрів функції дуже легко. Ви просто привласнюєте значення параметру за допомогою оператора присвоювання C + + прямо при оголошенні функції, як показано нижче:

```
void some_function (int size = 12, float cost = 19.95) / / ----> Значення за умовчанням

{
   / / Оператори функції
}
```

Наступна програма DEFAULTS. CPP присвоює значення за замовчуванням параметрами a, b і c всередині функції show_parameters. Потім програма чотири рази викликає цю функцію, спочатку не вказуючи параметрів взагалі, потім вказуючи значення тільки для a, потім значення для a і b і, нарешті, вказуючи значення для всіх трьох параметрів:

```
# Include <iostream.h>

void show__parameters (int a = 1, int b = 2, int c = 3)

{
   cout << "a" << a << "b" << b << "з" << з << endl;
}

void main (void)

{
    show_parameters ();
  show_parameters (1001);
  show_parameters (1001, 2002);
  show_parameters (1001, 2002, 3003);
}
```

Коли ви відкомпілюєте і запустіть цю програму, на вашому екрані з'явиться наступний висновок:

```
C: \> DEFAULTS <ENTER>

a 1 b 2 з 3
```

a 1001 b 2 з 3

a 1001 b 2002 з 3

a 1001 b 2002 з 3003

Як бачите, якщо необхідно, функція використовує значення параметрів за замовчуванням.

Правила для пропуску значень параметрів:

Якщо програма опускає певний параметр для функції, що забезпечує значення за умовчанням, то слід опустити і всі наступні параметри. Іншими словами, ви не можете опускати середній параметр. У разі попередньої програми, якщо потрібно опустити значення параметра b в show_parameters, програма також мала опустити значення параметра c. Ви не можете вказати значення для a і c, опускаючи значення Ь.

Завдання значень за замовчуванням

Коли ви визначаєте функцію, C + + дозволяє вам вказати значення за замовчуванням для одного або декількох параметрів. Якщо програма в подальших викликах цієї функції опускає один або декілька параметрів, то функція буде використовувати для них значення за замовчуванням. Щоб привласнити параметру значення за замовчуванням, просто використовуйте оператор присвоювання всередині визначення функції.

Наприклад, наступна функція payroll вказує значення за замовчуванням для параметрів hours та rate:

```
float payroll (int employ_id, float hours = 40, float rate = 5.50)

{

  / / Оператори
}
```

Коли програма опускає один параметр, вона повинна опускати всі наступні параметри.

9. МОДУЛЬ 9 : Робота зі строками мови C++

9.1 Строки. Символи

Строкові літерали

Найпростіша строкою сутність (під строкової сутністю я розумію щось, з чим можна працювати як зі звичною рядком) в C - це так званий «рядковий літерал». Він являє собою послідовність символів, укладену в подвійні лапки. Приклад: "Здрастуй, світ!"

Основна властивість строкового літерала - простота його використання. Не маючи ні найменшого уявлення про те, чим він є насправді, ми можемо використовувати його практично скрізь, де від нас чекають рядок. Наприклад, в WinAPI-функцію SetWindowText (вона задає текст, пов'язаний з вікном) потрібно передати описувач вікна і рядок тексту. І ми можемо викликати її дуже просто: SetWindowText (hwnd, "Новий заголовок вікна");

Але звичайно, строкових літералів, природно, що фіксуються при створенні програми, буде замало.

Строкові змінні

В C відсутні вбудовані рядкові типи в тому сенсі, в якому вони є в мовах типу Basic і Pascal. І притаманна цим мовам легкість оперування рядковими змінними (привласнення, порівняння) в C недоступна. Що ж таке рядок в C?

Для початку розберемося, що таке рядок в життя. Очевидно, що рядок - це послідовність символів. В C - як у житті. C-рядок - це послідовність символів. Як відомо, послідовності в C представляються масивами або покажчиками. Між першими і другими існує зв'язок, однак природа цього зв'язку виходить за рамки цієї статті. Передбачається, що читачеві знайомі такі особливості зв'язку між масивами і покажчиками:

масив можна привести до покажчика на його перший елемент, що неявно відбувається при передачі масивів у функції, які очікують покажчики

інформація про розмір масиву переданого таким чином у функцію втрачається

в C не існує способу передати масив за значенням із збереженням його розміру

покажчика на перший елемент масиву достатньо для роботи з усім масивом, за умови що нам відома його довжина.

Якщо вищенаведені висловлювання викликають у вас труднощі, рекомендую спочатку розібратися з цим питанням, а вже потім читати цю статтю.

Надалі для простоти я здебільшого буду говорити про масивах, але майже все сказане стосується і вказівниками.

Отже, тип рядків в С - масив. Однак який тип елементів цього масиву? Взагалі кажучи, можливі варіанти. Історично символ займає 1 байт, в цьому випадку він має тип char. При цьому існують і інші кодування, в яких символ видається, наприклад, двома байтами. Для роботи з такими рядками потрібні спеціальні функції.

Злегка відвернемося від рядків і розберемо поняття кодування. За визначенням, кодування - це спосіб представлення чого або, в нашому випадку символів. Кодування символів діляться на однобайтні - кожен символ представлений одним байтом і багатобайтові, в яких одному символу відповідає кілька байтів. У свою чергу багатобайтові кодування можна розділити на кодування з фіксованою кількістю байтів - кожному символу відповідає однакова кількість байтів, і «плаваючі», в яких один символ може представлятися різною кількістю байтів в залежності від його вмісту. До перших відносяться кодування типу Unicode, в якій кожен символ представлений двома байтами, до других - UTF-8 і ін Плаваючі кодування - окрема тема, мови C / C + + не пропонують для них ніякої підтримки.

Необхідність в багатобайтові кодуваннях виникла через те, що одним стандартним байтом можна уявити не так багато символів, наприклад восьмібітних байт здатний приймати значення від 0 до 255, а значить в такий кодуванні не може існувати більше 256 різних символів. Враховуючи, що, наприклад, в японській мові близько двох тисяч ієрогліфів, 256 символів японцям явно не вистачить. Але вже двох восьмібітних байтів вистачить для подання 65536 символів - цілком непогано, хоча і недостатньо для подання всіх символів на світі.

Між однобайтнимі і фіксованими багатобайтові рядками принципової різниці немає. В C / C + + існує спеціальний тип для багатобайтові символів - wchar_t та спеціальні функції для роботи з рядками, що складаються з таких символів. Розмір wchar_t не фіксований в стандарті і визначається реалізацією компілятора. На багатьох платформах і компіляторах це два байти, відповідних кодуванні Unicode. Кожній функції, що працює з рядками з однобайтних символів, відповідає функція - «побратим», приймаюча рядки з багатобайтові символів. Крім того, існує спеціальна форма для запису строкових літералів, в яких символи представлені декількома байтами: перед лапками ставиться буква L. Тобто, повертаючись до нашого першого прикладу, виклик функції SetWindowText в Unicode-програмі буде виглядати так: SetWindowText (hwnd, L "Новий заголовок вікна");

Фактично, на цьому з точки зору програміста різниця між рядками з мультибайтних і однобайтових символів закінчується, тому надалі я буду розглядати тільки традиційні рядки з однобайтових символів. При необхідності ви легко знайдете багатобайтові аналоги всіх згаданих у статті функцій.

Символи

Символи самі по собі теж досить цікаві. Як ви вже знаєте, символьна змінна - це змінна типу char, що займає в пам'яті 1 байт. На відміну від рядків, символ - це вбудований інтегральний тип в C / C + +, для нього допустимі всі операції, допустимі для інтегральних типів. Існують символьні літерали, вони записуються в одинарних лапках (прямих апострофах). Приклад символьного літерала: char sym;

sym = 'A'; / / Символьний літерал. Його значення - код символу A (латинське) в використовуваної кодуванні

У наведеному вище прикладі значенням sym є 65 в кодової таблиці ASCII. У цьому випадку рядок sym = 'A' абсолютно еквівалентна рядку sym = 65. Проте, з метою поліпшення пристосованість краще завжди використовувати запис в апострофах - рано чи пізно програму може знадобитися скомпілювати на платформі, де у символу A інший код.

Для запису символьних літералів типу wchar_t використовується запис, аналогічна записи для строкових літералів цього типу: wchar_t sym;

sym = L'ab '; / / Символьний багатобайтові літерал. Кількість символів між апострофами залежить від розміру типу wchar_t

Існує спеціальний формат для запису символьних літералів - слеш, за яким йде код символу. Така форма запису необхідна, якщо ми хочемо використовувати елемент, не відображається в друкований символ, наприклад нуль-термінатор, який представляється так: '\ 0'. ПРИМІТКА

У фрагменті

char sym1;

char sym2;

char sym3;

sym1 = 0; / / (1)

sym2 = '\ 0'; / / (2)

sym3 = '\ 0x00'; / / (3)

рядка (1), (2) і (3) мають один і той же ефект. Однак друга і третя запис вважаються наочніше - ми відразу бачимо, що працюємо саме з символом.

Не плутайте '\ 0', 0 "та" 0 ". Перше - символьний літерал, відповідний символу з кодом 0. Друге - такий же літерал, але позначає цифру 0, її код в ASCII-кодуванні 48. Третій - це строковий літерал, що містить два символи, цифру 0 і нуль-термінатор.

Як я вже згадував тип char - інтегральний, а значить для нього визначені всі операції, визначені для інтегральних типів, у тому числі +, -, *, / і операції порівняння. Хоча не всі вони мають сенс для символів (наприклад, я погано собі уявляю сенс перемножування двох символів), проте їх використання зовсім «законно» і їх результат зазвичай саме такий, який очікувався.

Застосування ж цих операцій до рядків або взагалі заборонено, або, що ще гірше, їх результат в більшості випадків далекий від очікуваного.

9.2 Операція з строками

Операції з рядками

Створення рядків

Проілюструю створення рядків на фрагментах коду з комментаріямі.char str1 [10]; / / Рядок - масив з 10 символів. Початкове значення символів не визначено.

char str2 [10] = "Hello";

/ * Використовується ініціалізація (не присвоювання!). У перші 5 символів записується "Hello", в 6 - нуль-термінатор, значення трьох останніх не визначено. * /

char str3 [10] = {'H', 'e', 'l', 'l', 'o', '\ 0'}; / / Еквівалентно попередньому.

char str4 [10] = "Very long line";

/ / Помилка. Масив з 10 елементів не можна ініціювати довшою послідовністю.

char str5 [] = "Very long line";

/ * Компілятор автоматично визначає довжину масиву (в нашому випадку 15) і ініціалізує його послідовністю символів. * /

char * str6;

/ * Рядок - покажчик на символ. В більшості випадків для її використання потрібно виділити пам'ять. * /

str6 = (char *) malloc (10);

free (str6);

Присвоєння рядків

Перший і найочевидніший спосіб привласнення рядків - присвоювання окремих символів. Наприклад, str1 [0] = 'H';

str1 [1] = 'e';

str1 [2] = 'l';

str1 [3] = 'l';

str1 [4] = 'o';

str1 [5] = '\ 0';

Однак, це зовсім незручно. Не знаючи про правильні способи, починаючі програмісти часто «вигадують» свої способи привласнення рядків, звичайно, неправильні. Наведу кілька прикладів: char str1 [10], str2 [10];

str1 = "Hello";

str2 = str1;

/ / Одна і та ж помилка в обох операторах =.

/ / Ім'я масиву не можна використовувати в лівій частині оператора присвоювання.

Ця помилка щодо безпечна, оскільки призводить до збою на етапі компіляції. Є і набагато більш небезпечна помилка. char str1 [10] = "Hello";

char * str2;

str2 = str1;

str2 [1] = 'u';

Цей код відкомпілює, але, можливо, містить «ідеологічну» помилку. Неправильно вважати, що в str2 тепер міститься копія str1. Насправді цей покажчик вказує не на копію, а на ту ж саму рядок. При будь-яку зміну вмісту str2 змінюється str1. Однак, якщо саме це і потрібно, то все в порядку.

Ще один варіант присвоювання покажчиків - присвоювання їх строкових літералів. Як ви пам'ятаєте, тип строкового літерала - const char *, а значить такий код працює: const char * str;

str = "Hello";

Знову ж слід пам'ятати, що str вказує на строковий літерал, а не на його копію. Але, на жаль, такий код теж спрацює: char * str; / / Нет const

str = "Hello";

Тут ми маємо справу зі спадщиною C, в якому був відсутній const. Тому стандарт C + + дозволяє таке присвоєння. Що може мати неприємні наслідки: char * str; / / Нет const

str = "Hello";

str [1] = 'u';

Результат роботи такої програми непередбачуваний. Компілятор може розмістити константи в пам'яті тільки для читання, і спроба їх змінити призведе до збою. Тому завжди оголошуйте покажчики, в які ви збираєтеся записувати адреси строкових літералів як const char *. В цьому випадку компілятор не дозволить модифікувати дані і діагностує помилку, що допоможе вам виправити логіку программи.const char * str; / / const є

str = "Hello";

str [1] = 'u'; / / error: l-value specifies const object

Питання неправильного і ризикованого присвоювання рядків ми розглянули. Прийшла пора обговорити правильне присвоєння або копіювання рядків.

Для копіювання рядків існують декілька бібліотечних функцій, найбільш загальновживаної з яких є функція char * strcpy (char * dest, const char * src)

Функція посимвольно копіює вміст рядка, на яку вказує src в рядок, на яку вказує dest і повертає dest. Так як масив може бути перетворений в покажчик, такий виклик функції абсолютно легальний: char str1 [10], str2 [10];

```
strcpy (str1, "Hello");
strcpy (str2, str1);
```

При використанні цієї функції слід дотримуватися обережності. Небезпека полягає в тому, що навіть якщо вихідна рядок виявиться більше, ніж пам'ять, виділена для другого рядка (програмістом через malloc або компілятором при використанні масивів), функція strcpy ніяк про це дізнатися не зможе і продовжить копіювання в невиділений пам'ять. Зрозуміло, наслідки будуть катастрофічними.

Знизити ризик такого розвитку подій здатна функція char * strncpy (char * dest, const char * src, size_t count)

Останній параметр - максимальна кількість копійованих символів. Таким чином, передаючи туди розмір приймача, ви гарантуєте, що функція ніколи не вийде за межі виділеної пам'яті. Однак пам'ятайте, що якщо вихідна рядок буде скопійована не повністю, нуль-термінатор не з'явиться в результуючому рядку. Його доведеться записати самостійно.

9.3 Символьні строки

Символьні рядки зберігають таку інформацію, як імена файлів, назви книг, імена службовців та інші символьні поєднання. Більшість програм на C + + широко використовують символьні рядки. Далі ви дізнаєтеся, що в C + + символьні рядки зберігаються в масиві типу char, що закінчується символом NULL (або ASCII 0). В даному уроці символьні рядки розглядаються більш докладно. Ви дізнаєтеся, як зберігати й обробляти символьні рядки, а також як використовувати функції бібліотеки етапу виконання, які маніпулюють символьними рядками. До кінця цього уроку ви освоїте наступні основні концепції:

Щоб оголосити символьну рядок, ви повинні оголосити масив типу char,

Щоб привласнити символи символьної рядку, ваші програми просто привласнюють символи елементів масиву символьних рядків.

Програми C + + використовують символ NULL (ASCII 0), щоб відзначити останній символ рядка.

C + + дозволяє вашим програмам ініціалізувати символьні рядки при їх оголошенні.

Програми можуть передавати символьні рядки у функцію, як і будь масив.

Більшість бібліотек етапу виконання C + + забезпечують набір функцій, які керують символьними рядками.

Програми на C + + зберігають символьні рядки як масив типу char. Більшість програм широко використовують символьні рядки. Експериментуйте з кожною програмою, представленою в цьому уроці, щоб освоїтися з символьними рядками.

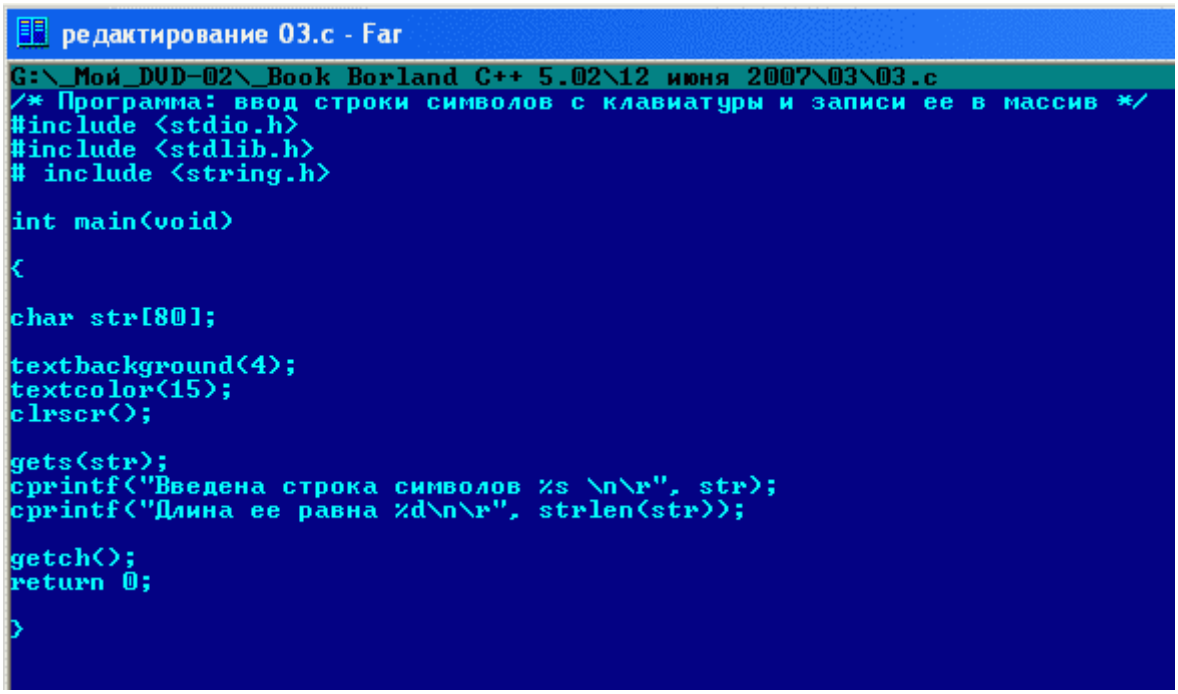
9.4 Введення/виведення строк

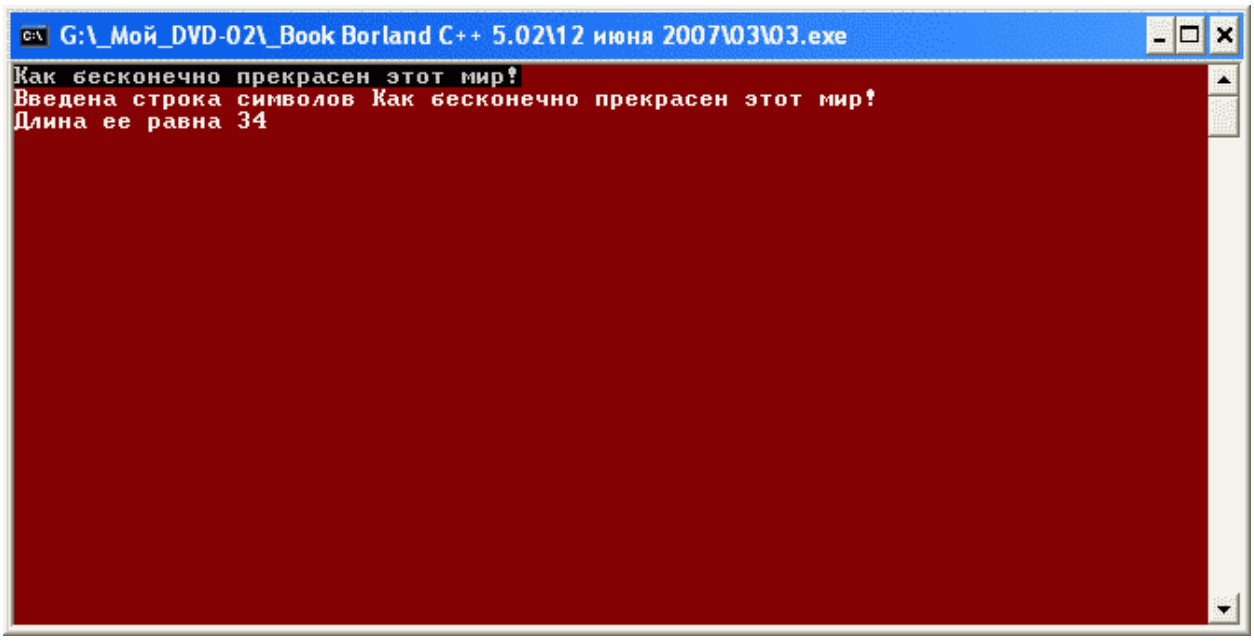
Функція gets () дозволяє читати рядок символів при введенні з клавіатури і записує її в пам'ять за адресою, на який вказує аргумент. Символи вводять до тих пір, поки не буде введений символ "повернення каретки" або, іншими словами, в кінці рядка повинна бути натиснута клавіша Enter. Символ "повернення каретки" не стане частиною рядка. Замість нього в кінець рядка буде введено символ кінця рядка - 0 ". Після цього відбудеться повернення з функції gets ().

Можна до натискання клавіші Enter виправляти неправильно введені символи. Для цього користуються клавішею backspace - повернення каретки на один позицію назад.

Прототип функції має вледующій вигляд: char * gets (char * str);

Тут * str - це покажчик на масив символів, в який записуються символи, що вводяться користувачем з клавіатури. Функція gets () повертає str. Нижче приведена програма, яка читає з клавіатури рядок символів і записує її в масив str - звичайну рядку Сі, що складається з 80 символів, тобто str [80]. Крім того, програма виводить на екран ще і довжину введеної з клавіатури рядка символів. Дивіться на малюнку результат роботи цієї програми.





Прототипи функцій такі:

int fputs (const char * str, FILE * покажчик файлу);
char * fgets (char * str, int довжина, FILE * увазатель файлу);

Функція fputs () поміщає (записує) в певний потік рядок, на яку вказує str. При наявності помилки ця функція повертає EOF.

Функція fgets () витягує (читає) з певного потоку рядок. Робиться це до тих пір, поки не буде прочитано символ нового рядка або кількість прочитаних (витягнутих) символів не стане рівним довжина - 1. При успішному завершенні роботи функція повертає str. У випадку помилки повертається порожній покажчик (null).

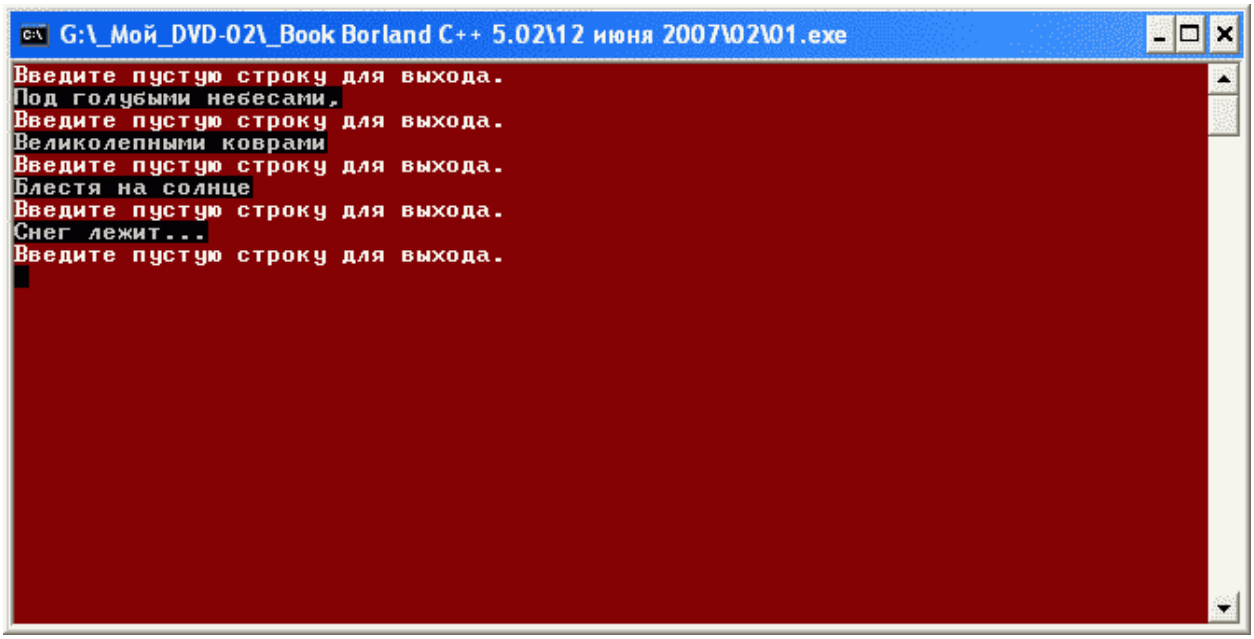
У наведеній програмі показано використання розглянутих функцій fputs () і fgets (). У програмі читається рядок склавіатури. Отримана рядок записується в файл data.txt. Завершується програма введенням порожнього рядка. Далі отриманий файл рядків закривається.

У програмі вставляється перед кожним рядком роздільник рядків, так як функція gets () не записує роздільник рядків. Це робиться для того, щоб файл легше було читати.

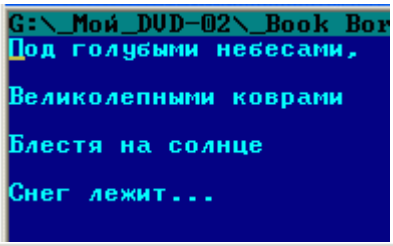
Програма представлена на малюнку нижче. На наступних малюнках показані результати роботи програми. Зокрема, показано роздруковане вміст записаного на диск файлу data.txt і довідка з тієї директорії, в яку записаний створений в програмі текстовий файл рядків data.txt.

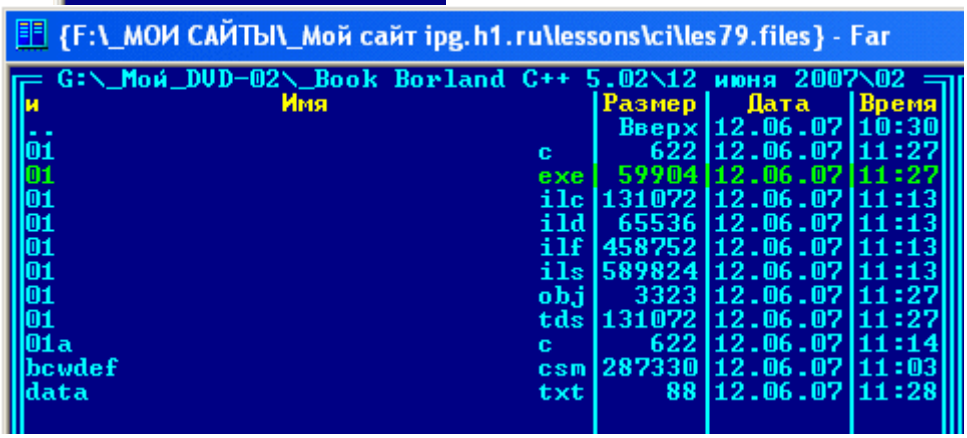
```
/* Программа: ввод строки с клавиатуры и записи ее в файл
#include <stdio.h>
#include <stdlib.h>
# include <string.h>

int main(void)
{
  FILE *fp;
  char str[80];

  textbackground(4);
  textcolor(15);
  clrscr();

  if((fp=fopen("data.txt", "w"))==NULL) {
    cprintf("Ошибка при открытии файла. \n\r");
    getch();
    exit(1);
}

  do {
    cprintf("Введите пустую строку для выхода. \n\r");
    gets(str);
    /* добавление разделителя строк */
    strcat(str, "\n\r");
    fputs(str, fp);
     }
    while(*str != ' ');
    fclose(fp);

    getch();
    return 0;
}
```

На цьому малюнку показаний процес введення декількох віршованих рядків з клавіатури.



На наступних двох малюнках Ви бачите текст, який знаходиться у файлі data.txt після введення тексту і запису його у файл і отриману в FAR довідку про файли, що знаходяться в директорії, з якою працювала програма.





9.5 Практична робота № 10: "Створення строки"

Рядки в с + + дозволяють нам працювати з символьними даними. Завдяки ним ми можемо читати з клавіатури текст, якось його обробляти і потім, наприклад, знову його виводити в консоль.

В С + + існує 2 типу рядків.

Перший з них - це масив змінних типу char.

Якщо хто не пам'ятає, то змінна типу char зберігає в собі 1 символ. Розмір такого рядка дорівнює розміру масиву - 1, тому що останній елемент містить NULL (порожній мінлива без значення), який позначає символ кінця рядка.

Наприклад:

```
char name [50];

cin >> name;

cout << "Hello" << name << endl;
```

Другий з варіантів, більш зручний - це спеціальний клас string

Для його роботи необхідно на початку програми підключити заголовний файл string:

```
# Include <string>
```

На відміну від типу char, string є класом. Більш докладно про класах я розповім пізніше, зараз вам достатньо знати, що класи містять в собі відразу кілька речей: змінні, константи та функції для роботи зі змінними. Це досить грубе пояснення, але на перший час вам вистачить.

Для створення рядка вам необхідно на початку програми написати using namespace std;

Тепер щоб створити рядок досить написати:

```
string s;
```

Для запису в рядок можна використовувати оператор =

```
s = "Hello";
```

Приклад роботи з класом string:

```
string name;

cout << "Enter your name" << endl;

cin >> name;

cout << "Hi" << s << "!" << endl;
```

Але поки ви скористалися лише однієї красою рядків: відсутністю необхідності ставити її розмір. Але крім цього існує безліч функцій для роботи з рядками.

s.append (str) - додає в кінець рядка рядок str. Можна писати як s.append (змінна), так і s.append ("рядок");

s.assign (str) - присвоює рядку s значення рядка str. Аналогічно записи s = str;

int i = s.begin () - записує в і індекс першого елемента рядки

int i = s.end () - аналогічно, але останнього

s.clear () - як випливає з назви, відчищає рядок. Тобто видаляє всі елементи в ній

s.compare (str)-порівнює рядок s з рядком str і повертає 0 в разі збіг (насправді порівнює коди символів і повертає з різницю)

s.copy (куди, скільки, починаючи з якого) - копіює з рядка s в куди (там може бути як рядок типу стринг, так і рядок типу char). Останні 2 параметра не обов'язкові (можна використовувати функцію з 1,2 або 3 параметрами)

bool b = s.empty () - якщо рядок порожній, повертає true, інакше false

s.erase (звідки, скільки) видаляє n елементів із заданою позиції

s.find (str, позиція) - шукає рядок str починаючи з заданої позиції

s.insert (позиція, str, починаючи, beg, count) - вставляє у рядок s починаючи з заданої позиції частина рядка str починаючи з позиції beg і вставляючи count символів

int len = s.length () - записує в len довжину рядка

s.push_back (symbol) - додає в кінець рядка символ

s.replace (index, n, str) - бере n перших символів з str і замінює символи рядка s на них, починаючи з позиції index

str = s.substr (n, m) - повертає m символів починаючи з позиції n

s.swap (str) змінює вміст s і str місцями.

s.size () - повертає число елементів в рядку.

Ось власне більшість необхідних функція для роботи з рядками в с + +. Набір досить непоганий, більше вам поки не знадобиться (Я особисто поки і цей набір не вивчив ;))

Тепер трохи прикладів і потім практика. Постарайтеся самі зрозуміти, що робить кожен приклад

```
string name, surname, text, fullname, s1, s2, s3, user;

user = "Petya Petrov";

cout << "Enter your name" << endl;

cin >> name;

cout << "Enter your surname" << endl;

cin >> surname;

fullname = name;
```

```
fullname + = ""; / / додаємо пробіл
fullname.append (surname);
if (fullname.compare (user) == 0) / / <=> if (! (fullname.compare (user)))
cout << "Your are good user" << endl;
else
cout << "Bad user" << endl;
cout << "enter s1" << endl;
cin >> s1;
cout << "enter s2" << endl;
cin >> s2;
s1.swap (s2);
cout <"new s1:" << s1 << endl << "new s2:" << s2 << endl;
cout << "Enter big text with your name" << endl;
cin >> text;
int i = 0;
i = text.find ("name");
while (i! = -1)
{
text.replace (i, name.length (), name);
s3 = text.substr (i, name.length ());
cout << "Replaced:" << s3 << endl;
i = text.find ("name");
}
```

```
cout << "New text:" << endl << text << endl;

text.clear ();

cout << "text:" << text << endl;
```

Ось власне невеликий шматок програми, що працює з рядками і незрозуміло що робить.

Ну а тепер, як завжди, трохи практики ;) (Хоча цілком можливо для новачків весь вищевикладений текст і так винос мозку і вони вже в висівок ;))

Завдання для рядків типу char:

Підрахувати кількість символів з рядку (рядок кінчається елементом 0: c = 0 if (c == 0) cout << "end" << endl;

Вам вводять рядок, потім вводять підрядок. Якщо підрядок є в введеної рядку вивести так, інакше немає

Вводять 3 рядки: а, б, с. Замінити в рядку з рядок а на рядок б

Завдання для рядків типу string:

Вводять текст і вводять підрядок, знайти всі входження підрядок в текст

Вводять текст і два слова, замінити всі слова 1 на слова 2 і всі слова 2 на слова 1

Вводять ім'я і текст, вивести всі входження імені в текст (ім'я + 5 символів до і 5 символів після нього)

Вводять 2 тексту. Порівняти їх, об'єднати, вивести всі прогалини, крапки, коми, двокрапки. Потім вивести розмір кожного тексту і загальний розмір. Потім поміняти всі букви а на a (росіяни на латинські) і до на k. Потім вивести кількість замін.

9.6 Функції обробки строк

Нижче наведені Функції рядків у мові Сі + +

```
char * strcpy (char * s1, const char * s2);
```

Копіює рядок s2 в масив символів s1. Повертає значення s1.

```
char * strncpy (char * s1, const char * s2, size_t n);
```

Копіює не більше n символів з рядка s2 в масив символів s1. Повертає значення s1.

```
char * strcat (char * s1, const char * s2);
```

Додає рядок s2 до рядка s1. Перший символ рядка s2 записується поверх завершального нульового символу рядка s1. Повертає значення s1.

char * strncat (char * s1, const char * s2, size_t n);

Додає не більше n символів рядка s2 в рядок s1. Перший символ рядка s2 записується поверх завершального нульового символу рядка s1. Возвращаетзначеніеs1.

int strcmp (const char * s1, const char * s2);

Порівнює рядки s1 і s2. Функція повертає 0, якщо рядки рівні; значення менше 0, якщо s1 менше s2 і значення більше 0, якщо s1 більше s2.

int strncmp (const char * s1, const char * s2, size_t n);

Порівнює до n символів рядків s1 і s2. Функція повертає 0, якщо рядки рівні; значення менше 0, якщо s1 менше s2 і значення більше 0, якщо s1 більше s2.

char * strtok (char * s1, const char * s2);

Послідовність викликів strtok розбиває рядок s1 на лексеми - логічні частини, такі як слова, розділені символами, що містяться в рядку s2. Перший виклик містить в якості першого аргументу s1, а наступні виклики для тієї ж рядки, містять в якості першого аргументу null. При кожному виклику повертається вказівник на поточну лексему. Якщо лексем більше немає повертається null.

size_t strlen (const char * s);

Визначає довжину рядка s. Повертає кількість символів, що передують завершального нульового символу.

char * strchr (const char * s, int c);

Знаходить позицію першого входження символу c в рядок s. Якщо c знайдено, функція повертає вказівник на c в рядку s, інакше повертається NULL.

size_t strcspn (const char * s1, const char * s2);

Визначає і повертає довжину початкового сегмента рядка s1, який містить тільки ті символи, які не входять в s2.

char * strpbrk (const char * s1, const char * s2);

Знаходить в рядку s1 позицію першого входження будь-якого із символів рядка s2. Якщо символ з рядка знайдено, повертається вказівник на цей символів рядку s1, інакше повертається NULL.

char * strrchr (const char * s, int c);

Знаходить позицію останнього входження символу c в рядок s. Якщо c знайдено, функція повертає вказівник на цей символ, інакше повертається NULL.

char * strstr (const char * s1, const char * s2);

Знаходить позицію першого входження рядка s2 в рядок s1. Якщо підрядок знайдено, функція повертає вказівник підрядка в рядку s1, інакше повертається NULL.

10. МОДУЛЬ 10: Файлові операції введення/виведення мови C++

10.1 Виведення у файловий потік

Запис у файл за допомогою fwrite

fwrite визначається як
int fwrite (const char * array, size_t size, size_t count, FILE * stream)
Функція fwrite записує блок даних у потік. Таким чином запишеться масив елементів array в поточну позицію в потоці. Для кожного елемента запишеться size байт. Індикатор позиції в потоці зміниться на число байт, записаних успішно. Повертане значення дорівнюватиме count в разі успішного завершення запису. У випадку помилки повертається значення буде менше count.

Наступна програма відкриває файл прімер.txt, записує в нього рядок символів, а потім його закриває.

```
# Include <stdio.h>
# Include <string.h>
# Include <stdlib.h>

int main (void)
{
   FILE * fp;
   size_t count;
   char const * str = "привіт \ n";

   fp = fopen ("прімер.txt", "wb");
   if (fp == NULL) {
      perror ("помилка відкриття прімер.txt");
      return EXIT_FAILURE;
   }
   count = fwrite (str, 1, strlen (str), fp);
     printf ("Записано% lu байт. fclose (fp)% s. \ n", (unsigned long) count, fclose (fp) == 0? "успішно": "з помилкою");

   return 0;
}
```

[Правити]

Запис в потік за допомогою fputc

Функція fputc застосовується для запису символу в потік.

int fputc (int c, FILE * fp);

Параметр с «тихо» конвертується в unsigned char перед висновком. Якщо пройшло успішно, то fputc повертає записаний символ. Якщо помилка, то fputc повертає EOF.

Стандартний макрос putc також визначено в <stdio.h>, працюючи в загальному випадку аналогічно fputc, за винятком того моменту, що будучи макросом, він може обробляти свої аргументи більше одного разу.

Стандартна функція putchar, також визначена у <stdio.h>, приймає тільки перший аргумент, і є еквівалентною putc (c, stdout), де с є згаданим аргументом.

Приклад використання

Нижченаведена програма на мові Сі відкриває двійковий файл з назвою мойфайл, читає п'ять байт з нього, а потім закриває файл.

```
# include <stdio.h>
# include <stdlib.h>

int main (void)
{
  char buffer [5] = {0}; / * инициализируем нулями * /
  int i, rc;
  FILE * fp = fopen ("мойфайл", "rb");
  if (fp == NULL) {
    perror ("Помилка при відкритті \" мойфайл \ "");
    return EXIT_FAILURE;
  }
  for (i = 0; (rc = getc (fp))! = EOF && i <5; buffer [i + +] = rc)
    ;
  fclose (fp);
  if (i == 5) {
    puts ("Прочитані байти ...");
    printf ("% x% x% x% x% x \ n", buffer [0], buffer [1], buffer [2], buffer [3],
buffer [4]);
  } else
    fputs ("Помилка читання файлу. \ n", stderr);
  return EXIT_SUCCESS;
}
```

10.2 Читання із вхідного файлового потоку

Тільки що ви дізналися, що, використовуючи клас ofstream, ваші програми можуть швидко виконати операції виводу у файл. Подібним чином

ваші програми можуть виконати операції введення з файлу, використовуючи об'єкти типу ifstream. Знову ж таки, ви просто створюєте об'єкт, передаючи йому як параметра потрібне ім'я файлу:

```
ifstream input_file ("filename.EXT");
```

Наступна програма FILE_IN.CPP відкриває файл BOOKINFO.DAT, який ви створили за допомогою попередньої програми, і читає, а потім відображає перші три елемента файла:

```
# include <iostream.h>

# include <fstream.h>

void main (void)

{
  ifstream input_file ("BOOKINFO.DAT");
  char one [64], two [64], three [64];
  input_file >> one;
  input_file >> two;
  input_file >> three;
  cout << one << endl;
  cout << two << endl;
  cout << three << endl;
}
```

Якщо ви відкомпілюєте і запустіть цю програму, то, ймовірно, припустити, що вона відобразить перші три рядки файлу. Однак, подібно cin, вхідні файлові потоки використовують порожні символи, щоб визначити, де закінчується одне значення і починається інше. В результаті при запуску попередньої програми на дисплеї з'явиться наступний висновок:

```
C: \> FILE_IN <ENTER>

вчимося

програмувати

на
```

10.3 Читання цілої строки файлового введення

Ви дізналися, що ваші програми можуть використовувати cin.getline для читання цілої рядки з клавіатури. Подібним чином об'єкти типу ifstream

можуть використовувати getline для читання рядки файлового введення. Наступна програма FILELINE.CPP використовує функцію getline для читання всіх трьох рядків файлу BOOKINFO.DAT:

```
# include <iostream.h>

# include <fstream.h>

void main (void)

{
  ifstream input_file ("BOOKINFO.DAT");
  char one [64], two [64], three [64];
  input_file.getline (one, sizeof (one));
  input_file.get line (two, sizeof (two));
  input_file.getline (three, sizeof (three));
  cout << one << endl;
  cout << two << endl;
  cout << three << endl;
}
```

В даному випадку програма успішно читає вміст файлу, тому що вона знає, що файл містить три рядки. Однак у багатьох випадках ваша програма не буде знати, скільки рядків міститься у файлі. У таких випадках ваші програми будуть просто продовжувати читання вмісту файлу поки не зустрінуть кінець файлу.

10.4 Визначення кінця файлу

Звичайною файлової операцією в ваших програмах є читання вмісту файлу, поки не зустрінеться кінець файлу. Щоб визначити кінець файлу, ваші програми можуть використовувати функцію eof потокового об'єкта. Ця функція повертає значення 0, якщо кінець файлу ще не зустрівся, і 1, якщо зустрівся кінець файлу. Використовуючи цикл while, ваші програми можуть безперервно читати вміст файлу, поки не знайдуть кінець файлу, як показано нижче:

```
while (! input_file.eof ())

{
  / / Оператори
}
```

В даному випадку програма буде продовжувати виконувати цикл, поки функція eof повертає брехня (0). Наступна програма TEST_EOF.CPP

використовує функцію eof для читання вмісту файлу BOOKINFO.DAT, поки не досягне кінця файла:

```
# Include <iostream.h>

# Include <fstream.h>

void main (void)

{
  ifstream input_file ("BOOKINFO.DAT");
  char line [64];
  while (! input_file.eof ())

  {
    input_file.getline (line, sizeof (line));
    cout << line << endl;
  }
}
```

Аналогічно, наступна програма WORD_EOF.CPP читає вміст файлу по одному слову за один раз, поки не зустрінеться кінець файлу:

```
# Include <iostream.h>

# Include <fstream.h>

void main (void)

{
  ifstream input_file ("BOOKINFO.DAT");
  char word [64];
  while (! input_file.eof ())

  {
    input_file >> word;
    cout << word << endl;
  }
}
```

І нарешті, наступна програма CHAR_EOF.CPP читає вміст файлу по одному символу за один раз, використовуючи функцію get, поки не зустріне кінець файлу:

```
# Include <iostream.h>

# Include <fstream.h>
```

```
void main (void)

{
  ifstream input_file ("BOOKINFO.DAT");
  char letter;
  while (! input_file.eof ())

  {
    letter = input_file.get ();
    cout << letter;
  }
}
```

10.5 Перевірка помилок при виконанні файлових операцій

Програми, представлені до теперішнього моменту, припускали, що під час файлових операцій В / В не відбуваються помилки. На жаль, це збувається не завжди. Наприклад, якщо ви відкриваєте файл для введення, ваші програми повинні перевірити, що файл існує. Аналогічно, якщо ваша програма пише дані в файл, вам необхідно переконатися, що операція пройшла успішно (наприклад, відсутність місця на диску, швидше за все, перешкодить записи даних). Щоб допомогти вашим програмам стежити за помилками, ви можете використовувати функцію fail файлового об'єкта. Якщо в процесі файлової операції помилок не було, функція поверне брехня (0). Однак, якщо зустрілася помилка, функція fail поверне істину. Наприклад, якщо програма відкриває файл, їй слід використовувати функцію fail, щоб визначити, чи відбулася помилка, як це показано нижче:

```
ifstream input_file ("FILENAME.DAT");
if (input_file.fail ())

{
  cerr << "Помилка відкриття FILENAME.EXT" << endl;
  exit (1);
}
```

Таким чином, програми повинні переконатися, що операції читання і записи пройшли успішно. Наступна програма TEST_ALL.CPP використовує функцію fail для перевірки різних помилкових ситуацій:

```
# Include <iostream.h>

# Include <fstream.h>
```

```
void main (void)

{
  char line [256];
  ifstream input_file ("BOOKINFO.DAT");
   if (input_file.fail ()) cerr << "Помилка відкриття BOOKINFO.DAT" <<
endl;
  else

  {
    while ((! input_file.eof ()) && (! input_file.fail ()))

    {
      input_file.getline (line, sizeof (line));
     if (! input_file.fail ()) cout << line << endl;
    }
  }
}
```

10.6 Закриття файлу. Управління файлом

Закриття файлу

При завершенні вашої програми операційна система закриє відкриті нею файли. Однак, як правило, якщо вашій програмі файл більше не потрібний, вона повинна його закрити. Для закриття файлу ваша програма повинна використовувати функцію close, як показано нижче:

```
input_file.close ();
```

Коли ви закриваєте файл, всі дані, які ваша програма писала в цей файл, скидаються на диск, і оновлюється запис каталогу для цього файлу.

Управління файлом

У прикладах програм, представлених в даному уроці, файлові операції введення і виведення виконувалися з початку файлу. Однак, коли ви записуєте дані в вихідний файл, можливо, ви захочете, щоб програма додавала інформацію в кінець існуючого файлу. Для відкриття файлу в режимі додавання ви повинні при його відкритті вказати другий параметр, як показано нижче:

```
ifstream output_file ("FILENAME.EXT", ios :: app);
```

В даному випадку параметр ios :: app вказує режим відкриття файлу. В міру ускладнення ваших програм вони будуть використовувати поєднання значень для режиму відкриття файлу.

ios :: app-ідкриває файл в режимі додавання, розташовуючи файловий покажчик в кінці файлу;

ios :: ate-має файловий покажчик в кінці файлу;

ios :: in-вказує відкрити файл для введення;

Ios ::nocreate-якщо зазначений файл не існує, не створювати файл і повернути помилку;

ios :: noreplace-якщо файл існує, операція відкриття повинна бути перервана і повинна повернути помилку;

ios :: out-вказує відкрити файл для виводу;

ios :: trunc-скидає (перезаписує) утримуємо, з існуючого файлу.

Наступна операція відкриття файлу відкриває файл для висновку, використовуючи режим ios :: noreplace, щоб запобігти перезапис існуючого файлу:

```
ifstream output_file ("FIlename.EXT", ios :: out | ios :: noreplace);
```

10.7 Виконання операцій читання та запису

Всі програми, виконували файлові операції над символьними рядками. В міру ускладнення ваших програм, можливо, вам знадобиться читати і писати масиви та структури. Для цього ваші програми можуть використовувати функції read і write. При використанні функцій read і write ви повинні вказати буфер даних, в який дані будуть читатися чи з якого вони будуть записуватися, а також довжину буфера в байтах, як показано нижче:

```
input_file.read (buffer, sizeof (buffer));
output_file.write (buffer, sizeof (buffer));
```

Наприклад, наступна програма STRU_OUT.CPP використовує функцію write для виведення вмісту структури в файл EMPLOYEE.DAT:

```
# Include <iostream.h>

# Include <fstream.h>

void main (void)

{
  struct employee
```

```
  {
    char name [64];
    int age;
    float salary;
  } Worker = {"Джон Дой", 33, 25000.0};

  ofstream emp_file ("EMPLOYEE.DAT");
  emp_file.write ((char *) & worker, sizeof (employee));
}
```

Функція write зазвичай отримує покажчик на символьний рядок. Символи (char *) є оператор приведення типів, який інформує компілятор, що ви передаєте покажчик на інший тип. Подібним чином наступна програма STRU_IN.CPP використовує метод read для читання з файлу інформації про службовця:

```
# Include <iostream.h>

# Include <fstream.h>

void main (void)

{
  struct employee

  {
    char name [6 4];
    int age;
    float salary;
  } Worker = {"Джон Дой", 33, 25000.0};

  ifstream emp_file ("EMPLOYEE.DAT");
  emp_file.read ((char *) & worker, sizeof (employee));
  cout << worker.name << endl;
  cout << worker.age << endl;
  cout << worker.salary << endl;
}
```

10.8 Строкові потоки

Потік може бути прив'язаний до файлу, тобто масиву символів, що зберігається не в основній пам'яті, а, наприклад, на диску. Точно так же потік можна прив'язати до масиву символів в основній пам'яті.

Наприклад, можна скористатися вихідним рядковим потоком ostrstream для форматування повідомлень, що не підлягають негайній друку:

```
char * p = new char [message_size];
ostrstream ost (p, message_size);
do_something (arguments, ost);
display (p);
```

За допомогою стандартних операцій виведення функція do_something може писати в потік ost, передавати ost підпорядковується їй функцій і т.п. Контроль переповнення не потрібен, оскільки ost знає свій розмір і при заповненні перейде в стан, обумовлений fail (). Потім функція display може послати повідомлення в "справжній" вихідний потік. Такий прийом найбільш підходить в тих випадках, коли остаточна операція виведення призначена для запису на більш складний пристрій, ніж традиційне, орієнтоване на послідовність рядків, вивідний пристрій. Наприклад, текст з ost може бути поміщений в фіксовану область на екрані. Аналогічно, istrstream є вступним рядковим потоком, що читає з послідовності символів, що закінчується нулем:

```
void word_per_line (char v [], int sz)
  / *
друкувати "v" розміром "sz" по одному слову в рядку
  * /
  {
    istrstream ist (v, sz); / / створити istream для v
    char b2 [MAX]; / / довше найдовшого слова
    while (ist >> b2) cout << b2 << "\ n";
   }
```

Завершальний нуль вважається кінцем файлу. Строкові потоки описані у файлі <strstream.h>.

10.9 Організація створення вхідніх строкових потоків

Вхідні рядкові потоки створюються за допомогою такого конструктора класу istrstream:

```
istrstream імя_потока (char * str);
```

Обов'язковою параметром конструктора об'єктів класу istrstream є покажчик str на вже існуючий ділянку основної пам'яті.

Наприклад, такі оператори

```
char buf [40];
```

istrstream inBuf (buf);

визначать вхідний рядковий потік з ім'ям inBuf і зв'яжуть його з ділянкою пам'яті, попередньо виділених для символьного масиву buf []. Тепер цей строковий потік inBuf може використовуватися як лівий операнд операції витягання >>.

Приклад 1. У наступній програмі визначена рядок, що містить символьну інформацію "123.5 Salve!", Потім визначений і пов'язаний з цим рядком вхідний рядковий потік instr. З потоку instr і тим самим з рядка, що адресується вказівником stroka, розділені пробілами значення переносяться в змінну real типу double і символьний масив array [10]. Текст програми:

```
/ / OOP13_1.CPP - рядкові потоки, операція витягання >>.
/ / Автоматично включається файл iostream.h:
# Include <strstrea.h>
void main ()
{
 / / Виділена область пам'яті (рядок):
 char * stroka = "123.5 Salve!";
 / / Створений вхідний рядковий потік instr:
 istrstream instr (stroka);
 char array [10];
 double real;
 / / Витягаємо інформацію з строкового потоку:
 instr >> real >> array;
 / / Вивід на екран:
 cout << "\ narray =" << array << "real =" << real << endl;
}
```

Текст цієї програми можна взяти тут.

Результат виконання програми:

array = Salve! real = 123.5

Приклад 2. У наступній програмі за допомогою вхідного строкового потоку виконується читання інформації, що передається як параметр командного рядка функції main:

```
/ / OOP13_2.CPP - вхідний рядковий потік; читання аргументу функції main ().
/ / Автоматично включається файл iostream.h:
# Include <strstrea.h>
void main (int argc, char * argv []) / / Визначено аргументи.
{
 char name [40]; / / Виділяється область пам'яті.
 / / Створює рядковий потік input:
 istrstream input (argv [1]);
```

```
 / / Витягаємо інформацію з строкового потоку:
 input >> name;
 / / Вивід в стандартний потік (на екран):
 cout << "\ nПрі виклику аргумент =" << name << endl;
}
```

Текст цієї програми можна взяти тут.

Якщо програма буде запущена на виконання командою:

C: \> OOP13_2.БХЕ Filename <Enter>

то на екрані з'явиться повідомлення:

При виклику аргумент = FileName

Витяг інформації з строкового потоку за допомогою операції >> виконується, починаючи від першого непробельний символ до найближчого пробіл. Якщо необхідно читати і пробільні символи, то можна скористатися функціями бесформатного обміну get () і getline (). Ці функції, що викликаються для символьних потоків, дозволяють організувати, наприклад, копіювання рядків. Для копіювання рядків в бібліотеці мови С існує спеціальна функція strcpy (), визначена в заголовному файлі string.h. Копіювання нескладно організувати і за допомогою циклічного перенесення символів з одного масиву в інший. Проте цікавий варіант копіювання виходить при використанні строкових потоків. Наприклад, в наступній програмі виконується "копіювання" вмісту рядка, що адресується вказівником line, в заздалегідь створений символьний масив array []. Відповідно до формату функції getline () її перший параметр - масив, в який ведеться читання, другий параметр - гранична кількість читаних символів, третій параметр - обмежує символ, після вилучення якого обмін припиняється. Функція getline () викликається для читання з потоку inpotok, пов'язаного з рядком line. Текст програми:

```
/ / OOP13_3.CPP - копіювання рядка функцією getline ().
# Include <strstrea.h>
void main ()
{
 char * line = "000 111 \ t222 \ n333 \ t444 555";
 istrstream inpotok (line);
 char array [80];
 inpotok.getline (array, sizeof (array), '\ 0');
 cout << "\ narray =" << array << endl;
}
```

Текст цієї програми можна взяти тут.

Результат виконання програми:

array = 000111222

333 444 555

У результатах виконання зверніть увагу на вплив символів '\ t', '\ n', присутніх як у вихідній рядку line, так і перенесених в символьний масив array. Вони, природно, не виводяться на екран, а забезпечують табуляцію і зміну рядка.

10.10 Створення, використання вихідних строкових потоків

Конструктор класу istrstream дозволяє створювати і пов'язувати із заданою областю пам'яті безіменні вхідні рядкові потоки. Наприклад, читання інформації, переданої при запуску функції main () в командному рядку, забезпечить наступна програма:

```
/ / OOP14_1.CPP - безіменний вхідний рядковий потік; читання
/ / Даних за допомогою операції витягання >>.
# Include <strstrea.h>
void main (int Narg, char * arg [])
{
 char path [80];
 / / Читання з безіменного потоку:
 istrstream (arg [0]) >> path;
 cout << "\ nПолное ім'я програми: \ n" << path << endl;
}
```

Результат виконання програми, наприклад, такий:

Повне ім'я програми: D: \ WWP \ TESTPROG \ OOP14_1.EXE

За угодами мови та операційної системи arg [0] завжди містить повне найменування програми із зазначенням диска і повного шляху до каталогу, де знаходиться EXE-файл. Виклик конструктора istrstream (arg [0]) створює безіменний об'єкт - вхідний рядковий потік, пов'язаний з символьним масивом, адреса якого передається як значення покажчика arg [0]. До цього потоку застосовна операція витягання >>, за допомогою якої повне найменування програми переноситься у вигляді рядка з arg [0] в символьний масив path [].

У наступній програмі з одним і тим же ділянкою пам'яті (рядок, адресується вказівником line) послідовно пов'язують три безіменних вхідних строкових потоку.

```
/ / OOP14_2.CPP - читання з безіменних строкових потоків.
# Include <strstrea.h>
void main ()
{
 char * line = "000 111 \ t222 \ n333 \ t444 555";
 char array [80]; / / Допоміжний масив /
 / / Читання з безіменного потоку до пробілу:
 istrstream (line) >> array;
 cout << "\ narray =" << array << endl;
 / / Повторне читання з безіменного потоку:
```

```
  istrstream (line) >> array;
  cout << "\ narray =" << array << endl;
  / / Виклик функції getline () для безіменного потоку:
  istrstream (line). getline (array, sizeof (array), '\ 0');
  cout << "array =" << array << endl;
}
```

Текст цієї програми можна взяти тут.

Результат виконання програми:

```
array = 000

array = 000
array = 000111222
333 444 555
```

З першого безіменного потоку дані в масив array витягуються (до першого пробілу) операцією >>. Другий безіменний потік пов'язаний з тією ж рядком. З нього в масив array знову з початку рядка читається та ж сама інформація. Третій безіменний потік читається за допомогою функції getline (), яка повністю копіює рядок line в символьний масив array []. Витяг даних функцією getline () триває до появи нульового ознаки кінця рядка '\ 0'. Цей символ також переноситься в масив array і служить ознакою кінця створеної рядка.

10.11 Основи потоків введення виведення

Бібліотека потоків C + + надає набір класів для управління введенням-висновком. Безпосередньо в мову C + + (як і в мову Сі) засоби введення-виведення не входять. У програмах часто використовується препроцесорну директива

```
# Include <iostream.h>
```

Призначення зазначеного в директиві заголовки - зв'язати модульна програми з однією з основних частин бібліотеки введення-виведення, побудованої на основі механізму класів. Ця бібліотека майже стандартна, так як включена практично у все компілятори C + +. Однак про стандарт тут можна говорити тільки неформально: бібліотека створена після появи мови. Вона розроблялася в деякому сенсі незалежно від створення мови, не входить в формальний опис мови і написана на мові C + +.

Потоки введення-виведення

Бібліотека забезпечує програміста механізмами для отримання даних з потоків і для приміщення даних в потоки. Кожен потік (за винятком

рядкових) асоціюється за допомогою операційної системи з певним зовнішнім пристроєм. При обміні з потоком використовується допоміжний ділянку пам'яті, званий буфером потоку. При введенні даних вони спочатку поміщаються в буфер і тільки потім передаються виконуваної програмі. При виведенні - дані заповнюють буфер перед передачею зовнішнього пристрою. Заповнення та очищення буферів операційна система виконує без явного участі програміста, тому потік в прикладній програмі можна розглядати як послідовність байтів, не залежну від конкретних пристроїв, з якими ведеться обмін даними.

Всі потоки бібліотеки послідовні, тобто в кожен момент для потоку визначені позиції записи і (або) читання, які після обміну переміщуються по потоку на довжину переданої порції даних.

В залежності від реалізованого напрямку передачі даних потоки ділять на три групи:

· Вхідні, з яких читається інформація;

· Вихідні, в які вводяться дані;

· Двонаправлені, що допускають як читання, так і запис.

Відповідно до особливостей обслуговується пристрою потоки прийнято ділити на наступні групи:

· Стандартні, для передачі даних від клавіатури і до дисплея (у всіх попередніх програмах використовувалися потоки даної групи);

· Файлові, при розміщенні інформації на зовнішніх носіях (наприклад, диск, магнітна стрічка);

· Рядкові, що дозволяють розміщувати дані потоку в пам'яті (символьний масив або рядок) і користуватися при цьому всіма засобами, наданими бібліотекою потоків (наприклад, форматний введення-виведення даних).

Класи потоків C + +

Як сказано вище, бібліотека вводу-виводу розроблена засобами об'єктно-орієнтованого програмування і є бібліотекою класів. Основні переваги такого підходу полягають в наступному.

По-перше, класам потоків доступний надійний механізм контролю типів переданих даних, заснований на перевантаженні операцій (що забезпечує для кожного типу даних виклик відповідної функції обробки). Таким чином, саме класами інтерпретуються «сирі» послідовності байтів відповідних потоків.

По-друге, класи потоків, завдяки поліморфізму, дозволяють одним і тим же функцій працювати з потоками різних типів, а бібліотеці в цілому - підтримувати однаковий інтерфейс вводу-виводу. Наприклад, інтерфейс, використовуваний стандартним вводом-висновком, застосуємо також до файлових і строковим потокам.

По-третє, бібліотека вводу-виводу розширювана. Це означає, що, з одного боку, класи бібліотеки можуть працювати з новими типами даних, визначеними програмістом, використовуючи згаданий механізм перевантаження операцій, а з іншого - програміст має можливість

створювати власні класи потоків на основі успадкування властивостей класів бібліотеки.

Тим не менш, для прикладного програміста зазвичай достатньо лише відомостей про показаних на малюнку основних класах:

ios - базовий потоковий клас;

strstreambase - базовий клас строкових потоків;

fstreambase - базовий клас файлових потоків;

istream - клас вхідних потоків;

ostream - клас вихідних потоків;

iostream - клас двонаправлених потоків вводу-виводу;

ifstream - клас вхідних файлових потоків;

ofstream - клас вихідних файлових потоків;

fstream - клас двонаправлених файлових потоків;

istrstream - клас вхідних строкових потоків;

ostrstream - клас вихідних строкових потоків;

strstream - клас двонаправлених строкових потоків.

Клас ios є базовим для всіх класів потоків і, таким чином, містить всі загальні засоби потоків. Наприклад, за допомогою методів і даних класу ios здійснюється управління процесом передачі даних з буфера і в буфер. При виконанні цих дій необхідні, наприклад, відомості про потрібний підставі числення, про точність подання дійсних чисел і т.д. Клас ios містить ці відомості, тобто дані, що відносяться до станів потоків, і методи, що дозволяють змінювати ці властивості.

Буферизація даних потоків реалізована за допомогою наступного механізму: клас ios, а отже, і похідні класи містять покажчик на об'єкт класу streambuf бібліотеки - базового класу, що забезпечує свої похідні класи загальними методами для буферизації даних. У свою чергу його похідні класи (filebuf і strstreambuf) забезпечують взаємодію створюваних потоків з фізичними пристроями. Дана паралельна ієрархічна структура класів-буферів не показана на малюнку через досить рідкісною необхідності звернення до методів і даних класу streambuf безпосередньо з прикладних програм.

Натомість частіше відбувається опосередковане їх використання викликом методів класів - спадкоємців ios.

Класи потоків, їх дані і методи стають видимими і доступними в програмі, якщо в неї включений відповідний заголовний файл:

iostream.h-для класів ios, istream, ostream, stream;

fstream.h-для класів fstreambase, ifstream, ofstream, fstream;

strstrea.h - для класів strstreambase, istrstream, ostrstream, strstream.

Оскільки клас ios є базовим для інших потокових класів, то включення в текст програми будь-якого з заголовних файлів fstream.h або strstrea.h автоматично підключає до програми і файл iostream.h. Відповідні перевірки виконуються на етапі препроцесорну обробки.

10.12 Файлові потоки

Включається файл:
Include <fstream>

Файлові потоки реалізовані наступними класами:

/ / Файлові потоки введення
typedef basic_ifstream <char> ifstream;
typedef basic_ifstream <wchar_t> wifstream;

/ / Файлові потоки виводу
typedef basic_ofstream <char> ofstream;
typedef basic_ofstream <wchar_t> wofstream;

/ / Файлові потоки введення / виводу
/ / Спадкоємці від попередніх
typedef basic_fstream <char> fstream;
typedef basic_fstream <wchar_t> wfstream;
методи файлових потоків
close () - закриває файл;
open (const char * s, ios_base :: openmode n, long protection = 0666) - відкриває файл;
bool is_open () - визначає чи відкритий файл.

Конструктор файлового потоку може замінити метод open. Значення режиму відкриття файлу задаються вкладеним типом open_mode:

```
typedef int openmode; / / режим відкриття потоку
enum open_mode {
 app = 0x01, / / завжди писати в кінець файлу
 binary = 0x02, / / введення / виведення в двійковому режимі
 in = 0x04, / / відкрити для введення
 out = 0x08, / / відкрити для виведення
 trunc = 0x10, / / знищити дані після відкриття існуючого потоку
 ate = 0x20 / / відкрити позиціонуючи покажчик на кінець файлу
};
```

текстової та двійковий режим

В двійковому режимі при операціях введення / виводу особливі символи не замінюються (типу кінця рядка), що дозволяє правильно обробити двійкові файли (наприклад, файли *. Bmp). При цьому для читання даних використовується метод read, а для запису метод write.

За замовчуванням потоки відкриваються в текстовому режимі і в залежності від типу потоку для введення, виведення або введення / виводу. Так у виклику ofstream fout ("out.txt", ios_base :: out | ios_base :: app); ios_base :: out | можна опустити.

приклад

```
/ / Читання елемента з поточної позиції потоку
template <class T>
inline void read_el (ifstream & fin, T el)
{Fin.read ((char *) & el, sizeof (el));}

/ / Запис елемента в поточну позицію потоку
template <class T>
inline void write_el (ofstream & fout, T el)
{Fout.write ((const char *) & el), sizeof (el));}

/ / ============================================
/ **************************
Вводимо ім'я файлу, потім
виводимо його розмір.
Для завершення необхідно ввести
некоректне ім'я файлу.
************************** /
# Include <iostream>
# Include <fstream>
using namespace std;
int main () {
char buf [100];
ifstream fin;
cout << "for close program input incorrect file name \ \ n";
```

```
      while (1) {
        cout << "input file name:";
        cin >> buf;
        fin.open (buf, ios_base :: binary);
if(!fin.is_open()) {
    cout<<"file not exist";
    break;
    }
  fin.seekg(0,ios_base::end);
  cout<<"file size is "<<fin.tellg()
      <<" bytes"<<endl;
  fin.close();
}
return 0;
}
```

11. МОДУЛЬ 11: Динамічні структури даних

11.1 Списки . Види списків

Списки

Поняття "списки" включає в себе самі різні структури даних. Це можуть бути і масиви, в тому числі масиви записів, і спеціальні динамічні структури даних, і навіть дерева. Спільним для них є те, що вони містять набір записів одного виду, обмежений за розміром або необмежений, упорядкований або неупорядкований. Дані, що зберігаються в цих записах, звичайно логічно пов'язані між собою, наприклад, прізвища студентів однієї групи і т.п. У тексті програми такий зв'язок може виражатися в тому, що всі такі елементи зберігаються в одному і тому ж масиві як безперервному блоці пам'яті. Але крім загального імені масиву (і адреси його початку) між цими елементами ніякої іншої фізичної зв'язку немає, і на фізичному рівні подібні списки можуть бути названі "незв'язними" (або, що (майже) те ж саме, "непов'язаними"). Тобто всередині них немає зв'язків (їх елементи не пов'язані один з одним фізично).

Типовим прикладом незвязного (фізично) списку є масив. У цьому розділі ми розглянемо ті самі "спеціальні динамічні структури даних", які і отримали назву зв'язкових списків.

Види списків

Зв'язкові списки

Згадаймо загальні риси черг і стеків:

Суворі правила доступу до даних;

Операції вилучення (зчитування) даних є руйнівними.

Зв'язкові списки вільні від цих обмежень. Вони допускають гнучкі методи доступу; витяг (читання) елемента зі списку не призводить до видалення його зі списку і втрати даних. Для фактичного видалення елемента з зв'язкового списку потрібна спеціальна процедура.

Зв'язкові списки являють собою (як уже було сказано) динамічні (фактично, лінійні!) Структури даних (динамічні ланцюжка ланок), в яких однотипні елементи (ланки) будь-яким чином пов'язані між собою, зазвичай на фізичному рівні. Зв'язок між елементами можна здійснити за рахунок зберігання в одному елементі адреси іншого такого ж елемента (того ж типу). Тобто кожен інформаційний елемент містить усередині себе покажчик на власний тип. Враховуючи, що окрім цього покажчика повинні бути присутніми корисні дані, тип інформаційного елемента виявляється записом. Наприклад, для найпростішого виду списку цей тип може бути таким (на мовах Сі / Сі + +):

```
struct Link1
{
  int data;
  Link1 * next;
};
```

Класифікація зв'язкових списків. За кількістю зв'язків (і одночасно, спрямуванням) списки бувають однозв'язний (односпрямованим), двозв'язним (двонаправленими) і багатозв'язних.

За способом організації зв'язків (або з архітектури) списки можуть бути лінійними і кільцевими (циклічними). (Якщо список не є ні лінійним, ні кільцевих, то залишається єдиний варіант - розгалужених список, що фактично є однією з деревовидних структур даних.)

За ступенем упорядкованості збережених даних списки можуть бути впорядкованими і неупорядкованими. Такий поділ іноді буває зручно на практиці, але для підтримки впорядкованості списків доводиться вдаватися до спеціальних заходів.

Для списків, у порівнянні з чергами і стеками, є значно більше операцій, які включають в себе:

додавання нового ланки списку (вставка ланки);

видалення ланки;

перегляд (або проходження) списку;

пошук даних у списку;

створення ведучого (заголовного) ланки (при необхідності);

сортування списку;

звернення (реверсування) списку, тобто перестановка всіх його ланок у зворотному порядку.

Розглянемо основні з цих операцій для кожного виду списку окремо, разом з особливостями цих видів списків.

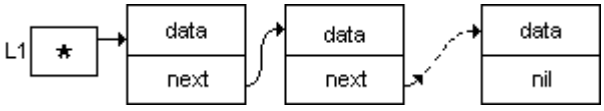
Лінійний однозв'язний список

Лінійний однозв'язний список є найпростішим видом зв'язкових списків. Такий список можна визначити за допомогою описів типів.

Процедури роботи з лінійним однозв'язний списком на мові Паскаль

```
Type
rel1 = ^ elem; (* Покажчик на запис *)
elem = record
next: rel1;
data: <Тип збережених даних> (* Будь допустимий тип даних *)
end;
var
```

L1: rel1;



ддя того, щоб такі операції, як додавання або видалення ланки, виконувалися однаково, незалежно від місця їх виконання в списку, зручно використовувати провідне або головній ланка - саме перша ланка списку, в якому не зобов'язані зберігатися корисні дані. Його створення для односвязного списку можна здійснити наступним чином:

```
var a, L1: list1;
begin
...
 new (L1);
 L1 ^. Next: = nil;
 a: = L1;
...
```

Покажчик на початок списку L1, значенням якого є адреса провідної ланки, представляє список як єдиний програмний об'єкт.

Покажчик на наступний елемент в останньому ланці списку має значення nil (NULL або просто 0 в Ci / Ci + +), що є ознакою лінійного списку.

```
procedure insert1 (link: rel1; info: <Тип>);
var q: rel1;
begin
 new (q);
 q ^. data: = info;
 q ^. next: = link ^. next;
 link ^. next: = q
end;

procedure delete1 (link: rel1);
var q: rel1;
begin
 if link ^. next <> nil then
 begin
  q: = link ^. next;
  link ^. next: = link ^. next ^. next;
  dispose (q);
 end
end;
```

```
function search1 (l: rel1; info: <Тип>; var r: rel1): boolean;
var q: rel1;
begin
 search1: = false;
 r: = nil;
 q: = l ^. next;
 while (q <> nil) and (r <> nil) do
 begin
  if q ^. data = info then
  begin
   search: = true;
   r: = q
  end;
  q: = q ^. next
 end
end;
```

Процедури додавання і видалення ланок є критичними з точки зору збереження цілісності списку. При неправильному виконанні цих процедур (тобто при неправильній черговості виконання операцій присвоювання) можливі 2 помилкові ситуації:

1. Список "рветься" за місцем вставки або видалення ланки, і ланка, яка надала останнім, замикається або саме на себе (найчастіше) (тобто покажчик next або аналогічний йому в цій ланці отримує значення адреси цього ж ланки), або на одне з попередніх ланок (в залежності від неправильної реалізації операцій вставки або видалення ланки). При спробі перегляду списку процедура перегляду зациклюється і нескінченно виводить вміст одного і того ж ланки (або декількох ланок).

2. Список так же "рветься" за місцем вставки або видалення ланки, але покажчик в ланці, що став останнім, отримує якесь довільне значення, яке трактується як адресу наступного ланки (реально не існуючого), у якого також є вказівник next, що містить який- то адресу, і так далі, до тих пір, поки випадково не попадеться блок даних, для якого покажчик next НЕ будер дорівнює нулю. При спробі перегляду списку на дисплей спочатку виводяться правильні дані, а потім випадковий набір символів.

В обох випадках до ланок в "відірвалася" частини ("хвості") списку більше немає доступу, і зберігаються в них дані можна вважати втраченими.

Для запобігання виникнення таких помилок слід дотримувати правильний порядок проведення зв'язків (тобто привласнення покажчиків) при вставці нового ланки і видаленні існуючого (черговість операцій вказана):

Додавання ланки в довільну позицію за провідною ланкою:

```
struct Link1
{
```

```
 int data;
 Link1 * next;
};
 void Insert1 (Link1 * link, int data) / / link - ланка, за яким вставляється нове
{
 Link1 * q = new Link1; / / 1 Виділення пам'яті під нова ланка
 q-> data = data; / / 2 Введення даних
   q-> next = link-> next; / / 3 Проведення зв'язку від нової ланки до наступного
 link-> next = q; / / 4 Проведення зв'язку від "старого" ланки у новому
}
```

Можливість переміщатися по 1-зв'язного списку тільки в одному напрямку призводить до того, що при видаленні ланки доводиться задавати не реально видалити ланка, а попереднє йому. Це робиться для того, щоб можна було скоригувати зв'язок для попереднього ланки, дістатися до якого від видаляється інакше неможливо.

Видалення ланки з будь-якого місця списку за провідною ланкою:

```
 void Delete1 (Link1 * link) / / link - ланка, що передує видаляється
{
 Link1 * q;
 if (link-> next) / / Перевірка на наявність ланки, наступного за link
 {
   q = link-> next / / 1 Запам'ятовування видаляється ланки для операції delete
     link-> next = q-> next; / / 2 Проведення нового зв'язку в обхід видаляється ланки
  delete q; / / 3 Очищення пам'яті
 }
}
```

Однією з найбільш простих операцій з усіма типами списків є їх проходження, тобто почергове отримання доступу до всіх елементів. Наведемо процедуру, що реалізує цю операцію для перегляду списку (інші варіанти використання проходження - пошук даних і збереження списку в файл). У разі переміщення по 2-зв'язного списку в "прямому" напрямі ця процедура є однаковою для 1 - та 2-зв'язкового лінійних списків.

Перегляд 1-зв'язкового лінійного списку

```
 void Show (Link1 * link)
{
```

```
  Link1 * q = link-> next; / / Враховується наявність "порожнього" провідної ланки
 while (q) / / або while (q! = NULL)
 {
  cout << q-> data << "; / / або інша операція
  q = q-> next; / / Перехід по списку
 }
 cout << endl;
}
```

Пошук в списку є варіантом операції перегляду і відрізняється тим, що:

1. замість операції виводу на екран (cout << q-> data) використовується операція порівняння шуканих даних з зберігаються в ланках списку;
2. якщо шукані дані знайдені, немає необхідності переміщатися по списку далі.

Пошук в однозв'язний списках має таку особливість. Якщо він виконується в поєднанні з видаленням, то результатом пошуку може (або повинна) бути не та ланка, в якому містяться шукані дані, а попереднє йому. У підсумку для пошуку у списку можуть бути необхідними або дві різні процедури - "звичайна" і знаходить попереднє ланка, або одна універсальна, що дозволяє знайти обидва ланки - з шуканими даними і попереднє йому. Розглянемо як приклад саме цей варіант:

Універсальна процедура пошуку (знаходить ланка з ключем пошуку та попереднє йому)

```
int Search (Link1 * Start, / / Точка початку пошуку
  Link1 * & Find, / / Покажчик для ланки з шуканими даними
  Link1 * & Pred, / / Покажчик для попереднього ланки
  int Key) / / Ключ пошуку
{
 Link1 * Cur = Start-> next; / / Поточне ланка
 Pred = Start; / / Попереднє ланка ("відстає" на 1 крок від поточного)
 int Success = 0; / / Ознака успіху пошуку (встановлений в 0)
 while (Cur &&! Success) / / Операція логічне "І"
 {
  if (Cur-> data == Key) / / знайшли
  {
   Find = Cur; / / Запам'ятовування знайденого ланки
   Success = 1; / / Установка в 1 ознаки успіху
   break; / / Вихід з циклу при вдалому пошуку
  }
  Pred = Cur; / / Переміщення попереднього ланки
  Cur = Cur-> next; / / Перехід по списку вперед
 }
```

```
 return Success;
}
```

Слід зазначити, що можливі різні (у тому числі більш короткі) варіанти реалізації такого алгоритму, наприклад, без змінної Success (замість неї використовується покажчик Find, який до початку пошуку повинен отримати значення NULL і зберегти його при невдалому пошуку).

Процедура, яка знаходить тільки шукане ланка, є більш простий, - в ній не потрібен покажчик Pred і всі оператори, в яких він використовується.

Схожа процедура застосовується для 1-зв'язкового кільцевого списку. Відмінність її від розглянутого прикладу полягає в умові продовження циклу в операторі while:

```
while (Cur! = Start &&! Success) / / Для кільцевого списку
```

Провідне (або заголовної) ланка. Всі наведені вище приклади на мові Сі + + увазі наявність у списку провідної ланки. Створюватися ця ланка може або окремою процедурою, або таким набором операторів (ці ж оператори і будуть перебувати в процедурі):

```
Link1 * L1 = new Link1; / / Виділення пам'яті під ланка

L1-> next = NULL; / / Провідне ланка одночасно є останнім
```

Можлива робота зі зв'язковим списком і без виділеного ведучого ланки, тобто перша ланка є звичайним, в ньому містяться корисні дані. Саме такий вид списку використовувався для організації стека. Ще один приклад використання подібних списків - черга.

Занесення в чергу і витяг з черги, побудованої на основі 2-зв'язкового списку

```
void queue_in (Link2 * & q, / / "Голова" черги
  Link2 * & e, / / "Хвіст" черги
  int k) / / вводяться дані
{
 Link2 * n = new node; / / Новий вузол
 n-> data = k;
 n-> next = q;
 if (q)
  q-> prev = n;
 else
  e = n;
 n-> prev = 0;
 q = n;
}
```

```
int queue_out (Link2 * & q, Link2 * & e)
{
 int k = e-> data;
 Link2 * d = e;
 e = d-> prev;
 if (e)
  e-> next = 0;
 else
 q = e;
 delete d;
 return k;
}
```

Для спрощення обробки "голови" та "хвоста" черги використаний 2-зв'язний список.

В інших випадках (тобто для звичайних списків) використання ведучого ланки є кращим, тому що дозволяє уникнути перевірок при додаванні і видаленні ланок.

Недоліки односвязного списку полягають в наступному:

1. За таким списком можна переміщатися тільки від початкової ланки до кінцевого. Починати можна з будь-якої ланки в списку, але якщо раптом виникне необхідність звернутися до попередніх елементів, доведеться починати з початкової ланки, що незручно, нераціонально і ускладнює алгоритми обробки даних.

2. Наявність тільки однієї зв'язку знижує надійність зберігання даних в 1-зв'язковому списку.

3. Наслідком першого недоліку є ускладнення взаємодії операцій пошуку і видалення.

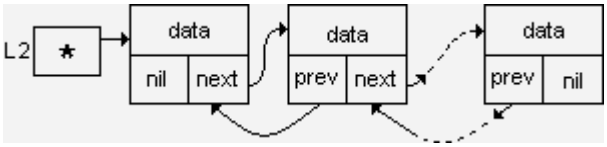
Перевагами цього списку є менша витрата пам'яті в порівнянні з іншими зв'язковими динамічними структурами даних (всього 1 покажчик) і простота операцій.

Лінійний двусвязний список

Цей список вільний від недоліків, властивих односвязного списку. Для цього в кожну ланку доданий ще один покажчик на тип ланки, значенням якого є адреса попереднього ланки списку. Тип ланки на мовах Сі / Сі + +:

```
struct Link2
{
  int data;
  Link2 * next, * prev;
};
```

Структура списку буде виглядати наступним чином

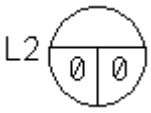


Провідна ланка цього списку створюється наступним набором операторів:

```
Link2 * L2 = new Link2;
L2-> next = NULL;
L2-> prev = NULL;
```

За операторам створення провідної ланки можна судити про те, є список, 1 - або 2-зв'язковим, лінійним чи кільцевим. Нульові значення покажчиків next і prev є ознакою лінійного списку.

Графічно стан списку після створення провідної ланки може бути відображено наступним чином:



Переваги двозв'язним списку:

-є можливість перебудувати пошкоджений список;

-простіше виконуються деякі операції (наприклад, видалення).

Операції з 2-зв'язковим спіском.Все, що стосувалося операцій додавання ланки і його видалення для 1-зв'язкового списку, справедливо і для 2-зв'язкового. Так само повинен дотримуватися правильний порядок проведення (чи видалення) зв'язків між ланками, але таких операцій стало більше, тому що повинні оброблятися 2 покажчика. Крім того, кожна з операцій має додаткові особливостями:

1. При додаванні нової ланки в порожній список, тобто містить лише ведуча ланка (див. попередній малюнок), або при додаванні ланки в кінець списку (для порожнього списку обидві ці ситуації збігаються) необхідно перевіряти наявність ланки, яке буде наступним за додаються (для порожнього списку або кінця списку такого ланки немає, а значить немає і покажчика, що позначає зв'язок). Така ж перевірка повинна виконуватися при видаленні ланки.

2. Можливість переміщатися по списку в обох напрямках дозволяє безпосередньо задавати видалити ланка (а не йому передує, як в 1-зв'язковому списку). Наслідком цього є спрощення як операції видалення, так і операції пошуку.

Додавання ланки в довільну позицію за провідною ланкою

```
void Insert2 (Link2 * St, int data)
```

```
{
 Link2 * q = new Link2; / / 1 Виділення пам'яті під ланка
 q-> data = data; / / 2 Введення даних
 q-> next = St-> next; / / 3 Проведення зв'язку від нової ланки вперед
 q-> prev = St; / / 4 Проведення зв'язку від нової ланки назад
 St-> next = q; / / 5 Проведення зв'язку від попередньої ланки до нового
 if (q-> next) / / Перевірка наявності наступного ланки
   q-> next-> prev = q; / / 6 Проведення зв'язку від наступного ланки до
нового
}
```

Видалення ланки з будь-якого місця списку за провідною ланкою

```
void Delete2 (Link2 * del)
{
 del-> prev-> next = del-> next; / / 1 Обробка зв'язку вперед
 if (del-> next)
  del-> next-> prev = del-> prev; / / 2 Обробка зв'язку назад
 delete del; / / 3 Звільнення пам'яті
}
```

Пошук в 2-зв'язковому списку

```
int Search2 (Link2 * Start, Link2 * & Find, int Key)
{
 Link2 * Cur = Start-> next;
 int Success = 0;
 while (Cur &&! Success)
 {
  if (Cur-> data == Key)
  {
   Find = Cur;
   Success = 1;
   break;
  }
 Cur = Cur-> next;
 }
 return Success;
}
```

Ця процедура пошуку фактично є (за винятком типу даних Link2 замість Link1) процедурою "звичайного" пошуку (тобто не для видалення) в 1-зв'язковому списку.

Операція переглянуду список в прямому напрямку нічим не відрізняється від перегляду 1-зв'язкового списку.

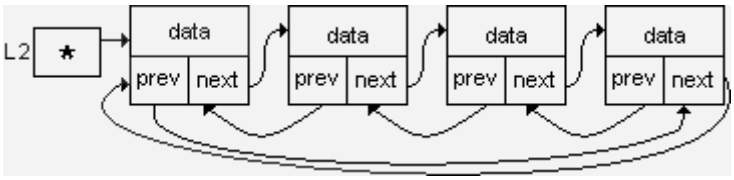
Кільцеві списки

Якщо значення покажчика останньої ланки лінійного односвязного списку замінити з nil (або NULL) на адресу ведучого ланки, то лінійний список перетвориться на однозв'язний кільцевої список.

Для двозв'язним списку, крім того, потрібно замінити з nil на адресу останньої ланки значення другого покажчика в провідному ланці. Вийде двусвязний кільцевої (циклічний) список.

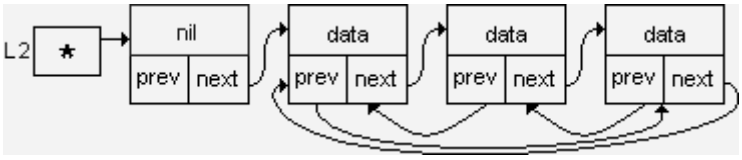
В односвязной кільцевому списку можна переходити від останньої ланки до заголовному, а в двозв'язним - ще й від заголовного до останнього.

Двусвязний кільцевої список виглядає так:



Кільцевій список, як і лінійний, ідентифікується як єдиний програмний об'єкт покажчиком, наприклад L2, значенням якого є адреса заголовного ланки.

Можливий інший варіант організації кільцевого списку:



Обидва варіанти можна порівняти по складності. Для першого варіанту простіше виконується вставка нового елемента як в початок списку (після заголовного ланки), так і в кінець - так як вставка ланки в кінець кільцевого списку еквівалентна вставці перед заголовним ланкою, але кожен раз при циклічної обробки списку потрібно перевіряти, чи не є поточне ланка заголовним (або не співпадає поточне ланка з точкою початку обробки).

Розглянемо операції з кільцевими списками.

Відсутність "останнього" ланки призводить до ще більшого спрощення операцій додавання і видалення, порівняно з 1 - та 2-зв'язковим лінійним списком. Наприклад, для 1-зв'язкового кільцевого списку в процедурі видалення відсутній оператор if - перевірка на існування ланки, наступного за заданим (в кільцевому списку таке ланка завжди є). Такі ж оператори

відсутні в процедурах додавання і видалення ланок для 2-зв'язкового кільцевого списку.

При циклічної обробці кільцевого списку потрібно врахувати, що формально останньої ланки немає.

Програма роботи з двозв'язним кільцевих списком на мові Паскаль

```
Type
 rel2 = ^ elem2;
 elem2 = record
  next: rel1;
  prev: rel2;
  data: <Тип збережених даних>
 end;
list2 = rel2;

procedure insert2 (pred: rel2; info: <Тип>);
var
 q: rel2;
begin
 new (q); (* Створення нової ланки *)
 q ^. data: = info;
 q ^. next: = pred ^. next;
 q ^. prev: = pred ^. next ^. prev;
 pred ^. next.prev: = q;
 pred ^. next: = q
end;
```

При вставці в початок списку (після заголовного ланки) потрібно вказати як аргумент pred адресу заголовного ланки, тобто покажчик на список L2.

```
procedure delete2 (del: rel2);
begin
 del ^. next ^. prev: = del ^. prev;
 del ^. prev ^. next: = del ^. next;
 dispose (del);
end;

function search2 (list: rel2; info: <Тип>; var point: rel2): boolean;
var
 p, q: rel2;
b: boolean;
begin
 b: = false;
 point: = nil;
```

```
 p: = list;
 q: = p ^. next;
 while (p <> q) and (not b) do
 begin
  if q ^. data = info then
  begin
   b: = true;
   point: = q
  end;
  q: = q ^. next
 end;
 search2: = b
end;
 ...
 ...

var
 l2: list2;
 r: rel2;
begin (* Створення заголовного ланки *)
 new (r);
 r ^. next: = r;
 r ^. pred: = r;
 l2: = r;
 ...
 ...

end.
```

Процедури роботи з двозв'язним кільцевих списком на мові Сі + +

Тип даних для кільцевого 2-зв'язкового списку такий же, як і для 2-зв'язкового лінійного. Те ж саме справедливо для 1-зв'язкових списків. За типом ланки списку не можна судити про його архітектурі, можна - тільки про кількість зв'язків.

Додавання ланки в довільну позицію за провідною ланкою

```
 void Insert (Link2 * Pred, int data)
 {
  Link2 * Loc = new Link2; / / 1
  Loc-> Value = data; / / 2
  Loc-> next = Pred-> next; / / 3
  Loc-> prev = Pred; / / 4
  Pred-> next = Loc; / / 5
  Loc-> next-> prev = Loc; / / 6
 }
```

Видалення ланки з будь-якого місця списку за провідною ланкою

```
void Delete2 (Link2 * Del)
{
 Del-> prev-> next = Del-> next; / / 1
 Del-> next-> prev = Del-> prev; / / 2
 delete Del; / / 3
}
```

Пошук

```
int Search (Link2 * Start, Link2 * & Find, int Key)
{
 Link2 * Cur = Start-> next;
 int Success = 0;
 while (Cur! = Start &&! Success)
 {
  if (Cur-> Value == Key)
  {
   Find = Cur;
   Success = 1;
   break;
  }
  Cur = Cur-> next;
 }
 return Success;
}
```

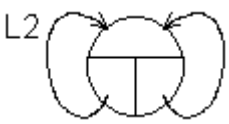
Припинення пошуку в разі невдачі відбувається при досягненні "поточним" покажчиком Cur точки початку пошуку Start, тобто при невиконанні умови в операторі while:

```
while (Cur! = Start ...)
```

Провідне ланка кільцевого 2-зв'язкового списку створюється набором операторів:

```
Link2 * L2 = new Link2;
L2-> next = L2;
L2-> prev = L2;
```

Графічно стан списку після створення цієї ланки може бути представлено таким чином:



Багатозв'язні списки

Багатозв'язних списки являють собою динамічні структури даних, в основу яких покладені 1 - або 2-зв'язкові списки, в яких є додаткові зв'язки між ланками. Найчастіше, такі зв'язки проводяться між далеко відстоять ланками, наприклад, що позначають категорії даних. Приклад многосвязной списку показаний на наступному малюнку.

Перехід між ланками АА і БА може виконаний за додатковою зв'язку, в обхід ланок АБ і АВ. Через такого характеру переміщення ці списки іноді називають скіп-списками (skip - перестрибувати). А при характері розміщення даних, подібному показаному на цьому малюнку, такі списки називають словниковими (іноді просто словниками, але термін "словник" може використовуватися в теорії структур даних в різних значеннях).

11.2 Операція зі списками

Операції зі списками при послідовному збереженні

При виборі методу зберігання лінійного списку слід враховувати, які операції будуть виконуватися і з якою частотою, час їх виконання та обсяг пам'яті, необхідний для збереження списку.

Нехай є лінійний список з цілими значеннями і для його зберігання використовується масив d (з числом елементів 100), а кількість елементів у списку вказується змінної l. Реалізація зазначених раніше операцій над списком представляється наступними фрагментами програм які використовують оголошення:

```
float d [100];
        int i, j, l;
   1) друк значення першого елемента (вузла)
     if (i <0 | | i> l) printf ("\ n немає елемента");
     else printf ("d [% d] =% f", i, d [i]);
   2) видалення елемента, наступного за i-тим вузлом
     if (i> = l) printf ("\ n немає наступного");
     l -;
     for (j = i +1; j = l) printf ("\ n немає сусіда");
     else printf ("\ n% d% d", d [i-1], d [i +1]);
   4) додавання нового елемента new за i-тим вузлом
     if (i == l | | i> l) printf ("\ n можна додати");
     else
     {For (j = l; j> i +1; j -) d [j +1] = d [j];
```

```
        d [i +1] = new; l + +;
    }
```

5) часткове упорядкування списку з елементами К1, К2, ..., KL в список K1 ', K2', ..., Ks, K1, Kt ", ..., Kt", s + t +1 = l так, щоб K1 '= K1.

```
{Int t = 1;
 float aux;
 for (i = 2; i <= l; i + +)
  if (d [i] = 2; j -) d [j] = d [j-1];
       t + +;
       d [i] = aux;
     }
}
```

Кількість дій Q, необхідних для виконання наведених операцій над списком, визначається співвідношеннями: для операцій 1 і 2 - Q = 1; для операцій 3,4 - Q = l; для операції 5 - Q = l * l.

Зауважимо, що взагалі операцію 5 можна виконати при кількості дій порядку l, а операції 3 і 4 для включення і виключення елементів в кінці списку, часто зустрічаються при роботі зі стеками, - при кількості дій 1.

Більш складна організація операцій потрібно при розміщенні в масиві d декількох списків, або при розміщенні списку без прив'язки його початку до першого елементу масиву.

Операції зі списками при зв'язному збереженні

При простому зв'язаному збереженні кожен елемент списку являє собою структуру nd, що складається з двох елементів: val - призначений для зберігання елемента списку, n - для покажчика на структуру, яка містить наступний елемент списку. На перший елемент списку вказує покажчик dl. Для всіх операцій над списком використовується опис:

```
typedef struct nd
        {Float val;
          struct nd * n;} ND;
      int i, j;
      ND * dl, * r, * p;
```

Для реалізації операцій можуть використовуватися наступні фрагменти програм:

1) друк значення i-го елемента

```
      r = dl; j = 1;
      while (r! = NULL && j + + n;
      if (r == NULL) printf ("\ n немає вузла% d", i);
      else printf ("\ n елемент% d дорівнює% f", i, r-> val);
```

2) друк обох сусідів вузла (елементу), що визначається покажчиком p

```
if ((r = p-> n) == NULL) printf ("\ n немає сусіда справа");
        else printf ("\ n сусід справа% f", r-> val);
        if (dl == p) printf ("\ n немає сусіда зліва");
        else {r = dl;
              while (r-> n! = p) r = r-> n;
              printf ("\ n лівий сусід% f", r-> val);
             }
```

3) видалення елемента, наступного за вузлом, на який вказує p

```
if ((r = p-> n) == NULL) printf ("\ n немає наступного");
        p-> n = r-> n; free (r-> n);
```

4) вставка нового вузла зі значенням new за елементом, визначеним покажчиком p

```
r = malloc (1, sizeof (ND));
        r-> n = p-> n; r-> val = new; p-> n = r;
```

5) часткове упорядкування списку в послідовність значень, s + t +1 = l, так що K1 '= K1; після упорядкування покажчик v вказує на елемент K1'

```
ND * v;
        float k1;
        k1 = dl-> val;
        r = dl;
        while (r-> n! = NULL)
        {V = r-> n;
          if (v-> valn = v-> n;
             v-> n = dl;
             dl = v;
            }
           else r = v;
        }
```

Кількість дій, необхідних для виконання зазначених операцій над списком у зв'язаному зберіганні, оцінюється співвідношеннями: для операцій 1 і 2 - Q = l; для операцій 3 і 4 - Q = 1; для операції 5 - Q = l.

11.3 Черга

Черга - структура даних з дисципліною доступу до елементів «перший прийшов - перший вийшов» (FIFO, First In - First Out). Додавання елемента (прийнято позначати словом enqueue - поставити в чергу) можливе лише в

кінець черги, вибірка - тільки з початку черги (що прийнято називати словом dequeue - прибрати з черги), при цьому обраний елемент з черги видаляється.

Черга в програмуванні використовується, як і в реальному житті, коли потрібно зробити якісь дії в порядку їх надходження, виконавши їх послідовно. Прикладом може служити організація заходів в Windows. Коли користувач робить якусь дію на додаток, то в додатку не викликається відповідна процедура (адже в цей момент додаток може здійснювати інші дії), а йому надсилається повідомлення, що містить інформацію про вчинений дії, це повідомлення ставиться в чергу, і лише коли будуть оброблені повідомлення, що прийшли раніше, додаток виконає необхідну дію.

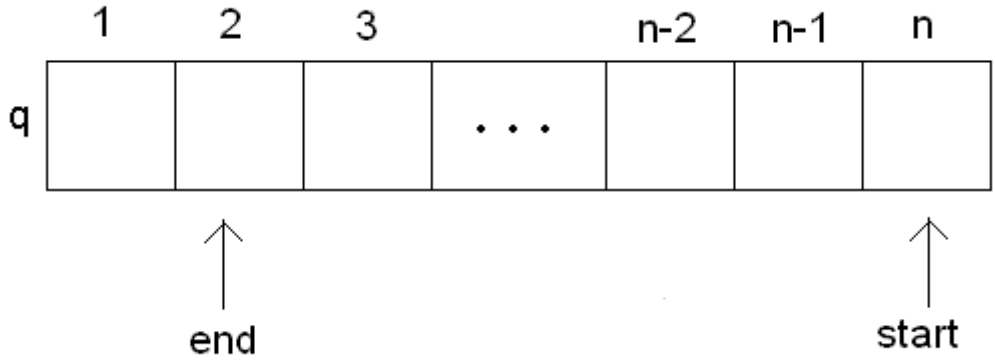
Клавіатурний буфер BIOS організований у вигляді кільцевого масиву, звичайно довжиною в 16 машинних слів, і двох покажчиків: на наступний елемент в ньому і на перший незайнятий елемент.

Способи реалізації черги

Існує кілька способів реалізації черги в мовах програмування.

Масив

Перший спосіб являє чергу у вигляді масиву і двох цілочисельних змінних start і end.



Зазвичай start вказує на голову черги, end - на елемент, який заповниться, коли в чергу увійде новий елемент. При додаванні елемента в чергу в q [end] записується новий елемент черги, а end зменшується на одиницю. Якщо значення end стає менше 1, то ми як би циклічно обходимо масив і значення змінної стає рівним n. Витяг елемента з черги проводиться аналогічно: після вилучення елемента q [start] з черги мінлива start зменшується на 1. З такими алгоритмами один осередок з n завжди буде незайнятої (так як черга з n елементами неможливо відрізнити від порожньої), що компенсується простотою алгоритмів.

Переваги даного методу: можлива незначна економія пам'яті в порівнянні з другим способом; простіше в розробці.

Недоліки: максимальна кількість елементів у черзі обмежена розміром масиву. При його переповненні потрібно перевиделеніе пам'яті і копіювання всіх елементів в новий масив.

Зв'язний список

Другий спосіб заснований на роботі з динамічною пам'яттю. Черга представляється як лінійного списку, в якому додавання / видалення елементів йде строго з відповідних його кінців.

Переваги даного методу: розмір черги обмежений лише обсягом пам'яті.

Недоліки: складніше в розробці; потрібно більше пам'яті; при роботі з такою чергою пам'ять сильніше фрагментіруется; робота з чергою дещо повільніше.

Реалізація на двох стеках

Черга може бути побудована з двох стеків S1 та S2 як показано нижче:

```
Процедура enqueue (x):
    S1.push (x)

Процедура dequeue ():
    якщо S2 порожній:
        якщо S1 порожній:
            повідомити про помилку: черга порожня

        поки S1 не порожній:
            S2.push (S1.pop ())

    return S2.pop ()
```

Такий спосіб реалізації найбільш зручний в якості основи для побудови персистентної черги.

11.4 Поняття деку. Стек

Поняття деку

Дек - особливий вид черги. Дек (від англ. Deq - double ended queue, т.е чергу з двома кінцями) - це такий послідовний список, в якому як включення, так і виключення елементів може здійснюватися з будь-якого з двох кінців списку. Окремий випадок дека - дек з обмеженим входом і дек з обмеженим виходом. Логічна і фізична структури дека аналогічні логічної і фізичної

структурі кільцевої FIFO-черзі. Однак, стосовно до деку доцільно говорити не про початок і кінець, а про лівому і правому кінці.

Операції над Деком:

включення елемента справа;
включення елемента зліва;
виключення елемента справа;
виключення елемента зліва;
визначення розміру;
очистка.

Фізична структура дека в статичній пам'яті ідентична структурі кільцевої черги. Динамічна реалізація є очевидним об'єднанням стека і черги.

Задачі, що вимагають структури дека, зустрічаються в обчислювальній техніці і програмуванні набагато рідше, ніж завдання, реалізовані на структурі стека або черги. Як правило, вся організація дека виконується програмістом без яких-небудь спеціальних засобів системної підтримки.

Прикладом дека може бути, наприклад, якийсь термінал, в який вводяться команди, кожна з яких виконується якийсь час. Якщо ввести наступну команду, не дочекавшись, поки закінчиться виконання попередньої, то вона встане в чергу і почне виконуватися, як тільки звільниться термінал. Це FIFO чергу. Якщо ж додатково ввести операцію скасування останньої введеної команди, то виходить дек.

Стек

Стеки - це безліч елементів, складених у стопку. Наприклад, у нас є коробка 3x5. Ми кладемо в неї дошки такого ж розміру з різними візерунками. У нас виходить стек. Дістати з нього ми можемо тільки починає з останнього елемента, тому перший покладені елемент виймуть буде останнім. У стеках реалізується принцип first in last out (FILO).

Для створення стека потрібно підключити <stack> і в коді програми його оголосити:

stack <type> name, де type - тип стека, а name - ім'я стека.

У стека є трохи нижче:
push () - додати елемент
pop () - видалити верхній елемент
top () - отримати верхній елемент
size () - розмір стека
empty () - true, якщо стек порожній

приклад:

```
string s;

stack <string> st;

while (cin >> s);

st.push (s);

while (! (st.empty ()))

{cout << st.top (); st.pop ()}
```

У цьому прикладі ми зчитуємо слова з клавіатури і виводимо їх у зворотному порядку.

11.5 Дерева.

Дерево - одна з найбільш широко поширених структур даних в інформатиці, що емулює деревоподібну структуру у вигляді набору пов'язаних вузлів. Є пов'язаною графом, що не містить цикли. Більшість джерел також додають умову на те, що ребра графа не повинні бути орієнтованими. На додаток до цих трьох обмеженням, в деяких джерелах вказуються, що ребра графа не повинні бути зваженими.

11.6 Двійкові дерева аба бінарні

Бінарне (двійкове) дерево (binary tree) - це впорядковане дерево, кожна вершина якого має не більше двох піддерев, причому для кожного вузла виконується правило: в лівому піддереві містяться тільки ключі, що мають значення, менші, ніж значення даного вузла, а в правому піддереві містяться тільки ключі, що мають значення, більші, ніж значення даного вузла.

Бінарне дерево є рекурсивної структурою, оскільки кожне його піддерево саме є бінарним деревом і, отже, кожен його вузол в свою чергу є коренем дерева.

Вузол дерева, що не має нащадків, називається листом.

Бінарне дерево може являти собою порожній безліч.

Побудова бінарного дерева

Спочатку ми повинні визначити структуру для створення кореня і вузлів дерева:

```
template <class T>
struct TNode {
```

```
  T value;
  TNode * pleft, * pright;
  / / Constructor
  TNode () {
    pleft = pright = 0;
  }};
```

Тут поля pleft і pright - це покажчики на нащадків даного вузла, а поле value призначено для зберігання інформації.

Тепер ми можемо написати рекурсивну функцію, яка буде викликатися при створенні дерева:

```
template <class T>
void makeTree (TNode <T> ** pp, T x) {
  if (! (* pp)) {
    TNode <T> * p = new TNode <T> ();
    p-> value = x;
    * Pp = p;
  }
  else {
    if ((* pp) -> value> x)
      makeTree (& ((* pp) -> pleft), x);
    else
      makeTree (& ((* pp) -> pright), x);
  }}
```

Ця функція додає елемент x до дерева, враховуючи величину x. При цьому створюється новий вузол дерева.

Обхід дерева

Функція, що виконує обхід дерева, дозволяє перебрати всі елементи, що містяться в дереві.

У наведеній нижче реалізації функція обходу дерева буде просто виводити на екран значення поля value кожного вузла дерева (включаючи його корінь):

```
template <class T>
void walkTree (TNode <T> * p) {
  if (p) {
    walkTree (p-> pleft);
    cout << p-> value << ";
    walkTree (p-> pright);
  }}
```

При роботі з деревами зазвичай використовуються рекурсивні алгоритми. Використання рекурсивних функцій менш ефективно, оскільки багаторазовий виклик функції витрачає системні ресурси. Тим не менш, в даному випадку використання рекурсивних функцій є виправданим, оскільки нерекурсівние функції для роботи з деревами набагато складніше і для написання, і для сприйняття коду програми.

Наприклад, нерекурсівние функція для обходу дерева може виглядати так:

```
template <class T>
void walkNotRec (TNode <T> * tree) {
  if (tree) {
    TNode <T> temp;
    TNode <T> * ptemp = &temp;
    TNode <T> * p = ptemp, * c = tree, * w;
    while (c! = ptemp) {
      cout << c-> value;
      w = c-> pleft;
      c-> pleft = c-> pright;
      if (c == p)
         c-> pright = 0;
      else
         c-> pright = p;
             p = c;
      if (w) c = w;
           }
  }}
```

Застосування

Організація даних за допомогою бінарних дерев часто дозволяє значно скоротити час пошуку потрібного елементу. Пошук елемента в лінійних структурах даних звичайно здійснюється шляхом послідовного перебору всіх елементів, присутніх в даній структурі. Пошук по дереву не вимагає перебору всіх елементів, тому займає значно менше часу. Максимальне число кроків при пошуку по дереву одно висоті даного дерева, тобто кількості рівнів в ієрархічній структурі дерева.

11.7 Реалізація дерев

Для реалізації ми створимо два класи: Tree і TreeIterator. Звичайно ж ми будемо створювати шаблонні класи.

Звернемо увагу на той факт, що кожен вузол дерева, сам є піддерево. А це означає, що дерево - рекурсивна структура даних.

Клас дерева включає в себе змінну зберігає значення вузла, покажчик на батьківський елемент і список синів. Для зберігання списку синів ми скористаємося класом DLinkedList.

```
template
class Tree
{
public:
  T data;
  Tree * parent;
  DLinkedList *> sons; / / список покажчиків на Tree
  Tree (): parent (NULL) {}
  ~ Tree ();
};
```

data - мінлива що зберігає дані вузла.

parent - покажчик на батьківський елемент. У кореня дана змінна буде дорівнює NULL.

sons - сини вузла. Для зберігання всіх синів ми використовуємо клас DlinkedList. Тобто sons - список покажчиків, елементами якого є дерева Tree.

11.8 Обхід дерева

Існує досить багато алгоритмів роботи з деревовидними структурами, в яких часто зустрічається поняття обходу (traversing) дерева або "проходу" по дереву. При такому методі дослідження дерева кожний вузол відвідується тільки один раз, а повний обхід задає лінійне упорядкування вузлів, що дозволяє спростити алгоритм, тому що при цьому можна використовувати поняття "наступний" вузол. тобто вузол стоїть після даного при обраному порядку обходу.

Існує кілька принципово різних способів обходу дерева:

Обхід в прямому порядку

Кожен вузол відвідується до того, як відвідані його нащадки.

Для кореня дерева рекурсивно викликається наступна процедура:

-Відвідати вузол
-Обійти ліве піддерево
-Обійти праве піддерево

Приклади використання обходу:
рішення задачі методом поділу на частини

разузлование зверху
стратегія "розділяй і володарюй" (Сортування Фон Неймана, швидка сортування, одночасне перебування максимуму і мінімуму послідовності чисел, множення довгих чисел).

Симетричний обхід

Відвідуємо спочатку ліве піддерево, потім вузол, потім - праве піддерево.

Для кореня дерева рекурсивно викликається наступна процедура:

-Обійти ліве піддерево
-Відвідати вузол
-Обійти праве піддерево
-Обхід в зворотному порядку

Вузли відвідуються 'знизу вгору ».

Для кореня дерева рекурсивно викликається наступна процедура:

-Обійти ліве піддерево
-Обійти праве піддерево
-Відвідати вузол

Приклади використання обходу:

-аналіз ігор з повною інформацією
-разузлование знизу
-динамічне програмування

Обхід в ширину

При обході в ширину вузли відвідуються рівень за рівнем (N-й рівень дерева - безліч вузлів з висотою N). Кожен рівень обходиться зліва направо.
Для реалізації використовується структура queue - черга з методами

enqueue - поставити в чергу
dequeue - взяти з черги
empty - повертає TRUE, якщо чергу порожня, інакше - FALSE

```
q.enqueue (root); / / корінь в чергу
while (! q.empty) {
  x = q.dequeue ();
  visit x; / / відвідати x
  if (! x.left.empty) / / x.left - ліве піддерево
    q.enqueue (x.left);
```

```
  if (! x.right.empty) / / x.right - праве піддерево
    q.enqueue (x.right);
}
```

Рекурсивні обходи можна, очевидно, організувати і за допомогою стека, 'розгорнувши' рекурсію.

11.9 Сортування та пошук за допомогою дерев

Сортування за допомогою двійкового дерева (сортування двійковим деревом, сортування деревом, деревна сортування, сортування з допомогою бінарного дерева, англ. Tree sort) - універсальний алгоритм сортування, який полягає в побудові двійкового дерева пошуку по ключам масиву (списку), з наступною збіркою результуючого масиву шляхом обходу вузлів побудованого дерева в необхідному порядку проходження ключів. Дана сортування є оптимальною при отриманні даних шляхом безпосереднього читання з потоку (наприклад з файлу, сокета або консолі).

Алгоритм

1. Побудова двійкового дерева.
2. Збірка результуючого масиву шляхом обходу вузлів в необхідному порядку проходження ключів.

Ефективність

Процедура додавання об'єкта в бінарне дерево має середню алгоритмічну складність порядку O (log (n)). Відповідно, для n об'єктів складність буде становити O (n log (n)), що відносить сортування за допомогою двійкового дерева до групи «швидких сортувань». Однак, складність додавання об'єкта в розбалансоване дерево може досягати O (n), що може привести до загальної складності порядку O (n ²).

При фізичному розгортанні деревоподібної структури в пам'яті потрібно не менш ніж 4n осередків додаткової пам'яті (кожен вузол повинен містити посилання на елемент початкового масиву, на батьківський елемент, на лівий і правий лист), проте, існують способи зменшити необхідну додаткову пам'ять.

Реалізація на C + +:

```
#include <set>
#include <algorithm>
#include <iterator>

template <typename Iterator>
```

```
void binary_tree_sort(Iterator begin, Iterator end) {
    std::multiset<typename std::iterator_traits<Iterator>::value_type> tree(begin, end);
    std::copy(tree.begin(), tree.end(), begin);
};
```

11.10 Контейнери та ітератори

До найбільш популярних типів класів відносяться класи, контейнери (звані також класи сукупностей), тобто класи, спроектовані для зберігання в них сукупностей об'єктів. Класи контейнери зазвичай забезпечені такими можливостями, як вставка, видалення, пошук, сортування, перевірка наявності елемента в класі тощо. Масиви, стеки, черги, зв'язкові списки - все це приклади класів контейнерів.

Прийнято асоціювати об'єкти ітератори, або, коротше - ітератори, з класами контейнерами. Ітератор - це об'єкт, який повертає наступний елемент сукупності (або визначає деяку дію над наступним елементом сукупності). Коли написаний ітератор класу, легко отримати наступний елемент цього класу. Ітератори зазвичай пишуться як друзі класів, з якими вони працюють. Це надає ітератора можливість прямого доступу до закритих даними цих класів. Подібно до того, як книга, що читається кількома людьми, могла б мати в собі відразу кілька закладок, клас контейнер може мати кілька одночасно працюючих итераторов. Кожен ітератор підтримує свою власну «позицію» інформації.

Асоціативні контейнери

Називають відображенням або карткою (map) або словником (dictionary). Містить пов'язані пари "ключ-значення" ("key-value"). Т.ч. асоціативний масив - це масив, для якого індекс може мати нецілочисельне значення. Бібліотека STL надає такі асоціативні контейнери:

map - асоціативний масив, по ключу в контейнері зберігається одне значення

multimap - асоціативний масив з повторюваними ключами

set - асоціативний масив унікальних ключів без значень

multiset - асоціативний масив з повторюваними ключами без значень

Асоціативні контейнери організовуються як збалансоване дерево вузлів, значення якого представляє собою пару значень pair <const key, mapped_type>.

Переміщення з використанням ітератора також справедливі як і для інших контейнерів, але при разименованія ітератора? примірник об'єкта pair.

Перший елемент пари first є ключем, а другий значенням.

```
map <string, int> book;

map <string, int> :: iterator i;

for (i = book.begin (); i! = book.end (); i + +)

{

cout << p-> first << "" << p-> second << endl;

}
```

Але більш типова робота зі значеннями зберігаються в асоціативному масиві - це доступ до елементів по ключу. При цьому використовується звичний оператор [].

Оператор індикації виконує пошук по ключу, заданому в якості індексу і повертає відповідне значення.

Якщо ключ не знайдений, то в асоціативний масив вставляється елемент з цим ключем і значенням за замовчуванням (для вбудованих типів 0).

```
map <string, float> m;

int x = m ["Бублик"]; / / т.к. масив пустий, то створюється новий запис

 / / I ініціалізується 0

m ["Бублик"] = 1.60f; / / присвоюємо значення записи з ключем

...

float total;

map <string, float> :: iterator i;

for (i = m.begin (); i! = m.end (); i + +)

{

total + = p-> second;
```

```
cout << p-> first << 't' << p-> second << endl;

}

cout << "---------- n total t" << total << endl;
```

Елементи в асоціативному масиві зберігаються впорядковано (відсортовано), тому будуть виведені в лексикографічному порядку).

Загальні методи для всіх асоціативних контейнерів:

find (key & key) - повертає ітератор, який вказує на елемент із заданим ключем

lower_bound (key) - початок послідовності елементів з ключем key

upper_bound (key) - кінець послідовності елементів з ключем key

equal_range (key) - повертає пару контейнерів first - початок, second - кінець послідовності з ключем key

count (key) - знаходить кількість елементів з ключем key

12. МОДУЛЬ 12: Класи мови C++

12.1 Класи. Приклад класу

Однією з основних рис C + +, якої немає в C, є концепція класів. По суті, класи - найважливіше поняття в C + +. Класи схожі на структури мови C. Проте структура З визначає лише дані, асоційовані з цією структурою. Ось приклад структури C:

```
struct CIRCLE
{
int radius;
int color;
{;
```

Після того як ви оголосили структуру, ви можете використовувати її в межах вашої функції main (), як показано нижче:

```
void main ()
CIRCLE MyCircle;
...
...
MyCircle.radius = 18;
MyCircle.color = 255; / / 255 задає колір
...
...
}
```

Зі структурою MyCircle (представляє окружність) асоціюються дані radius і color (радіус і колір). Клас в C + +, з іншого боку, має як асоційовані з ним дані, так і функції. Дані класу називаються елементами даних, а функції класу - елементами-функціями. Отже, в програмі, яка використовує класи, можна написати наступний код:

```
MyCircle.radius = 20;
MyCircle.color = 255;
MyCircle.DisplayCircle ();
```

Перші два оператори присвоюють значення елементів даних MyCircle radius і color; третій оператор викликає функцію-елемент DisplayCircle () для виводу кола MyCircle. MyCircle називається об'єктом класу circle. Ваша програма може оголосити інший об'єкт з ім'ям HerCircle класу circle наступним чином:

CIRCLE HerCircle;

Наступні оператори присвоюють значення елементів даних HerCircle radius і color:

```
HerCircle.radius = 30;
HerCircle.color = 0;
```

Потім ви можете використовувати функцію-елемент DisplayCircie () для виводу кола HerCircle:

```
HerCircle.DisplayCircle ();
```

Оголошення класу

Перед тим як працювати з класом, ваша програма повинна його оголосити (так само як перед роботою зі структурою mystructure ви повинні були оголосити її елементи даних). В даному розділі ви познайомитеся з синтаксисом оголошення класу. Ви будете і далі практикуватися з класом circle:

```
class Circle (
public:
Circle ();
void SetRadius (void);
void GetRadius (void);

~ Circle ();

private:
void CalculateArea (void);
int radius;

int color;

};
```

Оголошення класу має наступну будову:

```
class Circle {
...
...
Тут ви вводите оголошення класу
...
...
};
```

Ключове слово class показує компілятору, що все знаходиться в фігурних дужках ({}) належить оголошенню класу. (Не забувайте ставити крапку з комою в кінці оголошення.) Оголошення класу містить оголошення елементів даних (наприклад, int radius) і прототипи функцій-елементів класу. В оголошенні класу circle містяться такі елементи даних:

```
int radius;
int color;
```

Оголошення також містить п'ять прототипів функцій-елементів:

```
Circle ();

void SetRadius (void);
void GetRadius (void);
~ Circle ();
void CalculateArea (void);
```

Перший і четвертий прототипи виглядають дивно. Перший з них є прототипом функції конструктора:

```
Circle ();
```

Ви дізнаєтеся про роль конструктора пізніше в цьому розділі, а поки запам'ятайте синтаксис, який використовується в C + + для прототипу функції конструктора. Коли ви записуєте прототип конструктора, ви повинні слідувати правилам, наведеним нижче:

Кожне оголошення класу має включати прототип функції конструктора.

Ім'я функції конструктора має збігатися з ім'ям класу, а після нього повинні слідувати круглі дужки (). Якщо, наприклад, ви розкажете клас з ім'ям Rectangle, він повинен включати оголошення функції конструктора класу: Rectangle (). Отже, оголошення класу Rectangle має виглядати так:

```
class Rectangle
{
public:

Rectangle (); / / Конструктор
...
...
private:
...
...
};
```

Не згадуйте ніякого значення, що повертається для функції конструктора. (Функція конструктора повинна мати тип void, але не потрібно це вказувати.)

Функція конструктора повинна розташовуватися під ключовим словом

public.

Функція конструктора завжди повертає значення типу void (незважаючи на те, що ви не вказали його в прототипі). Як ви незабаром побачите, функція конструктора зазвичай має один або більше число параметрів.

Приклади класів

Програмування без сховування даних (у розрахунку на структури) вимагає меншого попереднього обдумування завдання, ніж програмування з сховуванням даних (у розрахунку на класи). Структуру можна визначити не дуже замислюючись про те, як її будуть використовувати. Коли визначається клас, увага концентрується на тому, щоб забезпечити для нового типу повний набір операцій. Це важливе зміщення акценту в проектуванні програм. Звичайний час, витрачений на розробку нового типу, багаторазово окупається в процесі налагодження і розвитку програми.

Ось приклад закінченого визначення типу intset, що представляє поняття "безліч цілих":

```
class intset {
        int cursize, maxsize;
        int * x;
      public:
        intset (int m, int n); / / не більше m цілих з 1 .. n
        ~ Intset ();

        int member (int t) const; / / чи є t членом?
        void insert (int t); / / додати до безлічі t

        void start (int & i) const {i = 0;}
        void ok (int & i) const {return i <cursize;}
        void next (int & i) const {return x [i + +];}
      };
```

Для перевірки цього класу спочатку створимо, а потім роздрукуємо безліч випадкових цілих чисел. Це просте безліч цілих можна використовувати для перевірки, чи є повторення в їх послідовності. Але для більшості завдань потрібен, звичайно, більш розвинений тип множини. Як завжди можливі помилки, тому потрібна функція:

```
# Include <iostream.h>

      void error (const char * s)
      {
        cerr << "set:" << s << '\ n';
```

```
  exit (1);
}
```

Клас intset використовується в функції main (), для якої має бути задано два параметри: перший визначає число створюваних випадкових чисел, а другий - діапазон їх значень:

```
int main (int argc, char * argv [])
{
 if (argc! = 3) error ("потрібно ставити два параметри");
 int count = 0;
 int m = atoi (argv [1]); / / число елементів множини
 int n = atoi (argv [2]); / / з діапазону 1 .. n
 intset s (m, n);

 while (count <m) {
  int t = randint (n);
  if (s.member (t) == 0) {
     s.insert (t);
     count + +;
  }
 }

 print_in_order (& s);
}
```

Значення лічильника параметрів програми argc дорівнює 3, хоча програма має тільки два параметри. Справа в тому, що в argv [0] завжди передається додатковий параметр, що містить ім'я програми. Функція

```
extern "C" int atoi (const char *)
```

є стандартною бібліотечної функцією, перетворюючої ціле з строкового подання у внутрішню двійкову форму. Як завжди, якщо ви не хочете мати такий опис в своїй програмі, то вам треба включити в неї відповідний заголовний файл, що містить описи стандартних бібліотечних функцій. Випадкові числа генеруються за допомогою стандартної функції rand:

```
extern "C" int rand (); / / будьте обережні:
                        / / Числа не зовсім випадкові
int randint (int u) / / діапазон 1 .. u
{
 int r = rand ();
 if (r <0) r =-r;
 return 1 + r% u;
}
```

Подробиці реалізації класу мало цікаві для користувача, але в будь-якому випадку будуть використовуватися функції-члени. Конструктор розміщує

масив цілих з розміром, рівним заданому максимальному розміру множини, а деструктор видаляє цей масив:

```
intset :: intset (int m, int n) / / не більше m цілих в 1 .. n
{
 if (m <1 | | n <m) error ("неприпустимий розмір intset");
 cursize = 0;
 maxsize = m;
 x = new int [maxsize];
}

intset :: ~ intset ()
{
 delete x;
}
```

Цілі додаються таким чином, що вони зберігаються в безлічі у зростаючому порядку:

```
void intset :: insert (int t)
{
 if (+ + cursize> maxsize) error ("занадто багато елементів");
 int i = cursize-1;
 x [i] = t;

 while (i> 0 && x [i-1]> x [i]) {
   int t = x [i]; / / поміняти місцями x [i] i x [i-1]
   x [i] = x [i-1];
   x [i-1] = t;
   i -;
 }
}
```

Щоб знайти елемент, використовується простий двійковий пошук:

```
int intset :: member (int t) const / / двійковий пошук
{
 int l = 0;
 int u = cursize-1;

 while (l <= u) {
  int m = (l + u) / 2;
  if (t <x [m])
    u = m-1;
  else if (t> x [m])
    l = m +1;
  else
    return 1; / / знайдений
```

```
  }
  return 0; / / не знайдено
}
```

Нарешті, потрібно надати користувачеві набір операцій, за допомогою яких він міг би організувати ітерацію по безлічі в деякому порядку (адже порядок, використовуваний в поданні intset, від нього приховано). Безліч за своєю суттю не є внутрішньо упорядкованим, і не можна дозволити просто вибирати елементи масиву (а раптом завтра intset буде реалізовано у вигляді пов'язаного списку?).

Користувач отримує три функції: start () - для ініціалізації ітерації, ok () - для перевірки, чи є наступний елемент, і next () - для отримання наступного елементу:

```
class intset {
  / / ...
  void start (int & i) const {i = 0;}
  int ok (int & i) const {return i <cursize;}
  int next (int & i) const {return x [i + +];}
};
```

Щоб забезпечити спільну роботу цих трьох операцій, треба запам'ятовувати той елемент, на якому зупинилася ітерація. Для цього користувач повинен задавати цілий параметр. Оскільки наше уявлення безлічі впорядковане, реалізація цих операцій тривіальна. Тепер можна визначити функцію print_in_order:

```
void print_in_order (intset * set)
{
 int var;
 set-> sart (var);
 while (set-> ok (var)) cout << set-> next (var) << '\ n';
}
```

12.2 Приватні та особисті дані

Прототипи функцій і оголошення елементів даних включаються в оголошенні класу в розділи public (відкритий) або private (закритий). Ключові слова public і private кажуть компілятору про доступність елементів-функцій і даних. Наприклад, функція SetRadius () визначена у розділі public, і це означає, що будь-яка функція програми може викликати функцію SetRadius (). Функція CalculateArea () визначена у розділі private, і цю функцію можна викликати тільки в коді функцій-елементів класу Circle

Аналогічно, оскільки елемент даних radius оголошений у розділі private, прямий доступ до нього (для установки або читання його значення) можливий тільки в коді функцій-елементів класу Circle. Якби ви оголосили

елемент даних radius в розділі public, то будь-яка функція програми мала б доступ (для читання і присвоювання) до елементу даних radius.

12.3 Конструктори

В об'єктно-орієнтованому програмуванні конструктор класу (від англ. Constructor, іноді скорочують ctor) - спеціальний блок інструкцій, що викликається при створенні об'єкта.

Конструктор схожий з методом, але відрізняється від методу тим, що не має явно визначеного типу повертаються даних, не успадковується, і зазвичай має різні правила для розглянутих модифікаторів. Конструктори часто виділяються наявністю однакового імені з ім'ям класу, в якому оголошується. Їх завдання - ініціалізувати члени об'єкта і визначити інваріант класу, повідомивши в разі некоректності інваріанта. Коректно написаний конструктор залишить об'єкт в «правильному» стані. Незмінні об'єкти теж повинні бути проініціалізувати конструктором.

Термін «конструктор» також використовується для позначення одного з тегів, що описують дані в алгебраїчному типі даних. Це використання дещо відрізняється від описуваного в статті. Для додаткової інформації дивіться Алгебраїчний тип даних.

В більшості мов конструктор може бути перевантажений, що дозволяє використовувати кілька конструкторів в одному класі, причому кожен конструктор може мати різні параметри.

Одна з ключових особливостей ООП - інкапсуляція: внутрішні поля об'єкта безпосередньо недоступні, і користувач може працювати з об'єктом тільки як з єдиним цілим, через відкриті (public) методи. Кожен метод, в ідеалі, має бути влаштований так, щоб об'єкт, що знаходиться в «допустимому» стані (тобто коли виконується інваріант класу), після виклику методу також виявився в допустимому стані. І перше завдання конструктора - перевести поля об'єкта в такий стан.

Друге завдання - спростити користування об'єктом. Об'єкт - не «річ у собі», йому часто доводиться вимагати якусь інформацію від інших об'єктів: наприклад, об'єкт File, створюючи, повинен отримати ім'я файлу. Це можна зробити і через метод:

```
File file;
  file.open ("in.txt", File :: omRead);
```

Але зручніше відкриття файлу зробити в конструкторі: [1]

```
  File file ("in.txt", File :: omRead);
```

Види конструкторів

Деякі мови програмування розрізняють кілька особливих типів конструкторів:

конструктор за замовчуванням - конструктор, не приймає аргументів;

конструктор копіювання - конструктор, який приймає в якості аргументу об'єкт того ж класу (або посилання з нього);

конструктор перетворення - конструктор, який приймає один аргумент (ці конструктори можуть викликатися автоматично для перетворення значень інших типів в об'єкти даного класу).

```
class Complex
{
 public:
  / / Конструктор за умовчанням
  / / (В даному випадку є також і конструктором перетворення)
  Complex (double i_re = 0, double i_im = 0)
     : Re (i_re), im (i_im)
  {}

  / / Конструктор копіювання
  Complex (const Complex & obj)
  {
   re = obj.re;
   im = obj.im;
  }
  private:
    double re, im;
};
```

Конструктор за умовчанням

Конструктор не має обов'язкових аргументів. Використовується при створенні масивів об'єктів, викликаючи для створення кожного екземпляра. За відсутності явно заданого конструктора за замовчуванням його код генерується компілятором (що на початковому тексті, природно, не відображається).

Конструктор копіювання

Конструктор, аргументом якого є посилання на об'єкт того ж класу. Застосовується в C + + для передачі об'єктів у функції за значенням.

Конструктор копіювання в основному необхідний, коли об'єкт має покажчики на об'єкти виділені в купі. Якщо програміст не створює конструктор копіювання, то компілятор створить неявний конструктор

копіювання, який копіює покажчики як є, то є фактичне копіювання даних не відбувається і два об'єкти посилаються на одні і ті ж дані в купі. Відповідно спроба зміни «копії» зашкодить оригінал, а виклик деструктора для одного з цих об'єктів при подальшому використанні іншого призведе до звернення в область пам'яті, вже не належить програмі.

Аргумент повинен передаватися саме по посиланню, а не за значенням. Це випливає з колізії: при передачі об'єкта за значенням (зокрема, для виклику конструктора) потрібно скопіювати об'єкт. Але для того, щоб скопіювати об'єкт, необхідно викликати конструктор копіювання.

Конструктор перетворення

Конструктор, що приймає один аргумент. Визначає перетворення типу свого аргументу в тип конструктора. Таке перетворення типу неявно застосовується тільки якщо воно унікальне.

Віртуальний конструктор

Конструктор не буває віртуальним в сенсі віртуального методу - для того, щоб механізм віртуальних методів працював, потрібно запустити конструктор, який автоматично налаштує таблицю віртуальних методів даного об'єкта.

«Віртуальними конструкторами» називають схожий, але інший механізм, присутній в деяких мовах - наприклад, він є в Delphi, але немає в C + + і Java. Цей механізм дозволяє створити об'єкт будь-якого заздалегідь невідомого класу за двох умов:

цей клас є нащадком якогось наперед заданого класу (в даному прикладі це клас TVehicle);

на всьому шляху успадкування від базового класу до створюваного ланцюжок перевизначення не обривалася. При перевизначенні віртуального методу синтаксис Delphi вимагає ключове слово overload, щоб стара і нова функції з різними сигнатурами могли співіснувати, override для перевизначення функції або reintroduce для завдання нової функції з тим же ім'ям - останнє неприпустимо.

```
type
  TVehicle = class
     constructor Create; virtual;
   end;

  TAutomobile = class (TVehicle)
     constructor Create; override;
   end;

  TMotorcycle = class (TVehicle)
```

```
    constructor Create; override;
  end;

  TMoped = class (TMotorcycle) / / обриваємо ланцюжок перевизначення - заводимо новий Create
    constructor Create (x: integer); reintroduce;
  end;
```

У мові вводиться так званий класовий тип (метакласи). Цей тип як значення може приймати назву будь-якого класу, похідного від TVehicle.

```
type
  CVehicle = class of TVehicle;
```

Такий механізм дозволяє створювати об'єкти будь-якого заздалегідь невідомого класу, похідного від TVehicle.

```
var
  cv: CVehicle;
  v: TVehicle;

cv: = TAutomobile;
v: = cv.Create;
```

Зауважте, що код

```
cv: = TMoped;
v: = cv.Create;
```

є некоректним - директива reintroduce розірвала ланцюжок перевизначення віртуального методу, і в дійсності буде викликаний конструктор TMotorcycle.Create (а значить, буде створений мотоцикл, а не мопед!)

12.4 Деструктори

Деструктор автоматично запускається кожного разу, коли програма знищує об'єкт. В наступних уроках ви дізнаєтеся, як створити списки об'єктів, які збільшуються або зменшуються в міру виконання програми. Щоб створити такі динамічні списки, ваша програма для зберігання об'єктів розподіляє пам'ять динамічно (що ви ще не навчилися робити). До теперішнього моменту ви можете створювати і знищувати об'єкти в процесі виконання програми. У таких випадках має сенс застосування деструкторів.

Кожна зі створених вами до сих пір програм створювала об'єкти на самому початку свого виконання, просто оголошуючи їх. При завершенні програм C + + знищував об'єкти. Якщо ви визначаєте деструктор всередині своєї програми, C + + буде автоматично викликати деструктор для кожного об'єкта, коли програма завершується (тобто коли об'єкти знищуються). Подібно конструктору, деструктор має таке ж ім'я, як і клас об'єкта. Однак у випадку деструктора ви випереджає його ім'я символом тильди (~), як показано нижче:

```
~ Class_name (void) / / ----------> вказує деструктор

{
/ / Оператори деструктора
}
```

На відміну від конструктора ви не можете передавати параметри деструктор. Наступна програма DESTRUCT.CPP визначає деструктор для класу employee:

```
void employee ::-employee (void)

{
  cout << "Знищення об'єкта для" << name << endl;
}
```

В даному випадку деструктор просто виводить на ваш екран повідомлення про те, що C + + знищує об'єкт. Коли програма завершується, C + + автоматично викликає деструктор для кожного об'єкта. Нижче наведена реалізація програми DESTRUCT.CPP:

```
# Include <iostream.h>

# Include <string.h>

class employee

{
public:
  employee (char *, long, float);
  ~ Employee (void);
  void show_employee (void);
  int change_salary (float);
  long get_id (void);
private:
  char name [64];
```

```
  long employee_id;
  float salary;
};

employee :: employee (char * name, long employee_id, float salary)

{
  strcpy (employee :: name, name);
  employee :: employee_id = employee_id;
  if (salary <50000.0) employee :: salary = salary;
  else / / Неприпустимий оклад
  employee :: salary в 0.0;
}

void employee ::-employee (void)

{
  cout << "Знищення об'єкта для" << name << endl;
}

void employee :: show_employee (void)

{
  cout << "Службовець:" << name << endl;
  cout << "Номер службовця:" << employee_id << endl;
  cout << "Оклад:" << salary << endl;
}

void main (void)

{
  employee worker ("Happy Jamsa", 101, 10101.0);
  worker.show_employee ();
}
```

Якщо ви відкомпілюєте і запустіть цю програму, на вашому екрані з'явиться наступний висновок:

```
C: \> DESTRUCT <ENTER>

Службовець: Happy Jamsa

Номер службовця: 101

Оклад: 10101
```

Знищення об'єкта для Happy Jamsa

Як бачите, програма автоматично викликає деструктор, без якого-небудь явного виклику функції деструктора. До теперішнього моменту вашим програмам, ймовірно, не потрібно використовувати деструктор. Однак, коли програми почнуть розподіляти пам'ять всередині об'єктів, ви виявите, що деструктор забезпечує зручний спосіб звільнення пам'яті при знищенні об'єкта.

Деструктори

Деструктор являє собою функцію, яку C + + автоматично запускає, коли він чи ваша програма знищує об'єкт. Деструкція має таке ж ім'я, як і клас об'єкта, а проте ви випереджає ім'я деструктора символом тильди (~), наприклад ~ employee. У своїй програмі ви визначаєте деструктор точно так само, як і будь-який інший метод класу.

12.5 Абстрактні класи

Багато класи схожі з класом employee тим, що в них можна дати розумне визначення віртуальним функцій. Однак, є й інші класи. Деякі, наприклад, клас shape, представляють абстрактне поняття (фігура), для якого не можна створити об'єкти. Клас shape набуває сенсу тільки як базовий клас в деякому похідному класі. Причиною є те, що неможливо дати осмислене визначення віртуальних функцій класу shape:

```
class shape {
        / / ...
      public:
        virtual void rotate (int) {error ("shape :: rotate");}
        virtual void draw () {error ("shape :: draw"):}
        / / Не можна ні обертати, ні малювати абстрактну фігуру
        / / ...
      };
```

Створення об'єкта типу shape (абстрактної фігури) законна, хоча абсолютно безглузда операція:

```
shape s; / / нісенітниця: `` фігура взагалі"
```

Вона безглузда тому, що будь-яка операція з об'єктом s призведе до помилки.

Краще віртуальні функції класу shape описати як чисто віртуальні. Зробити віртуальну функцію чисто віртуальної можна, додавши ініціалізатор = 0:

```
class shape {
  / / ...
public:
  virtual void rotate (int) = 0; / / чисто віртуальна функція
  virtual void draw () = 0; / / чисто віртуальна функція
};
```

Клас, в якому є віртуальні функції, називається абстрактним. Об'єкти такого класу створити не можна:

```
shape s; / / помилка: мінлива абстрактного класу shape
```

Абстрактний клас можна використовувати лише в якості базового для іншого класу:

```
class circle: public shape {
  int radius;
public:
  void rotate (int) {} / / нормально:
                / / Перевизначення shape :: rotate
  void draw (); / / нормально:
                / / Перевизначення shape :: draw

  circle (point p, int r);
};
```

Якщо чисто віртуальна функція не визначається в похідному класі, то вона і залишається такою, а значить похідний клас теж є абстрактним. При такому підході можна реалізовувати класи поетапно:

```
class X {
public:
  virtual void f () = 0;
  virtual void g () = 0;
};

X b; / / помилка: опис об'єкта абстрактного класу X

class Y: public X {
  void f (); / / перевизначення X :: f
};

Y b; / / помилка: опис об'єкта абстрактного класу Y
```

```
class Z: public Y {
  void g (); / / перевизначення X :: g
};

Z c; / / нормально
```

Абстрактні класи потрібні для завдання інтерфейсу без уточнення будь-яких конкретних деталей реалізації. Наприклад, в операційній системі деталі реалізації драйвера пристрою можна приховати таким абстрактним класом:

```
class character_device {
public:
  virtual int open () = 0;
  virtual int close (const char *) = 0;
  virtual int read (const char *, int) = 0;
  virtual int write (const char *, int) = 0;
  virtual int ioctl (int ...) = 0;
  / / ...
};
```

Справжні драйвери будуть визначатися як похідні від класу character_device.

Після введення абстрактного класу у нас є всі основні засоби для того, щоб написати закінчену програму.

12.6 Вложені класи

Описання класу може бути вкладеним. Наприклад:

```
class set {
    struct setmem {
      int mem;
      setmem * next;
      setmem (int m, setmem * n) {mem = m; next = n;}
    };
    setmem * first;
  public:
    set () {first = 0;}
    insert (int m) {first = new setmem (m, first);}
    / / ...
  };
```

Доступність вкладеного класу обмежується областю видимості лексично осяжний класу:

```
setmem m1 (1,0); / / помилка: setmem не знаходиться
                 / / В глобальній області видимості
```

Якщо тільки опис вкладеного класу не є зовсім простим, то краще описувати цей клас окремо, оскільки вкладені описи можуть стати дуже заплутаними:

```
class setmem {
friend class set; / / доступно тільки для членів set
  int mem;
  setmem * next;
  setmem (int m, setmem * n) {mem = m; next = n;}

  / / Багато інших корисних членів
};

class set {
  setmem * first;
public:
  set () {first = 0;}
  insert (int m) {first = new setmem (m, first);}
  / / ...
};
```

Корисна властивість вкладеності - це скорочення числа глобальних імен, а недолік його в тому, що воно порушує свободу використання вкладених типів.

Ім'я класу-члена (вкладеного класу) можна використовувати поза опису осяжний його класу так само, як ім'я будь-якого іншого члена:

```
class X {
    struct M1 {int m;};
  public:
    struct M2 {int m;};

    M1 f (M2);
};

void f ()
{M1 a; / / помилка: ім'я `M1 'поза області видимості
   M2 b; / / помилка: ім'я `M1 'поза області видимості
   X :: M1 c; / / помилка: X :: M1 приватний член
   X :: M2 d; / / нормально
}
```

Зазначимо, що контроль доступу відбувається і для імен вкладених класів.

У функції-члені область видимості класу починається після уточнення X :: і тягнеться до кінця опису функції. Наприклад:

```
M1 X :: f (M2 a) / / помилка: ім'я `M1 'поза області видимості
    {/ * ... * /}

  X :: M1 X :: f (M2 a) / / нормально
    {/ * ... * /}

  X :: M1 X :: f (X :: M2 a) / / нормально, але третє уточнення X :: зайво
    {/ * ... * /}
```

12.7 Наслідування

Спадкування являє собою здатність похідного класу успадковувати характеристики існуючого базового класу. Наприклад, припустимо, що у вас є базовий клас employee:

```
class employee

{
public:
  employee (char *, char *, float);
  void show_employee (void);
private:
  char name [64];
  char position [64];
  float salary;
};
```

Далі припустимо, що вашій програмі потрібно клас manager, який додає наступні елементи даних в клас employee:

```
float annual_bonus;
char company_car [64];
int stock_options;
```

В даному випадку ваша програма може вибрати два варіанти: по-перше, програма може створити новий клас manager, який дублює багато елементів класу employee, або програма може породити клас типу manager з базового класу employee. Породжуючи клас manager з існуючого класу employee, ви знижуєте обсяг необхідного програмування і виключаєте дублювання коду всередині вашої програми.

Для визначення цього класу ви повинні вказати ключове слово class, ім'я manager, наступне за ним двокрапка і ім'я employee, як показано нижче:

Похідний клас / / -----> class manager: public employee {<------- / / Базовий клас

/ / Тут визначаються елементи
};

Ключове слово public, яке передує ім'я класу employee, вказує, що загальні (public) елементи класу employee також є загальними і в класі manager. Наприклад, такі оператори породжують клас manager.

```
class manager: public employee

{
public:
  manager (char *, char *, char *, float, float, int);
  void show_manager (void);
private:
  float annual_bonus;
  char company_car [64];
  int stock_options;
};
```

Коли ви породжує клас з базового класу, приватні елементи базового класу доступні похідному класу тільки через інтерфейсні функції базового класу. Таким чином, похідний клас не може безпосередньо звернутися до приватних елементів базового класу, використовуючи оператор точку.

Наступна програма MGR_EMP.CPP ілюструє використання успадкування в C + +, створюючи клас manager з базового класу employee:

```
# Include <iostream.h>

# Include <string.h>

class employee

{
public:
  employee (char *, char *, float);
  void show_employee (void);
private:
  char name [64];
```

```
  char position [64];
  float salary;
};

employee :: employee (char * name, char * position, float salary)

{
  strcpy (employee :: name, name);
  strcpy (employee :: position, position);
  employee :: salary = salary;
}

void employee :: show_employee (void)

{
  cout << "Ім'я:" << name << endl;
  cout << "Посада:" << position << endl;
  cout << "Оклад: $" << salary << endl;
}

class manager: public employee

{
public:
  manager (char *, char *, char *, float, float, int);
  void show_manager (void);
private:
  float annual_bonus;
  char company_car [64];
  int stock_options;
};

manager :: manager (char * name, char * position, char * company_car, float salary, float bonus, int stock_options): employee (name, position, salary)

{
  strcpy (manager :: company_car, company_car);
  manager :: annual_bonus = bonus;
  manager :: stock_options = stock_options;
}

void manager :: show_manager (void)

{
  show_employee ();
```

```
  cout << "Машина фірми:" << company_car << endl;
  cout << "Щорічна премія: $" << annual_bonus << endl;
  cout << "Фондовий опціон:" << stock_options << endl;
}

void main (void)

{
  employee worker ("Джон Дой", "Програміст", 35000);
   manager boss ("Джейн Дой", "Віце-президент", "Lexus", 50000.0, 5000,
1000);
  worker.show_employee ();
  boss.show_manager ();
}
```

Як бачите, програма визначає базовий клас employee, а потім визначає похідний клас manager. Зверніть увагу на конструктор manager. Коли ви породжує клас з базового класу, конструктор похідного класу повинен викликати конструктор базового класу. Щоб викликати конструктор базового класу, помістіть двокрапку відразу ж після конструктора похідного класу, а потім вкажіть ім'я конструктора базового класу з необхідними параметрами:

```
manager :: manager (char * name, char * position, char * company_car, float
salary, float bonus, int stock_options):
employee (name, position, salary) / / ---------- Конструктор базового класу

{
strcpy (manager :: company_car, company_car);
manager :: annual_bonus = bonus;
manager :: stock_options = stock_options;
}
```

Також зверніть увагу, що функція show_manager викликає функцію show_employee, яка є елементом класу employee. Оскільки клас manager є похідним класу employee, клас manager може звертатися до загальних елементів класу employee, як якщо б всі ці елементи були визначені усередині класу manager,

Подання про спадкування

Спадкування являє собою здатність похідного класу успадковувати характеристики існуючого базового класу. Простими словами це означає, що, якщо у вас є клас, чиї елементи даних або функції-елементи можуть бути використані новим класом, ви можете побудувати цей новий клас в термінах існуючого (або базового) класу. Новий клас в свою чергу буде

успадковувати елементи (характеристики) існуючого класу. Використання успадкування для побудови нових класів заощадить вам багато часу і сили на програмування. Об'єктно-орієнтоване програмування широко використовує спадкування, дозволяючи вашій програмі будувати складні об'єкти з невеликих легко керованих об'єктів.

12.8 Статичні члени класів

Клас - це тип, а не якийсь дане, і для кожного об'єкта класу створюється своя копія членів, що представляють дані. Однак, найбільш вдала реалізація деяких типів вимагає, щоб всі об'єкти цього типу мали деякі загальні дані. Краще, якщо ці дані можна описати як частина класу. Наприклад, в операційних системах або при моделюванні управління завданнями часто потрібен список завдань:

```
class task {
        / / ...
        static task * chain;
        / / ...
      };
```

Описавши член chain як статичний, ми отримуємо гарантію, що він буде створений в однині, тобто не буде створюватися для кожного об'єкта task. Але він знаходиться в області видимості класу task, і може бути доступний поза цій галузі, якщо тільки описаний в загальній частині. В цьому випадку ім'я члена має уточнюватися ім'ям класу:

```
if (task :: chain == 0) / / якісь оператори
```

У функції-члені його можна позначати просто chain. Використання статичних членів класу може помітно скоротити потребу в глобальних змінних.

Описуючи член як статичний, ми обмежуємо його область видимості і робимо його незалежним від окремих об'єктів його класу. Це властивість корисно як для функцій-членів, так і для членів, що представляють дані:

```
class task {
        / / ...
        static task * task_chain;
        static void shedule (int);
        / / ...
      };
```

Але опис статичного члена - це тільки опис, і десь в програмі має бути єдине визначення для описуваного об'єкта або функції, наприклад, таке:

```
task * task :: task_chain = 0;
        void task :: shedule (int p) {/ * ... * /}
```

Природно, що й приватні члени можуть визначатися таким чином.

Зазначимо, що службове слово static не потрібно і навіть не можна використовувати у визначенні статичного члена класу. Якби воно було присутнє, виникла б неоднозначність: вказує воно на те, що член класу є статичним, або використовується для опису глобального об'єкта або функції? Слово static одне з перевантажених службових слів в C і C + +. До статичному члену, який представляє дані, відносяться обидва основні його значення: "статично розміщується", тобто протилежний об'єктів, розміщених в стеку або вільної пам'яті, і "статичний" в сенсі з обмеженою областю видимості, тобто протилежний об'єктів, що підлягають зовнішньому зв'язування. До функцій-членам належить тільки друге значення static.

12.9 Вказівники на члени класів

Можна брати адресу члена класу. Операція взяття адреси функції-члена часто виявляється корисною, оскільки цілі і способи застосування покажчиків на функції, про які ми говорили в $ $ 4.6.9, в рівній мірі відносяться і до таких функцій. Покажчик на член можна отримати, застосувавши операцію взяття адреси & до повністю уточненого імені члена класу, наприклад, & class_name :: member_name. Щоб описати змінну типу "покажчик на член класу X", треба використовувати описувач виду X :: *. Наприклад:

```
# Include <iostream.h>

        struct cl
        {
          char * val;
          void print (int x) {cout << val << x << '\ n';}
          cl (char * v) {val = v;}
        };
```

Покажчик на член можна описати і використовувати так:

```
  typedef void (cl :: * PMFI) (int);

       int main ()
       {
```

```
    cl z1 ("z1");
    cl z2 ("z2");
    cl * p = &z2;
    PMFI pf = & cl :: print;
    z1.print (1);
    (Z1. * pf) (2);
    z2.print (3);
    (P-> * pf) (4);
  }
```

Використання typedef для заміни важко сприймається описувача в С досить типовий випадок. Операції. * I -> * налаштовують покажчик на конкретний об'єкт, видаючи в результаті функцію, яку можна викликати. Пріоритет операції () вище, ніж у операцій. * I -> *, тому потрібні дужки.

У багатьох випадках віртуальні функції успішно замінюють покажчики на функції.

12.10 Ініціалізація даних членів класу

Нехай клас містить в собі члени абстрактних типів. Особливістю їх ініціалізації є те, що вона виконується за допомогою відповідного конструктора. Розглянемо клас

```
class coord {double x, y, z;
    public:
    coord () {x = y = z = 0;}
    coord (double xv, double yv, double zv = 0) {x = xv; y = yv; z = zv;}
    coord (coord & c) {x = c.x; y = c.y; z = c.z;}
    };
    class triang {
    coord vert1, vert2, vert3; / / Координати вершин трикутника.
    public:
    triang ();
    triang (coord & v1, coord & v2, coord & v3);
    };
```

При ініціалізації деякого об'єкту класу triang потрібно три рази викликати конструктори для його вершин - об'єктів типу coord. Для цього у визначенні конструктора класу triang після двокрапки потрібно помістити список звернень до конструкторам класу coord:

```
Traing :: triang (coord & v1, coord & v2, coord & v3):
    vert1 (v1), vert2 (v2), vert3 (v3) {. . .}
```

Виклик конструкторів класу coord відбувається до виконання тіла самого конструктора класу triang. Порядок їх виклику визначається порядком появи оголошень членів типу coord при створенні класу triang.

Клас coord містить конструктор без аргументів. Замість запису при зверненні до такого конструктору

```
triang :: triang (): vert1 (), vert2 (), vert3 () {. . .}
```

допускається написати просто так:

```
triang :: triang () {. . .}
```

Ініціалізація констант

Якщо серед даних-членів класу є члени, описані з модифікатором const, то при ініціалізації використовується та ж форма запису конструктора, що й у випадку з даними абстрактних типів:

```
class cl {int v;
    const c;
    public:
    cl (int a, int b): c (b) {v = a;}
    };
```

Константу можна ініціалізувати тільки в конструкторі, спроба зробити це будь-яким іншим способом (наприклад, за допомогою іншої компонентної функції) призведе до повідомлення про помилку. Ініціалізація констант в тілі конструктора теж неприпустима.

Зауважимо, що спосіб запису конструктора, обов'язковий для констант і даних абстрактних типів, можна використовувати і для звичайних членів класу:

```
class ro {int var; const c;
    public:
    ro (int v, int u): c (u), var (v) {}
    };
```

12.11 Перевантаження операторів

Як ви вже знаєте, тип змінної визначає набір значень, які вона може зберігати, а також набір операцій, які можна виконувати над цієї змінної. Наприклад, над значенням змінної типу int ваша програма може виконувати додавання, віднімання, множення і ділення. З іншого боку, використання

оператора плюс для складання двох рядків позбавлене всякого сенсу. Коли ви визначаєте в своїй програмі клас, то по суті ви визначаєте новий тип. А якщо так, C + + дозволяє вам визначити операції, що відповідають цьому новому типу.

Перевантаження оператора полягає в зміні сенсу оператора (наприклад, оператора плюс (+), який зазвичай в C + + використовується для складання) при використанні його з певним класом. У даному уроці ви визначите клас string, увійдіть оператори плюс і мінус. Для об'єктів типу string оператор плюс буде додавати зазначені символи до поточного вмісту рядка. Подібним чином оператор мінус буде видаляти кожне входження зазначеного символу з рядка. До кінця цього уроку ви вивчите такі основні концепції:

Ви перевантажуєте оператори для поліпшення зручності читання ваших програм, але перевантажувати оператори слід тільки в тому випадку, якщо це спрощує розуміння вашої програми.

Для перевантаження операторів програми використовують ключове слово C + + operator.

Перевизначаючи оператор, ви вказуєте функцію, яку C + + викликає кожен раз, коли клас використовує перевантажений оператор. Ця функція, в свою чергу, виконує відповідну операцію.

Якщо ваша програма перевантажує оператор для певного класу, то зміст цього оператора змінюється тільки для зазначеного класу, що залишилася частина програми буде продовжувати використовувати цей оператор для виконання його стандартних операцій.

C + + дозволяє перевантажувати більшість операторів, за винятком чотирьох, перерахованих в таблиці 24, які програми не можуть перевантажувати.

Перевантаження операторів може спростити найбільш загальні операції класу і поліпшити читаність програми. Знайдіть час для експерименту з програмами, представленими в цьому уроці, і ви виявите, що перевантаження операторів виконується дуже просто.

Перевантаження операторів плюс і мінус

Коли ви перевантажуєте оператор для будь-якого класу, то сенс даного оператора не змінюється для змінних інших типів. Наприклад, якщо ви перевантажуєте оператор плюс для класу string, то зміст цього оператора не змінюється, якщо необхідно скласти два числа. Коли компілятор C + + зустрічає в програмі оператор, то на підставі типу змінної він визначає ту операцію, яка повинна бути виконана.

Нижче наведено визначення класу, що створює клас string. Цей клас містить один елемент даних, який представляє собою власне символьну рядок. Крім того, цей клас містить кілька різних методів і поки не визначає будь-яких операторів:

```
class string
```

```
{
public:
  string (char *); / / Конструктор
  void str_append (char *);
  void chr_minus (char);
  void show_string (void);
private:
  char data [256];
};
```

Як бачите, клас визначає функцію str_append, яка додає вказані символи до вмісту рядка класу. Аналогічним чином функція chr_minus - видаляє кожне входження зазначеного символу з рядка класу. Наступна програма STRCLASS.CPP використовує клас string для створення двох об'єктів символьних рядків і маніпулювання ними.

```
# Include <iostream.h>

# Include <string.h>

class string

{
public:
  string (char *); / / Конструктор
  void str_append (char *);
  void chr_minus (char);
  void show_string (void);
private:
  char data [256];
};

string :: string (char * str)

{
  strcpy (data, str);
}

void string :: str_append (char * str)

{
  strcat (data, str);
}
```

```
void string :: chr_minus (char letter)

{
  char temp [256];
  int i, j;
  for (i = 0, j = 0; data [i]; i + +) / / Цю букву необхідно видалити?
  if (data [i]! = letter) / / Якщо немає, привласнити її temp
  temp [j + +] = data [i];
  temp [j] = NULL; / / Кінець temp
/ / Копіювати вміст temp назад в data
  strcpy (data, temp);
}

void string :: show_string (void)

{
  cout << data << endl;
}

void main (void)

{
  string title ("Вчимося програмувати на мові C + +");
  string lesson ("Перевантаження операторів");
  title.show_string ();
  title.str_append ("я вчуся!");
  itle.show_string ();
  lesson.show_string ();
  lesson.chr_minus ('p');
  lesson.show_string ();
  }
```

Як бачите, програма використовує функцію str_append для додавання символів до строкової змінної title. Програма також використовує функцію chr_minus для видалення кожної букви "p" з символьного рядка lesson. В даному випадку програма використовує виклики функції для виконання цих операцій. Однак, використовуючи перевантаження операторів, програма може виконувати ідентичні операції з допомогою операторів плюс (+) і мінус (-).

При перевантаженні оператора використовуйте ключове слово C + + operator разом з прототипом і визначенням функції, щоб повідомити компілятору C + +, що клас буде використовувати цей метод як оператор. Наприклад, таке визначення класу використовує ключове слово operator, щоб призначити оператори плюс і мінус функцій str_append і chr_minus всередині класу string:

```
class string

{
public:
  string (char *); / / Конструктор
  void operator + (char *);
  void operator - (char); ----- Визначення операторів класу void show_string (void);
private:
  char data [256];
};
```

Як бачите, клас перевантажує оператори плюс і мінус. Як уже згадувалося, коли клас перевантажує оператор, він повинен вказати функцію, яка реалізує операцію, відповідає цьому оператору. У разі оператора плюс визначення такої функції стає наступним:

```
void string :: operator + (char * str)

{
  strcat (data, str);
}
```

Як бачите, визначення цієї функції не містить імені, оскільки тут визначається перевантажений оператор класу. Для перевантаження оператора плюс програма не змінила обробку, яка здійснюється всередині функції (код цієї функції ідентичний коду попередньої функції str_append). Замість цього програма просто замінила ім'я функції ключовим словом operators відповідним оператором. Наступна програма OPOVERLD.CPP ілюструє використання перевантажуються операторів плюс і мінус:

```
# Include <iostream.h>

# Include <string.h>

class string

{
public:
  string (char *); / / Конструктор
  void operator + (char *);
  void operator - (char);
  void show_string (void);
private;
```

```
  char data [256];
};

string :: string (char * str)

{
  strcpy (data, str);
}

void string :: operator + (char * str)

{
  strcat (data, str);
}

void string :: operator - (char letter)

{
  char temp [256];
  int i, j;
  for (i = 0, j = 0; data [i]; i + +) if (data [il 1 = letter) temp [j + +] = data [i];
  temp [j] = NULL;
  strcpy (data, temp);
}

void string :: show_string (void)

{
  cout << data << endl;
}

void main (void)

{
  string title ("Вчимося програмувати на C + +");
  string lesson ("Перевантаження операторів");
  title.show_string ();
  title + "я вчуся!";
  title.show_string ();
  lesson.show_string ();
  lesson - 'P';
  lesson.show_string ();
}
```

Як бачите, програма використовує перевантажені оператори:

title + "я вчуся!"; / / Додати текст "я вчуся!"

lesson - 'p'; / / Видалити букву 'p'

В даному випадку синтаксис оператора законний, але трохи незвичний. Зазвичай ви використовуєте оператор плюс у виразі, яке повертає результат, наприклад, як в операторі some_str = title + "текст";. Коли ви визначаєте оператор, C + + надає вам повну свободу відносно поведінки оператора. Однак, як ви пам'ятаєте, ваша мета при перевантаженні операторів полягає в тому, щоб спростити розуміння ваших програм. Тому наступна програма STR_OVER.CPP трохи змінює попередню програму, щоб дозволити їй виконувати операції над змінними типу string, використовуючи синтаксис, який більш узгоджується зі стандартними операторами присвоювання:

```
# Include <iostream.h>

# Include <string.h>

class string

{
public:
  string (char *); / / Конструктор
  char * operator + (char *);
  char * operator - (char);
  void show_string (void);
private:
  char data [256];
};

string :: string (char * str)

{
  strcpy (data, str);
}

char * string :: operator + (char * str)

{
  return (strcat (data, str));
}

char * string :: operator - (char letter)
```

```
{
  char temp [256];
  int i, j;
  for (i = 0, j = 0; data [i]; i + +) if (data [i] 1 = letter) temp [j + +] = data [i];
  temp [j] = NULL;
  return (strcpy (data, temp));
}

void string :: show_string (void)

{
  cout << data << endl;
}

void main (void)

{
  string title ("Вчимося програмувати на C + +");
  string lesson ("Перевантаження операторів");
  title.show_string ();
  title = title + "я вчуся";
  title.show_string ();
  lesson.show_string ();
  lesson = lesson - '?';
  lesson.show_string ();
}
```

Змінивши перевантажені оператори плюс і мінус таким чином, щоб вони повертали покажчик на символьний рядок, програма може тепер використовувати ці оператори в звичному для оператора присвоювання вигляді:

```
title = title + "вчимося програмувати!";

lesson = lesson - 'p';
```

Другий приклад

При створенні ваших власних типів даних за допомогою класів найбільш загальною операцією буде перевірка, чи є два об'єкти однаковими. Використовуючи перевантаження, ваші програми можуть перевантажити оператори рівності (==), нерівності (! =) Або інші оператори порівняння. Наступна програма COMP_STR.CPP додає новий оператор в клас string, який перевіряє, чи рівні два об'єкти string. Використовуючи перевантаження операторів, ваші програми можуть перевіряти, чи містять рядкові об'єкти однакові рядки, як показано нижче:

if (some_string == another_string)

Нижче наведена реалізація програми COMP_STR.CPP:

```
# Include <iostream.h>

# Include <string.h>

class string

{
public:
  string (char *); / / конструктор
  char * operator + (char *);
  char * operator - (char);
  int operator == (string);
  void show_string (void);
private:
  char data [256];
};

string :: string (char * str)

{
  strcpy (data, str);
}

char * string :: operator + (char * str)

{
  return (strcat (data, str));
}

char * string :: operator - (char letter)

{
  char temp [256];
  int i, j;
  for (i = 0, j = 0; data [i]; i + +) if (data [i] 1 = letter) temp [j + +] = data [i];
  temp [j] = NULL;
  return (strcpy (data, temp));
}

int string :: operator == (string str)
```

```
{
  int i;
  for (i = 0; data [i] == str.data [i]; i + +)
  if ((data [i] == NULL) && (str.data [i] == NULL)) return (1); / / Так само
  return (0); / / Не одно
}

void string :: show_string (void)

{
  cout << data << endl;
}

void main (void)

{
  string title ("Вчимося програмувати на C + +");
  string lesson ("Перевантаження операторів");
  string str ("Вчимося програмувати на C + +");
  if (title == lesson) cout << "title i lesson рівні" << endl;
  if (str == lesson) cout << "str i lesson рівні" << endl;
  if (title == str) cout << "title i str рівні" << endl;
}
```

Як бачите, перевантажуючи оператори подібним чином, ви спрощуєте розуміння ваших програм.

Оператори які ви не можете перевантажувати

У загальному випадку ваші програми можуть перевантажити майже всі оператори C + +.

Оператор	Призначення	Приклад
.	Выбор элемента	object.member
.*	Указатель на элемент	object.*member
::	Разрешение области видимости	classname::member
?:	Условный оператор сравнения	c = (a > b) ? a : b;

12.12 Залікове заняття

1. Що визначає клас? Чим обличается клас від об'єкта?
2. Чи можна оголошувати масив об'єктів? А масив класів?
3. Чи дозволяється оголошувати покажчик на об'єкт? А покажчик на клас?
4. Чи допускається передавати об'єкти в якості параметрів, і якими способами? А повертати як результат?
5. Як називається використання об'єкта одного класу в якості поля іншого класу?
6. Чи є структура класом? Чим клас відрізняється від структури?
7. Які ключові слова в C + + позначають клас?
8. Поясніть принцип інкапсуляції.
9. Що таке композиція?
10. Для чого використовуються ключові слова public і private?
11. Чи можна використовувати ключові слова public і private в структурі?
12. Чи існують обмеження на використання public і private в класі? А в структурі?
13. Чи обов'язково робити поля класу приватними?
14. Що таке метод? Як викликається метод?
15. Чи може метод бути приватний?
16. Як визначити метод безпосередньо всередині класу? А поза класом? Чим ці визначення відрізняються?
17. Можна в методах привласнювати параметрам значення за замовчуванням?
18. Що позначається ключовим словом this?
19. Навіщо потрібні константні методи? Чим відрізняється визначення константного методу від звичайного?
20. Чи може константних метод викликатися для об'єктів-змінних? А звичайний метод - для об'єктів-констант?
21. Поясніть принцип поліморфізму.
22. Скільки місця в пам'яті займає об'єкт класу? Як це дізнатися?
23. Який розмір «порожнього» об'єкта?
24. Чи впливають методи на розмір об'єкта?
25. Однаковий чи розмір класу та аналогічної структури?
26. Які операції не можна перевантажувати? Як ви думаєте, чому?
27. Чи можна перевантажувати операції для вбудованих типів даних?
28. Чи можна при перевантаженні змінити пріоритет операції?
29. Чи можна визначити нову операцію?
30. Перерахуйте особливості перевантаження операцій як методів класу. Чим відрізняється перевантаження зовнішнім чином від перевантаження як методу класу?
31. Який результат повинні повертати операції з привласненням?

32. Як розрізняються перевантажена префіксная і постфіксной операції інкремента і декремента?

33. Що означає вираз * this? У яких випадках воно використовується?

34. Які операції не рекомендується перевантажувати як методи класу? Чому?

35. Які операції дозволяється перевантажувати тільки як методи класу?

36. Дайте визначення дружньої функції. Як оголошується дружня функція? А як визначається?

37. Дайте визначення конструктора. Яке призначення конструктора? Перерахуйте відмінності конструктора від методу.

38. Скільки конструкторів може бути в класі? Чи допускається перевантаження конструкторів? Які види конструкторів створюються за замовчуванням?

39. Чи може конструктор бути приватним? Які наслідки тягне за собою оголошення конструктора приватним?

40. Наведіть кілька випадків, коли конструктор викликається неявно.

41. Як проініціалізувати динамічну змінну?

42. Як оголосити константу в класі? Чи можна оголосити дробову константу?

43. Яким чином вирішується ініціалізувати константні поля в класі?

44. В якому порядку ініціалізувалися поля в класі? Чи збігається цей порядок з порядком перерахування ініціалізаторів в списку ініціалізації конструктора?

45. Які конструкції С + + дозволяється використовувати в списку ініціалізації як ініціюючих виразів?

46. Який вид конструктора фактично є конструктором перетворення типів?

47. Для чого потрібні функції перетворення? Як оголосити таку функцію в класі?

48. Як заборонити неявне перетворення типу, що виконується конструктором ініціалізації?

49. Які проблеми можуть виникнути при визначенні функцій перетворення?

50. Для чого служить ключове слово explicit?

51. Чи впливає наявність цілочисельних констант-полів на розмір класу?

52. Чи дозволяється оголошувати масив в якості поля класу. Як присвоїти елементів масиву початкові значення?

53. Скільки операндів має операція індексування []? Який вид результату повинна повертати ця операція?

54. Для чого потрібні статичні поля в класі? Як вони визначаються?

55. Як оголосити в класі і проініціалізувати статичний константний масив?

56. Що таке вирівнювання і від чого воно залежить? Чи впливає вирівнювання на розмір класу?

57. Дайте визначення контейнера.

58. Які види вбудованих контейнерів в C + + ви знаєте?

59. Які види доступу до елементів контейнера вам відомі?

60. Чим відрізняється прямий доступ від асоціативного?

61. Перелічіть операції, які зазвичай реалізуються для послідовного доступу до елементів контейнера.

62. Дайте визначення ітератора.

63. Чи можна реалізувати послідовний доступ без ітератора? У чому переваги реалізації послідовного доступу за допомогою ітератора?

64. Що грає роль ітератора для масивів C + +?

65. Що таке деструктор? Чи може деструктор мати параметри?

66. Чому для класів-контейнерів деструктор треба писати явним чином?

67. Чи допускається перевантаження деструкторів?

68. Що таке «глибоке копіювання» і коли в ньому виникає необхідність?

69. Яке копіювання здійснює стандартний конструктор копіювання?

70. Чим відрізняється копіювання від присвоювання?

71. Поясніть, чому в операції привласнення потрібна перевірка присвоювання самому собі?

72. Чи можна в якості операції індексування використовувати операцію виклику функції ()? У чому її переваги перед операцією []?

73. Чому необхідно писати два визначення операції індексування? Чим вони відрізняються?

74. Дайте визначення вкладеного класу.

75. Чи можна клас-ітератор реалізувати як зовнішній клас? А як вкладений? У чому відмінності цих методів реалізації?

76. Чи може осяжний клас мати необмежений доступ до елементів вкладеного класу? А вкладений клас - до елементів осяжний?

77. Обмежена чи глибина вкладеності класів?

78. Чи можна визначити вкладений клас зовнішнім образом? Навіщо це може знадобитися?

79. Яким чином вкладений клас може використовувати методи осяжний класу? А осяжний - методи вкладеного?

80. Що таке «позамежний» елемент, яку роль він грає в контейнерах?

81. Поясніть, з яких причин важко написати універсальний контейнер, елементи якого можуть мати довільний тип.

82. Назвіть ключові слова C + +, які використовуються для обробки винятків.

83. Виняток - це:

1) подія;

2) ситуація;

3) об'єкт;

4) помилка в програмі;

5) переривання;

84. Яким чином виключення генерується?

85. Які функції контрольованого блоку?

86. Що позначається ключовим словом catch?

1) контрольований блок;
2) блок обробки виключення;
3) секція-пастка;
4) генератор виключення;
5) обробник переривання;
87. Якого типу може бути виняток?
88. Скільки параметрів дозволяється писати в заголовку секції-пастки?
89. Якими способами дозволяється передавати виключення в блок обробки?
90. Поясніть, яким чином подолати обмеження на передачу єдиного параметра в блок обробки.
91. Чому не можна виконувати перетворення типів виключень при передачі в секцію-пастку?
92. Напишіть конструкцію, яка дозволяє перехопити будь виняток.
93. Чи можуть контрольовані блоки бути вкладеними?
94. Навіщо потрібен «контрольований блок-функція» і чим він відрізняється від звичайного контрольованого блоку?
95. Перерахуйте можливі способи виходу з блоку обробки.
96. Яким чином виключення «передати далі»?
97. Скільки секцій-пасток має бути задано в контрольованому блоці?
98. Що таке «специфікація винятків»?
99. Що відбувається, якщо функція порушує специфікацію винятків?
100. Чи враховується специфікація винятків при перевантаженні функцій?
101. Що таке «ієрархія винятків»?
102. Чи існують стандартні винятки? Назвіть два-три типи стандартних винятків.
103. Поясніть «взаємовідношення» виключень і деструкторів.
104. Поясніть, навіщо може знадобитися підміна стандартних функцій завершення.
105. Які види нестандартних винятків ви знаєте?
106. У чому відмінність механізму структурної обробки виключень Windows від стандартного механізму?
107. Які дві ролі виконує спадкування?
108. Які види спадкування можливі в C + +?
109. Чим відрізняється модифікатор доступу protected від модифікаторів private і public?
110. Чим відкрите спадкування відрізняється від закритого і захищеного?
111. Які функції не успадковуються?
112. Сформулюйте правила написання конструкторів в похідному класі.
113. Який порядок виклику конструкторів? А деструкторів?
114. Чи можна в похідному класі оголошувати нові поля? А методи?
115. Якщо ім'я нового поля збігається з ім'ям успадкованого, то яким чином вирішити конфлікт імен?

116. Що відбувається, якщо ім'я методу-спадкоємця збігається з ім'ям базового методу?

117. Сформулюйте принцип підстановки.

118. Коли виконується понижувальний приведення типів?

119. Поясніть, що таке «зрізання» або «розщеплення».

120. Поясніть, навіщо потрібні віртуальні функції.

121. Що таке зв'язування?

122. Чим «раннє» зв'язування відрізняється від «пізнього»?

123. Які два види поліморфізму реалізовані в C + +?

124. Дайте визначення поліморфного класу.

125. Чи може віртуальна функція бути дружньою функцією класу?

126. Успадковуються чи віртуальні функції?

127. Які особливості виклику віртуальних функцій в конструктори і деструктори?

128. Чи можна зробити віртуальної перевантажену операцію, наприклад, складання?

129. Чи може конструктор бути віртуальним? А деструктор?

130. Як віртуальні функції впливають на розмір класу?

131. Як оголошується «чиста» віртуальна функція?

132. Дайте визначення абстрактного класу.

133. Успадковуються чи чисті віртуальні функції?

134. Чи можна оголосити деструктор чисто віртуальним?

135. Чим відрізняється чистий віртуальний деструктор від чистого віртуальної функції?

136. Навіщо потрібна визначення чистого віртуального деструктора?

137. Успадковується чи визначення чистої віртуальної функції?

138. Наведіть класифікацію цілей успадкування.

139. Поясніть різницю успадкування інтерфейсу від спадкування реалізації.

140. Назвіть причини, що вимагають поділу програм на частини.

141. Дайте визначення терміна «одиниця трансляції»?

142. Чим відрізняється файл з вихідним текстом від одиниці трансляції?

143. Чи існують в C + + конструкції, що дозволяють ідентифікувати окремий модуль?

144. Які способи збирання програми ви можете назвати?

145. Що таке «об'єктний модуль»? Програма, яка «збирає» об'єктні модулі в програму, називається _____?

146. У чому полягає відмінність аргументу «файл» від <файл> в директиві # include?

147. Що таке ODR?

148. Поясніть, що таке «страж» включення і навіщо він потрібний.

149. Чи є інтерфейс класу його визначенням?

150. Скільки визначень класу може бути в одиниці трансляції?

151. Скільки визначень класу може бути в багатофайлову програмі?

152. Чим відрізняються стандартні заголовки <string>, <string.h> і <cstring>?

153. Поясніть суть ідіоми Pimpl.

154. Що таке делегування і як його можна використовувати для підвищення ступеня інкапсуляції?

155. Яким чином глобальну змінну, визначену в одній одиниці трансляції, зробити доступною в інший одиниці трансляції? А константу?

156. Чи можна використовувати слово extern при оголошенні функцій?

157. Як локалізувати оголошення функції в файлі?

158. Чим відрізняється «зовнішнє» зв'язування від «внутрішнього» зв'язування?

159. Що таке «специфікації компонування»?

160. Які об'єкти мають внутрішнім зв'язуванням за замовчуванням?

161. Які області видимості імен ви знаєте?

162. Для чого використовуються простору імен?

163. Чим відрізняються іменовані і неіменовані простору імен?

164. Чи можуть простору імен бути вкладеними?

165. Для чого застосовуються аліаси простору імен?

166. Як зробити члени простору імен доступними в декількох (в межі - у всіх) файлах програмного проекту?

167. Поясніть різницю між статичної та динамічної ініціалізацією.

168. В чому полягає проблема ініціалізації глобальних статичних змінних?

169. Які елементи класу можна оголошувати статичними?

170. Чи можна оголосити в класі статичну константу? А константний статичний масив?

171. А які статичні поля можна ініціалізувати безпосередньо в класі?

172. Як визначаються статичні поля? В який момент роботи програми виконується ініціалізація статичних полів?

173. Скільки місця в класі займають статичні поля?

174. Чим відрізняється статичний метод від звичайного?

175. Які методи класу не можуть бути статичними?

176. Які застосування статичних полів ви можете навести? А яким чином застосовуються статичні методи?

177. Наведіть структуру і принцип дії патерну Singleton.

178. Для чого призначені шаблони?

179. Які види шаблонів в C + + ви знаєте?

180. Поясніть термін «інстанціроване шаблону».

181. У чому різниця між визначенням і оголошенням шаблону?

182. Поясніть призначення ключового слова typename.

183. Які види параметрів дозволяється ставити в шаблоні класу? А в шаблоні функції?

184. Чи можна параметрам шаблону присвоювати значення за замовчуванням?

185. Чи може параметром шаблона бути інший шаблон? Які особливості оголошення параметра-шаблону?

186. Що таке спеціалізація шаблону? Поясніть різницю між повною і частковою спеціалізацією.

187. Чи дозволяється спеціалізувати шаблон функції?

188. Чи може клас-шаблон бути вкладеним в інший клас-шаблон? А в звичайний клас?

189. Чи можна оголосити в класі шаблонний метод? А шаблонний конструктор?

190. Чи можна перевантажувати функцію-шаблон?

191. Які параметри функції-шаблону виводяться автоматично?

192. Чи може шаблон класу бути спадкоємцем звичайного класу? А звичайний клас від шаблону?

193. Поясніть, що таке клас властивостей (клас трактувань).

194. Яким чином можна використовувати можливість успадкування звичайного класу від шаблону?

195. Чи може шаблонний конструктор бути конструктором за умовчанням?

196. Для чого застосовуються директиви явного інстанціювання?

197. Поясніть, в чому полягають проблеми, що виникають при поділі шаблонного класу на інтерфейс і реалізацію?

198. Що таке «модель явного інстанціювання» і як вона працює?

199. Чи може шаблонний клас мати «друзів»?

200. Які проблеми виникають при оголошенні дружньої функції для класу-шаблону?

201. Чи дозволяється визначати в класі-шаблоні статичні поля? А статичні методи?

202. Що таке «ініціалізація нулем»?

203. Що є одиницею пам'яті в C + +? Які вимоги до розміру одиниці пам'яті прописані в стандарті C + +?

204. У яких одиницях видає результат операція sizeof? Які типи даних мають розмір 1?

205. Які три види пам'яті входять в модель пам'яті C + +?

206. Скільки видів динамічної пам'яті забезпечує C + +?

207. Які функції для роботи з динамічною пам'яттю дісталися C + + у спадок від C? У яку бібліотеку вони включені?

208. Які функції виділяють пам'ять, і за допомогою яких функцій пам'ять звільняється?

209. Яке важлива відмінність має функція calloc () від функції malloc ()?

210. Які дії виконують функції виділення пам'яті, якщо пам'ять не може бути виділена?

211. Чи залежить обсяг виділеної пам'яті від типу покажчика? Чи впливає вирівнювання на обсяг виділеної динамічної пам'яті?

212. Чи можна за допомогою функції realloc () зменшити обсяг виділеної пам'яті?

213. Що станеться, якщо функції free () передати в якості аргументу нульової покажчик?

214. У чому головна відмінність об'єктно-орієнтованого механізму new / delete від механізму malloc () / free ()?

215. Скільки існує форм new / delete? У чому їхня відмінність?

216. Які типи є POD-типами? Чим відрізняється робота механізму new / delete з POD-об'єктами і nonPOD-об'єктами?

217. Які функції виконує обробник new?

218. Чи можна реалізувати власний обробник new і «причепити» його до механізму new / delete?

219. У чому головна відмінність об'єднання від інших видів класів C + +?

220. Чи може об'єднання брати участь в ієрархії успадкування?

221. Чи дозволяється визначати для об'єднання конструктори і деструктор? А віртуальні функції?

222. У чому схожі і чим відрізняються об'єднання і розміщує new?

223. Поясніть, чому при використанні розміщуючої new потрібно явно викликати деструктор?

224. Навіщо потрібні інтелектуальні покажчики?

225. Що таке «стратегія володіння»? Скільки стратегій володіння ви знаєте?

226. Який інтелектуальний покажчик реалізований в стандартній бібліотеці STL, і яку стратегію володіння він реалізує?

227. Поясніть, в чому переваги і недоліки інтелектуальних покажчиків з лічильником посилань.

228. Чи дозволяється перевантажувати new і delete і якими способами?

229. Опишіть схему функції, перевантажують глобальну функцію new.

230. Чи відрізняється реалізація перевантаженій функції new [] () для масивів від реалізації «звичайної» функції new ()?

231. Як ви думаєте, чому функції new / delete, перевантажуються для класу, є статичними?

232. Навіщо при перевантаженні new / delete для класу потрібно перевіряти розмір запитуваної пам'яті?

233. Поясніть, чим визначається «динамічність» контейнерів?

234. Що таке «стратегія розподілу пам'яті», і які стратегії виділення пам'яті ви знаєте?

235. Розгляньте наступну стратегію розподілу пам'яті: пам'ять виділяється для декількох елементів блоками фіксованої довжини, але блоки зв'язуються в список. Для якого виду контейнера можна використовувати таку стратегію?

236. Які операції можна перевантажити для доступу до елементів двовимірного масиву?

237. У чому полягають складнощі використання операції індексування [] для доступу до елементів двовимірного масиву?

238. Які способи реалізації операцій з контейнерами?

239. Яку конструкцію можна назвати «узагальнений алгоритм»?

240. Яким чином оголосити покажчик на метод?

241. Поясніть різницю між покажчиком на функцію і покажчиком на метод.

242. Яким чином отримати адресу методу?

243. Чи можна вказівником на функцію привласнювати адреса методу?

244. Які операції визначені в C + + для непрямого виклику методу через покажчик?

245. Що таке «функтор»? Наведіть приклад функціонального класу.

246. Якими способами функтор викликається?

247. Чи можна використовувати успадкування при розробці функторів?

248. Чи дозволяється операцію виклику функції () визначати як віртуальний метод? А як статичний?

249. У чому переваги функторів перед покажчиками на функції?

250. Поясніть, навіщо потрібні адаптери функторів? Які види адаптерів ви знаєте?

251. Як використовуються класи властивостей при розробці функторів?

252. Поясніть, що таке «композиція» та наведіть приклади?

253. Поясніть, чим відрізняється множинне успадкування від простого?

254. Наведіть структуру і принцип дії патерну Adapter.

255. Сформулюйте основну проблему множинного спадкоємства.

256. Чи виконується принцип підстановки при відкритому множині спадкування?

257. Що таке віртуальне успадкування? Які його переваги і недоліки в порівнянні із звичайним наслідуванням?

258. Чи може віртуальне успадкування бути поодиноким?

259. Чи впливає віртуальне успадкування на розмір класу?

260. Поясніть, яким чином за допомогою віртуального наслідування можна взагалі заборонити спадкоємство.

261. Які засоби C + + складають RTTI?

262. Поясніть різницю між підвищує, що знижує і перехресним приведенням.

263. Якими властивостями повинен володіти клас, щоб з ним працював механізм RTTI?

264. У чому приведення покажчиків відрізняється від приведення посилань?

265. Які винятки пов'язані з механізмом RTTI?

266. Що таке «потік» - дайте визначення.

267. Як класифікуються потоки, реалізовані в бібліотеках введення / виведення C + +?

268. Що таке буферизація і навіщо вона потрібна?

269. Які бібліотеки введення / виводу реалізовані в C + + і чим вони відрізняються?

270. Перерахуйте стандартні потоки і поясніть їх призначення.

271. Навіщо потрібен процес форматування і коли він виконується?

272. Що таке «форматна рядок», і в яких функціях вона використовується?

273. Поясніть призначення елементів специфікатора формату.

274. Скільки специфікаторів формату може бути в форматної рядку?

275. Який з елементів специфікатора формату не є замовчуваних?

276. Перерахуйте кілька відомих вам позначень типів в специфікатор формату, і вкажіть їх призначення.

277. Скільки модифікаторів типу ви знаєте, і яку роль модифікатор типу грає в специфікатор формату?

278. За допомогою якого прапора можна вирівняти виведене значення вліво? А яким чином вивести провідні нулі?

279. Яке дію надають на виведену рядок ширина, точнойть і прапори в специфікатор формату?

280. Для чого в специфікатор формату може використовуватися символ зірочка ("*")? Чим відрізняється дія цього символу при воді і при виведенні?

281. Які особливості введення рядків?

282. Яким чином обмежити набір символів, що вводять при введенні?

283. Що є головною проблемою при використанні форматного введення / виводу з бібліотеки <cstdio>?

284. Поясніть, для чого потрібні рядкові потоки. Чому рядкові потоки-завжди форматується?

285. За допомогою яких функцій виконується робота зі строковими потоками?

286. Чи можна використовувати тип string (і яким чином) зі строковими потоками?

287. Поясніть, в чому полягає відмінність між текстовим і двійковим файлом.

288. Поясніть, що означає «відкрити» файл і «закрити» файл?

289. Яким чином зовнішній файл зв'язується з потоком?

290. Чи можна один і той же потік пов'язати з різними файлами? А один і той же файл з різними потоками?

291. Перерахуйте режими відкриття файлу. Чим відрізняється режим "r" від режиму "a"?

292. Яку роль в режимі відкриття грає знак плюс («+»)?

293. У яких випадках необхідно стежити за ситуацією «кінець файлу»? Яким способом це робиться?

294. Чи можна текстовий файл відкрити як двійковий? А двійковий - як текстовий?

295. Які функції введення / виводу використовуються для обміну з текстовими файлами?

296. Перерахуйте функції введення / виводу для роботи з двійковими файлами.

297. Які функції реалізовані в бібліотеці <cstdio> для забезпечення прямого доступу до записів двійкового файлу? Чи можна їх використовувати для роботи з текстовими файлами?

298. Поясніть призначення функції fseek ().

299. Чим відрізняється функція ftell () від функції fgetpos ()?

300. Поясніть, що означає «перенаправлення» потоку? Які потоки можна перенаправляти і куди?

301. Яким чином перенаправлення вводу можна використовувати для введення рядків з пробілами?

302. У чому переваги об'єктно-орієнтованої бібліотека порівняно з процедурної?

303. У яких станах може перебувати потік? Яким чином отслжівается стан «кінець потоку»?

304. Які об'єктно-орієнтовані потоки пов'язані зі стандартними потоками?

305. Чим відрізняються об'єктно-орієнтовані рядкові потоки від процедурних строкових потоків?

306. Яким чином рядкові потоки можна використовувати для обмеження ширини поля введення? А чи можна з тією ж метою використовувати рядкові потоки <cstdio>?

307. Порівняйте засоби форматування об'єктно-орієнтованої та процедурної бібліотеки.

308. Яким чином ввести рядок типу string з пробілами?

309. Яке призначення прапорів форматування? Які засоби реалізовані в бібліотеці для роботи з прапорами форматування?

310. Що таке «маніпулятор»? У чому переваги маніпуляторів перед прапорами форматування?

311. Як зв'язуються файли з потоками в об'єктно-орієнтованої бібліотеці?

312. Чи можна файли, записані функціями бібліотеки <cstdio>, прочитати об'єктно-орієнтованими засобами? А навпаки?

313. Перерахуйте режими відкриття об'єктно-орієнтованих файлових потоків. яким чином комбінуються режими відкриття файлоавих потоків?

314. Чи обов'язково закривати файл, пов'язаний з об'єктно-орієнтованим файловим потоком? А відкривати?

315. Яким чином відкрити файловий потік для читання і запису одночасно?

316. Як відкрити файловий потік для дозапису?

317. Чи можна вивести значення змінної в двійковому вигляді і як це зробити?

318. Чи дозволяється успадковувати від класів бібліотеки введення / виводу?

319. Яким чином можна еренаправіть об'єктно-орієнтована потік?

320. Як використовується буфер потоку для копіювання потоку?

321. Якими операціями виконується форматований введення / виведення в файлові потоки? А неформатований?

322. Реалізовано чи в об'єктно-орієнтованої бібліотеці засоби прямого доступу до файлових потоків? Порівняйте їх з аналогічними засобами бібліотеки <cstdio>.

323. З якими об'єктно-орієнтованими потоками дозволяється, і з якими не дозволяється використовувати засоби прямого доступу?

324. Покажіть, яким чином можна виконати перевантаження операцій введення / виводу для нового типу даних.

325. Як виконується обробка помилок введення / виводу в об'єктно-орієнтованої бібліотеці?

326. Яке стандартне виняток генерується при помилках введення / виводу? Чи обов'язково воно генерується?

327. Чим стандартні широкі потоки відрізняються від вузьких?

328. Що таке - «локаль», і яке її призначення?

329. Як встановити російський шрифт при виведенні в консольне вікно?

330. Чим відрізняється введення / виведення широких файлових потоків від вузьких?

331. Перерахуйте всі послідовні контейнери стандартної бібліотеки. Чим вони відрізняються один від одного?

332. Перерахуйте адаптери послідовних контейнерів і дайте їх детальну характеристику.

333. Чому для адаптерів-черг не можна використовувати вектор в якості базового?

334. Чим проста чергу queue відрізняється від пріоритетною черзі priority_queue?

335. Яким вимогам повинні задовольняти елементи контейнера?

336. Чи можуть бути покажчики елементами контейнера? А ітератори?

337. Чому не можна використовувати в якості елементів контейнера стандартний інтелектуальний покажчик auto_ptr?

338. Навіщо в контейнері list реалізовані власні методи сортування пошуку та злиття? Чи можна користуватися відповідними стандартними алгоритмами при обробці списку?

339. Перерахуйте типові види конструкторів, за допомогою яких можна створювати послідовний контейнер.

340. Чи можна ініціалізувати контейнер елементами вбудованого масиву? А елементами іншого контейнера? Якими способами це можна зробити?

341. Чому конструктор ініціалізації, параметрами якого є ітератори, зроблено шаблонним у всіх контейнерах?

342. Які методи реалізовані в контейнері-векторі для доступу до елементів?

343. Чи відрізняється функція at () доступу за індексом від перевантаженої операції індексування і чим?

344. Перерахуйте методи контейнера deque пов'язані з визначенням розмірів контейнера.

345. Чим метод size () відрізняється від методу capacity ()? А в чому відмінність цих методів від методу max_size ()?

346. Перерахуйте методи контейнера list, призначені для вставки видалення і заміни елементів. Чи відрізняються ці методи від відповідних методів вектора і дека?

347. Яким чином виконуються операції порівняння контейнерів?

348. Чи дозволяється змінювати елемент асоціативного контейнера, доступний в даний момент по ітератор?

349. Які контейнери називаються асоціативними і чому?

350. Чим контейнер map відрізняється від контейнера multimap?

351. Поясніть, чому в асоціативних контейнерах не можна змінювати елемент, доступний в даний момент по ітератор.

352. З яких причин в контейнері-множині не реалізовані типові операції об'єднання, перетину, різниці і інші?

353. Як використовується структура-пара в асоціативних контейнерах?

354. Поясніть, що таке «критерій сортування», і яким вимогам він повинен задовольняти? Який критерій сортування прийнятий за замовчуванням?

355. Якими перевагами володіє функція make_pair () в порівнянні з конструктором pair ()?

356. Чому в контейнерах-відображеннях операція індексування перевантажена, а в контейнерах-множинах - ні?

357. Які гарантії безпеки забезпечують контейнери стандартної бібліотеки?

358. Що таке «транзакційна гарантія безпеки» і чим вона відрізняється від базової?

359. На які 4 класу по надійності можна розділити всі операції з контейнерами?

360. Що таке «розподільник пам'яті» і навіщо він потрібен?

361. Чим відрізняється бітовий вектор bitset від бітового вектора vector <bool>?

362. Дайте визначення ітератора.

363. Що таке «початковий» ітератор і «кінцевий» ітератор? Які методи, пов'язані з ітераторами, обов'язково включає кожен контейнер?

364. Чим константний ітератор відрізняється від неконстантного?

365. Поясніть, що таке «недійсний» ітератор. У яких випадках ітератори стають недійсними?

366. Які категорії итераторов ви знаєте? Які операції обов'язково реалізуються для всіх категорій итераторов?

367. До якого виду итераторов можна віднести вбудований покажчик і чому?

368. Які допоміжні функції для итераторов ви знаєте? У яких випадках виправдано їх застосування?

369. Які адаптери итераторов реалізовані в бібліотеці?

370. Поясніть, чому ітератори реалізовані як вкладені класи в контейнерах.

371. Чим відрізняються ітератори вставки від звичайних итераторов?

372. Яким чином використовуються потокові ітератори?

373. Які стандартні функтори реалізовані в бібліотеці STL? Яке їх основне призначення?

374. Для чого потрібні адаптери функторів bind1st () і bind2nd ()?

375. Як застосовуються адаптери-заперечники?

376. Чому алгоритми remove () не видаляють елементи з контейнерів? Як реально видалити елементи з контейнера?

377. Чим відрізняється стабільна сортування від звичайної?

378. Яку функцію виконує алгоритми unique ()?

379. Чи можуть стандартні алгоритми працювати з рядками?

380. Чи потрібно сортувати асоціативні контейнери?

381. Чи можна алгоритми для роботи з множинами застосовувати для послідовних контейнерів? За яких умов?

382. Які алгоритми призначені для заповнення контейнера значеннями? З якими контейнерами вони можуть працювати?

383. Яким чином заповнити за допомогою алгоритму generate () послідовний контейнер, який не має жодного елемента?

384. Перерахуйте алгоритми, призначені для операцій з кожним елементом контейнера.

385. Чи можна за допомогою алгоритму for_each () змінити елементи контейнера?