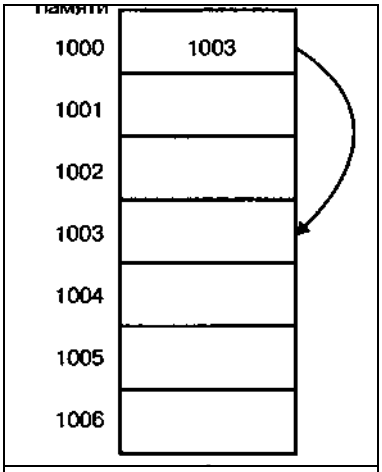


Мал. 1

Існує три причини, по яких неможливо написати гарну програму мовою C/C++ без використання вказівників. По-перше, вказівники дозволяють функціям змінювати свої аргументи. По-друге, за допомогою вказівників здійснюється динамічний розподіл пам'яті. І, по-третє, вказівники підвищують ефективність багатьох процедур. Як ми побачимо надалі, у мові C++ вказівники мають додаткові переваги.

Вказівники - один із самих потужних й, у той же час, самих небезпечних засобів мови C/C++. Наприклад, неініціалізовані вказівники (або вказівники, що містять невірні адреси) можуть знищити операційну систему комп'ютера. І, що ще гірше, неправильне використання вказівників породжує помилки, які вкрай важко виявити.

Оскільки вказівники настільки важливі й небезпечні, ми розглянемо їх досить докладно.

Вказівник (pointer) — це змінна, у якій зберігається адреса іншого об'єкта (як правило, інший змінної). Наприклад, якщо одна змінна містить адресу іншої змінної, говорять, що перша змінна *посилається* (point) на другу. Ця ситуація показана на мал. 1.

Вказівники Змінна, що зберігає адресу комірки пам'яті, повинна бути оголошена як вказівник. Оголошення вказівника складається з імені базового типу, символу * й імені змінної. Загальна форма цього оголошення така.

*тип_вказівника * ім'я_вказівника;*

Тут *тип_вказівника* означає базовий тип вказівника. Ним може бути будь-який допустимий тип.

Базовий тип вказівника визначається типом змінної, на яку він може посилатися. З формальної точки зору вказівник будь-якого типу може посилатися на будь-яке місце в пам'яті. Однак операції адресної арифметики тісно пов'язані з базовим типом вказівників, тому дуже важливо правильно їх оголосити. (Адресна арифметика обговорюється нижче.)

Оператори для роботи з вказівниками

Як відомо, існують два спеціальних оператори для роботи з вказівниками: оператор розіменування вказівника * й оператор одержання адреси &. Оператор & є унарним і повертає адресу свого операнда. (Не забувайте: унарні оператори мають тільки один операнд.) Наприклад, оператор присвоювання

m = &ccount,-

записує в вказівник m адресу змінної **count.** Ця адреса комірки пам'яті, що займає змінна **count**. Адреса й значення змінної ніяк не зв'язані один з одним. Оператор & означає "адресу".

Щоб глибше розібратися в механізмі, що лежить в основі попереднього оператора присвоювання, припустимо, що змінна **count** зберігається в осередку з номером 2000.

Оператор розіменування вказівника * є антиподом оператора &. Цей унарний оператор повертає значення, що зберігається по зазначеній адресі. Наприклад, якщо вказівник m містить адреса змінної **count,** те оператор присвоювання

q = *m;

помістить значення **count** у змінну q. Отже, змінна q стане рівна 100, оскільки саме це число записане в осередку 2000, адреса якої зберігається в вказівнику m. Символ * можна інтерпретувати як "значення, що зберігається за адресою". У цьому випадку попередній оператор означає: "присвоїти змінній q значення, що зберігається за адресою m".

Пріоритет операторів & й * вище, ніж пріоритет всіх арифметичних операторів, за винятком унарного мінуса.

Необхідно мати гарантії, що вказівник завжди посилається на змінну правильного типу. Наприклад, якщо в програмі оголошений вказівник на змінну цілого типу, компілятор думає, що адреса, що у ньому зберігається, ставиться до змінного типу **int,** незалежно від того, чи не так це насправді. Оскільки вказівнику можна привласнити будь-яку адресу, наступна програма буде скомпільована без помилок, але бажаного результату не дасть.

```
#include <iostream>
using namespace std;
int main(void) {
double x = 100.1, y;
int  *p;
/* Наступний оператор змусить вказівник p (який маї цілий тип) що посилатися на змінну типу double. */
```

```
p = &x;
/* Наступний оператор працює не так, як задумано. */
y = *p;
cout<<y; /* Не виводить число 100.1 */
return 0; }
```

Ця програма ніколи не присвоїть змінній y значення змінної x. Оскільки вказівник p є целочисленным, у змінну в будуть скопійовані лише 4 байт (оскільки цілі числа займають 4 байт), а не 8.

Вирази, що містять вказівники

Як правило, вирази, що містять вказівники, підкоряються загальноприйнятим правилам. Однак у них є свої особливості.

Присвоювання вказівників

Вказівник можна присвоїти іншому вказівнику. Розглянемо приклад.

```
#include <iostream>
using namespace std;
int main() {
int x;
int *pl, *p2;
pl = &x;
p2 = p1;
cout<<p2; /* Виводить адресу змінної x, а не її значення! */
return 0; }
```

Тепер на змінну x посилаються обидва вказівники p1 і p2.

Адресна арифметика

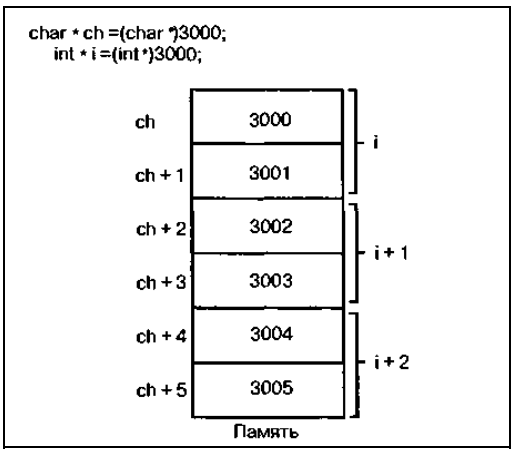
До вказівників можна застосовувати тільки дві арифметичні операції: додавання й віднімання. Припустимо, що вказівник p1 посилається на змінну цілого типу, розміщену за адресою 2000. Крім того, будемо вважати, що цілі числа займають 2 байт у пам'яті комп'ютера. Після обчислення виразу

```
p1++
```

змінна p1 буде дорівнює не 2001, а 2002, оскільки при збільшенні вказівника p1 на одиницю він посилається на наступне ціле число. Це ж стосується й оператора декрементації. Наприклад, якщо вказівник p1 дорівнює 2000, вираз

```
p1--
```

присвоїть вказівнику p1 значення 1998.



Мал. 2

Узагальнюючи наведений вище приклад, сформулюємо наступні правила адресної арифметики. При збільшенні вказівник посилається на осередок, у якій зберігається наступний елемент базового типу. При зменшенні він посилається на попередній елемент. Для вказівників на символи зберігаються правила "звичайної" арифметики, оскільки розмір символів дорівнює 1 байт. Всі інші вказівники збільшуються або зменшуються на довжину відповідних змінних, на які вони посилаються. Це гарантує, що вказівники завжди будуть посилатися на елемент базового типу Описана вище концепція проілюстрована на мал. 2.

Дії над вказівниками не обмежуються операторами інкрементації й декрементації. Наприклад, до них можна додавати й цілі числа. Вираз

```
pi = pi + 12;
```

зміщає вказівник pi на 12-й елемент базового типу, розташований після поточної адреси.

Крім того, вказівники можна віднімати. Це дозволяє визначити кількість об'єктів базового типу, розташованих між двома вказівниками. Всі інші арифметичні операції заборонені.

Порівняння вказівників

Вказівники можна порівнювати між собою. Наприклад, в прикладі нижче порівнювання вказівників p і q, є зовсім правильним.

```
if(p<q) cout<<"Вказівник p містить менше адресу, ніж; вказівник q;
```

Як правило, вказівники порівнюються між собою, коли вони посилаються на той самий об'єкт, наприклад масив.

Вказівники й масиви

Вказівники й масиви тісно зв'язані між собою. Розглянемо наступний фрагмент програми.

```
char  str[80],  *рш;
pi  =  Str;
```

Тут вказівнику рі привласнена адреса першого елемента масиву str. Щоб одержати доступ до п'ятого елемента цього масиву, варто виконати один із двох операторів:

```
str[4]
```

або

```
*(pl+4)
```

Обоє оператора повернуть значення п'ятого елемента масиву str. Нагадаємо, що індексація масивів починається з нуля, тому індекс п'ятого елемента масиву str дорівнює 4. Крім того, п'ятий елемент масиву можна одержати, додавши 4 до вказівника рі, що спочатку посилається на перший елемент.

Цей приклад можна узагальнити. По суті, у мові C/C++ існують два способи звертання до елементів масиву: індексація й адресна арифметика. Хоча індексація масиву наочніше, адресна арифметика іноді виявляється ефективніше.

У наступному фрагменті програми наведені два варіанти функції putstr(), що ілюструє доступ до елементів масиву. У першому варіанті використається індексація, а в другому - адресна арифметика. Функція putstr призначена для посимвольного запису рядка в стандартний потік виводу.

```
/* Індексація вказівника. */
void putstr(char *s) <
register int t;
for(t=0; s[t]; ++t) putchar(s[t]); }

/* Доступ за допомогою вказівника. */
void putstr(char *s)
{   while(*s) putchar(*s++);
 }
```

Більшість професійних програмістів порахують другу версію більш зрозумілою й наочною. На практиці саме цей спосіб доступу до елементів масиву розповсюджений найбільше широко.

Масиви вказівників

Як і всі звичайні змінні, вказівники можна поміщати в масив. Оголошення масиву, що складає з 10 цілочисельних вказівників, виглядає в такий спосіб.

```
int   *x[10];
```

Щоб присвоїти адреса цілочисленной змінної var третьому елементу масиву вказівників, потрібно виконати оператор

```
x[2]   =  &var;
```

Щоб витягти значення змінної var, використовуючи вказівник x[2], необхідно його розіменувати:

```
*x[2]
```

Масив вказівників передається у функцію як звичайно - досить указати його ім'я як параметр. У наступному фрагменті коду на вхід функції надходить масив вказівників.

```
void display_array (int   *q[])
{
int   t ;
for(t=0;   t<10;    t++)
cout<<*q[t];
}
```

Нагадаємо, що змінна q не є вказівником на цілочисельні змінні, її варто інтерпретувати як вказівник на масив цілочисельних вказівників Отже, необхідно оголосити, що параметр q являє собою масив цілочисельних вказівників. Це оголошення зроблене в заголовку функції.

Масиви часто містять вказівники на рядки. Спробуємо створити функцію, що виводить на екран повідомлення про помилку із заданим номером.

```
void svntax_error(int num)
static char *err[] = { "Неможливо відкрити файл\n", "Помилка при читанні\n", "Помилка при записі\n", "Відмова апаратури"};
cout<<  err[num]; }
```

Вказівники на рядки знаходяться в масиві **err.** Вивід відбувається усередині функції **syntax_error** () і виводить на екран рядок із зазначеним номером. Наприклад, якщо параметр num дорівнює 2, на екрані з'явиться повідомлення "Помилка при записі".

Відзначимо, що аргумент командного рядка **argv** також являє собою масив вказівників на символьні змінні (див. главу 3).

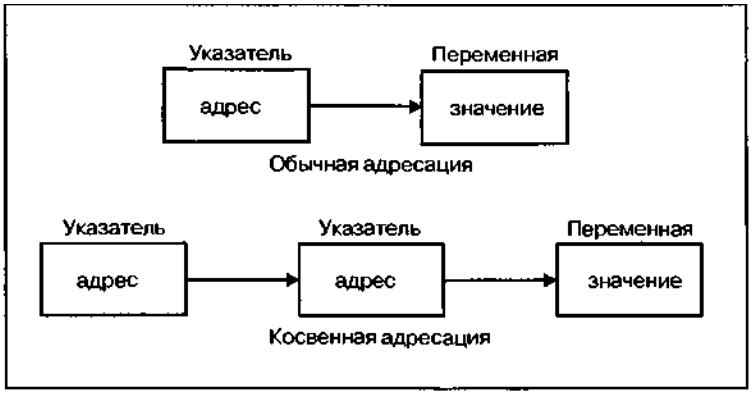


Рис. 5.3. Обычная и косвенная адресация

Непряма адресація

Іноді вказівник може посилатися на інший вказівник, що, у свою чергу, містить адресу звичайної змінної. Така схема називається *непрямою адресацією* (multiple inderection), або *вказівником на вказівник* (pointer to pointer). Застосування непрямої адресації знижує наочність програми. Концепція непрямої адресації проілюстрована на мал. 5.3. Як бачимо, звичайний вказівник зберігає адресу об'єкта, що містить необхідне значення. У випадку непрямої адресації один вказівник посилається на інший вказівник, а той, у свою чергу, містить адресу об'єкта, у якому записане потрібне значення.

Глибина непрямої адресації не обмежена, однак майже завжди можна обійтися "вказівником на вказівник". Більше громіздкі схеми адресації складніше зрозуміти, отже, імовірність помилок при їхньому використанні різко зростає.

Змінна, що представляє собою вказівник на вказівник, оголошується в такий спосіб: перед її ім'ям записується додаткова зірочка. Наприклад, у наступному операторі змінна **newbalance** оголошується як вказівник на вказівник на число із плаваючою крапкою.

```
float **newbalance;
```

Варто пам'ятати, що змінна **newbalance** не є вказівником на число із плаваючою крапкою, вона лише посилається на вказівник цього типу.

Щоб витягти значення змінної за допомогою непрямої адресації, не обходи мо двічі застосувати оператор рзіменування.

```
#include <iostream>
using namespace std;
int main(void) {
int x,   *p,   **q;
x = 10; p = &x; q =  &p;
cout<< **q;  /*  Вивід числа x  */
return  0; }
```

Тут змінна p оголошена як цілочисельний вказівник, а змінна q являє собою вказівник на цілочисельний вказівник. В результаті виконання на екран виведеться число 10.

Ініціалізація вказівників

Якщо нестатичний локальний вказівник оголошений, але не ініціалізований, його значення залишається невизначеним. (Глобальні й статичні локальні вказівники автоматично ініціалізуються нулем.) При спробі застосувати вказівник, що містить невизначену адресу, може зруйнуватися як програма, так і вся операційна система - гірше не придумаєш!

При роботі з вказівниками більшість професійних програмістів мовою C/C++ дотримуються наступного правила: вказівник, що не посилається на конкретну комірку пам'яті, повинен бути рівний нулю. По визначенню будь-який нульовий вказівник ні на що не посилається й не повинен використовуватися. Однак це ще не гарантує безпеки. Використання нульового вказівника - усього лише загальноприйняте правило. Це зовсім не правило, обумовлене мовою C або C++. Наприклад, якщо нульовий вказівник помістити в ліву частину оператора присвоювання, ризик руйнування програми й операційної системи залишається високим.

Оскільки передбачається, що нульовий вказівник в обчисленнях не використається, його можна застосовувати для підвищення наочності й ефективності програм. Наприклад, його можна використати як ознаку кінця масиву, що містить вказівники. Це дозволить запобігти виходу за межі допустимого діапазону. Подібна ситуація ілюструється на прикладі функції **search().**

```
/*  Пошук імені */
int search(char  *p[],   char  *name)
{
register int t;
for(t=0;   p[t];   ++t)
if(!strcmp(p[t], name))   return  t;
return -1;   /*  Ім'я не знайдене  */ }
```

Цикл **for** усередині функції **search** () виконується доти, поки не буде знайдене необхідне ім'я, або не виявиться нульовий вказівник. Як тільки нульовий вказівник

буде виявлений, умова циклу стане помилковим, і програма передасть керування наступному операторові.

Програмісти на C/C++ часто ініціалізують рядки. Приклад такої ініціалізації наведений нижче:

```
char *p = "Привіт";
```

Як бачимо, вказівник p не є масивом. Спробуємо зрозуміти, чому можливий такий спосіб ініціалізації. Всі компілятори мови C/C++ створюють *таблицю рядків* (string table), у якій зберігаються строкові константи, які використовуються в програмі. Отже, поередній оператор привласнить вказівнику p адресу рядкової константи "Привіт", записаної в таблицю рядків. Вказівник p використається в програмі як звичайний рядок (однак змінювати його небажано). Проілюструємо це наступним прикладом.

```
#include <iostream>
# include <cstring>
using namespace std;
char *p = "Привіт";
int main(void) {
register int t;
/* Виводимо рядок у прямому й зворотному порядку. */
cout<<p;
for(t=strlen(p)-l; t>-l; t-) cout<<p[t];
return 0; }
```

З формальної точки зору в стандарті мови C++ рядковий літерал має тип **const char** *. Однак у мові C++ передбачене автоматичне перетворення в тип **char** *. Таким чином, розглянута вище програма є абсолютно правильною. І все-таки ця особливість мови вважається небажаної, тому це перетворення не слід застосовувати. Створюючи нові програми, рядкові літерали необхідно дійсно вважати константами, а оголошення вказівника p записувати в такий спосіб.

```
const char *p = "Привіт";
```

Вказівники на функції

Особливо малозрозумілим, хоча й діючому механізмі мови C++ є *вказівники на функції* (function pointer). Незважаючи на те що функція не є змінною, вона розташовується в пам'яті, і, отже, її адреса можна присвоювати вказівнику. Ця адреса вважається точкою входу у функцію. Саме вона використається при її виклику. Оскільки вказівник може посилатися на функцію, її можна викликати за допомогою цього вказівника. Це дозволяє також передавати функції іншим функціям як аргументи.

Адреса функції задається її ім'ям, зазначеним без дужок й аргументів. Щоб розібратися в цьому механізмі, розглянемо наступну програму.

```
#include <iostream>
#include <cstring>
using namespace std;
void check(char *a, char *b, int (*cmp)(const char *, const char *));
int main(void) {
char s1[80] , s2[80];
int (*p)(const char *, const char *);
p = strcmp;
gets(s1); gets(s2) ;
check(s1, s2, p);
return 0; }
void check(char *a, char *b, int (*cmp)(const char *, const char *))
{
cout<<"Перевірка рівності"<<endl;
if(!(*cmp)(a, b)) cout<"Рівні";
else cout<<"Не рівні"; }
```

При виклику функція **check**() одержує два вказівники на символьні змінні й вказівник на функцію. Відповідні аргументи оголошені в її заголовку. Зверніть увагу на те, як оголошений вказівник на функцію. Цю форму оголошення варто застосовувати для будь-яких вказівників на функції, незалежно від того, який тип мають їхні аргументи й повертають значення, що. Оголошення *cmp знаходиться в дужках для того, щоб компілятор правильно його інтерпретував. Вираз

```
(*cmp)(a, b)
```

усередині функції **check**() означає виклик функції strcmp, на яку посилається вказівник cmp, з аргументами a й b. Дужки як і раніше необхідні. Це один зі способів викликати функцію за допомогою вказівника. Альтернативний виклик виглядає так:

```
cmp(a, b);
```

Перший спосіб застосовується частіше, оскільки він дозволяє явно продемонструвати, що функція викликається через вказівник. (Інакше кажучи, читач програми відразу побачить, що змінна cmp є вказівником, а не ім'ям функції.) Проте ці два способи еквівалентні.

Зверніть увагу на те, що функцію **check** () можна викликати, безпосередньо вказавши функцію strcmp() в якості її аргументу.

```
check(sl,  s2,  strcmp);
```

У цьому випадку відпадає необхідність у додатковому вказівнику на функцію.

Може виникнути закономірне питання: навіщо викликати функції за допомогою вказівників? На перший погляд, це лише ускладнює програму, не надаючи переваг. Проте іноді вигідніше викликати функції через вказівники й навіть створювати масиви вказівників на функції. Розглянемо як приклад синтаксичний аналізатор - складову частину компілятора, що обчислює вирази. Він часто викликає різні математичні функції (синус, косинус, тангенс й т.д.), засобу вводу-виводу або функції доступу до ресурсів системи. Замість створення великого оператора switch, у якому довелося б перераховувати всі ці функції, можна створити масив вказівників на них. У цьому випадку до функцій можна було б звертатися по індексу. Щоб оцінити ефективність такого підходу, розглянемо розширену версію попередньої програми. У цьому прикладі функція check() перевіряє на рівність рядка, що складаються з букв або цифр. Для цього вона просто викликає різні функції, що виконують порівняння.

```
#include <iostream>
#include <cstring>
using namespace std;
void check(char *a, char *b, int (*cmp)(const char *, const char *));
int numcmp(const char *a, const char *b) ;
int main(void) {
char s1[80], s2[80];
gets(s1); gets(s2);
if(isalpha(*s1))
check(sl, s2, strcmp); else
check(sl, s2, numcmp);
return 0; }
void check(char *a, char *b, int (*cmp)(const char *, const char *)) {
cout<<"Перевірка рівності.\n";
if(!(*cmp)(a, b)) cout<<"Рівні";
else cout<<"Не рівні"; }
int numcmp(const char *a, const char *b) {
if (atoi (a) ==atoi (b) ) return 0;
else return 1;}
```

Якщо користувач уведе рядок, що складається з букв, функція **check ()** викличе функцію **strcmp**, одержавши вказівник на неї. У противному випадку буде переданий вказівник на функцію **numcmp ().** Отже, у різних ситуаціях функція **check ()** може викликати різні функції.

Оператори динамічного розподілу пам'яті

У мові C++ передбачені два оператори динамічного розподілу пам'яті: **new** й **delete**. Ці оператори виділяють і звільняють пам'ять у ході виконання програми. Динамічний розподіл пам'яті являє собою важливу частину практично всіх реальних програм. У мові C++ є альтернативний спосіб динамічного розподілу пам'яті, заснована на функціях **malloc ()** і **free()**. Вони збережені для того, щоб досягти сумісності з мовою C. Однак у програмах мовою C++ варто застосовувати оператори **new** й **delete**, оскільки вони мають ряд важливих переваг.

Оператор **new** виділяє область пам'яті й повертає вказівник на її перший клітинку. Оператор **delete** звільняє пам'ять, виділену раніше за допомогою оператора **new**. Загальний вид цих операторів такий.

вказівник = new *тип;*
delete *вказівник;*

Тут *вказівнику* присвоюється адреса першої клітинки виділеної області пам'яті, розмір якої визначається *типом.*

Оскільки обсяг виділеної пам’яті обмежений, пам'ять може виявитися вичерпаною. У цьому випадку оператор **new** згенерує виняткову ситуацію **bad_alloc,** визначену в заголовку **<new>.** Програма повинна перехопити цю ситуацію й обробити її. Якщо в програмі не передбачена обробка виняткової ситуації **bad_alloc,** її виконання буде перервано.

Дія оператора **new** у виняткових ситуаціях визначено стандартом мови C++. Проблема полягає в тім, що не всі компілятори повністю відповідають стандарту. У ранніх версіях мови C++ оператор **new** у випадку відмови повертав нульовий вказівник. Пізніше він став генерувати виняткову ситуацію. У підсумку було вирішено, що за замовчуванням оператор **new** повинен генерувати виняткову ситуацію, однак повертати нульовий вказівник також не забороняється. Отже, у кожному конкретному випадку поводження оператора **new** регламентується розроблювачами компілятора. Зрозуміло, зрештою всі компілятори будуть змушені дотримуватися стандарту, але поки варто орієнтуватися на їхню документацію.

Всі приклади, описані в нашій книзі, написані у відповідності зі стандартом. Якщо ваш компілятор не повністю відповідає стандарту, програми варто змінити.

Розглянемо приклад, у якому виділяється динамічна пам'ять для цілочисельної змінної.

```
#nclude <iostream>
#include <new>
using namespace std;
int main() {
int *p;
try {
p = new int; // Виділити пам'ять для цілочисельної змінної }
catch (bad_alloc xa) {
cout << "Виняткова ситуація\n";
return 1; }
*p = 100;
cout « "За адресою " « p « " ";
cout « "записане значення " « *p « endl;
delete p;
return 0; }
```

Ця програма присвоїть змінній p адресу динамічної пам'яті, у якій може зберігатися цілочислова змінна. Потім у цю область пам'яті записується число 100, а її вміст виводиться на екран. На закінчення програма звільняє виділену пам'ять. Врахуйте, якщо ваш компілятор реалізує оператор **new** так, що у випадку відмови він повертає нульовий вказівник, цю програму прийдеться модифікувати.

Оператор **delete** варто застосовувати тільки до результату оператора new. В іншому випадку можуть виникнути проблеми, наприклад, крах операційної системи.

Оператори **new** й **delete** аналогічні функціям **malloc ()** і **free ()**, але в порівнянні з ними володіють декількома перевагами. По-перше, оператор new автоматично виділяє кількість пам'яті, необхідна об'єкту зазначеного типу. Оператор **sizeof** тепер застосовувати не слід. Оскільки розмір виділюваної пам'яті обчислюється автоматично, помилки виключені. По-друге, оператор **new** автоматично повертає вказівник заданого типу. Тепер не потрібно виконувати приведення типу, як це було при використанні функції **malloc ()**. Оператори **new** й **delete** можна перевантажувати, що дозволяє створювати налаштовувані системи, для виділення динамічної пам'яті.

Хоча формальних правил на цей рахунок не існує, оператори **new** й **delete** не слід змішувати з функціями **malloc ()** і **free ()** у рамках однієї й тієї ж програми, оскільки вони можуть виявитися несумісними.

Ініціалізація виділюваної пам'яті

Виділювану пам'ять можна проініціалізувати заданим значенням. Для цього варто вказати початкове значення після імені типу в операторі new. Загальний вид оператора new у цьому випадку виглядає в такий спосіб.

вказівник = new *тип* *(початкове_значення)*

Зрозуміло, тип початкового значення повинен бути сумісним з типом даних, для яких виділяється пам'ять.

Розглянемо програму, що записує у виділену пам'ять число 87.

```
#include <iostream>
#include <new>
using namespace std;
int main() {
int *p;
try {
p = new int (87); // ініціалізуємо числом 87 }
catch (bad_alloc xa) {
cout « "Виняткова ситуація\n";
return 1; }
cout « "За адресою " « p << " ";
cout « "записане число " « *p « "\n";
delete p;
return 0; }
```

Виділення пам'яті для масивів

За допомогою оператора new можна виділяти пам'ять для масивів. У цьому випадку оператор **new** має такий вигляд.

вказівник = new *тип_масиву [розмір]* ;

Тут розмір задає кількість елементів розташовуваного масиву.

Для звільнення пам'яті, зайнятий масивом, застосовується наступна форма оператора **delete**:

delete [] *вказівник;*

Наприклад, що випливає програма виділяє пам'ять для цілочисельного масиву, що складає з 10 елементів.

```
#include <iostream>
#include <new> using namespace std;
int main() {
    int *p, i;
    try {
        p = new int [10]; /* Виділяємо пам'ять для цілочисленного масиву, що складає з 10
елементів*/
    } catch (bad_alloc xa) {
        cout << "Виняткова ситуація\n";
        return 1; }
    for(i=0; i<10; i++ ) p[i] = i;
    for(i=0; i<10; I++) cout << p[i] << " ";
    delete [] p; // Звільняємо пам'ять, зайняту масивом
    return 0; }
```

Зверніть увагу на оператор **delete**. Як уже вказувалося, при звільненні пам'яті, виділеної для масиву за допомогою оператора **new**, розмір масиву не вказується. (Як буде показано в наступному розділі, ця властивість особливо важливо при роботі з масивами об'єктів.)

При створенні динамічного багатовимірного масиву необхідно в операції **new** вказати всі його розмірності (перший може бути змінною), наприклад:

int n = 5; *// n — количество строк*

int **m = (int **) new int [n][5];.

Розглянемо більш універсальний і безпечний спосіб виділення динамічної пам’яті під двовимірний масив, коли обидві його розмірностізадаються на етапі виконання програми. Наприклад, розподіл динамічної пам’яті для матриці, що має **n** рядків і **m** стовпців та елементи цілоготипу, можна здійснити так:

int n, m;

cout << ” Введіть кількість рядків та стовпців: “;

cin >> n >> m;

int **a = new int *[n]; — оголошення змінної тип «покажчи на покажчик на int» і виділення пам’яті для масиву покажчиків на рядки матриці;

for (int i = 0; i < n; i++)

a[i] = new int [m]; — кожному елементу масиву покажчиків на рядки присвоюється адреса початку ділянки пам’яті, виділеної для рядка матриці.

Виділення пам'яті для об'єктів

Використовуючи оператор **new**, можна динамічно виділяти пам'ять для об'єктів. У цьому випадку оператор поверне вказівник на створений об'єкт. Динамічно створений об'єкт нічим не відрізняється від інших. При його створенні також викликається конструктор (якщо він передбачений), а при звільненні пам'яті викликається відповідний деструктор.

Розглянемо невелику програму, у якій визначений клас **balance**, призначений для зберігання імені людини й суми, що лежить на його банківському рахунку. Усередині функції **main()** об'єкт класу **balance** створюється динамічно.

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;
class balance { double cur_bal; char name[80];
public:
void set(double n, char *s) {
cur_bal = n;
strcpy(name, s); }
void get_bal(double &n, char *s) {
n = cur_bal;
strcpy(s, name); } };
int main() {
balance *p;
char s[80] ;
double n;
try {
p = new balance; } catch (bad_alloc xa){
cout « "Виняткова ситуація\n";
return 1; }
p->set(12387. 87, "Ральф Уилсон");
p->get_bal(n, s) ;
cout « s << ": сума = " << n,-cout « "\n";
delete p;
return 0; }
```

Оскільки змінна p є вказівником на об'єкт, для доступу до його членів використається оператор "->".

Як відомо, динамічні об'єкти можуть містити конструктори й деструктор. Крім того, конструктори можуть мати параметри. Розглянемо модифікацію попередньої програми.

```
#include <iostream>
#include <new>
#nclude <cstring>
using namespace std;
class balance { double cur_bal; char name[80]; public:
balance(double n, char *s) { cur_bal = n; strcpy(name, s); }
~balance() {
cout « "Знищення об'єкта ";
cout << name « "\n"; } void get_bal(double &n, char *s) {
n = cur_bal;
strcpy(s, name); } };
int main() {
balance *p;
char s[80];
double n;
// У цій версії використається ініціалізація
try {
p = new balance (12387.87, "Ральф Уилсон"),-} catch (bad_alloc xa) {
cout << "Виняткова ситуація\n" ,-
return 1; }
p->get_bal(n, s) ;
cout « s « ": сума = " « n; cout « "\n";
delete p;
return 0; }
```

Зверніть увагу на те, що параметри конструктора об'єкта зазначені після імені типу, як при звичайній ініціалізації.

У динамічній пам'яті можна розмістити масиви об'єктів, однак варто мати на увазі одну пастку. Масиви, розміщені в динамічній пам'яті за допомогою оператора **new**, не можна ініціалізувати, тому варто переконатися, що клас містить конструктор, що не має параметрів. У противному випадку компілятор мови C++ не знайде підходящого конструктора й при спробі розмістити масив у динамічній пам'яті видасть повідомлення про помилку.

У наступній версії нашої програми в динамічній пам'яті розміщається масив **balance**, і викликається конструктор, що не має параметрів.

```
#include <iostream>
#include <new>
#nclude <cstring>
using namespace std;
class balance { double cur_bal; char name[80]; public:
balance(double n, char *s) { cur_bal = n; s trcpy(name, s); }
balance() {} // Конструктор без параметрів
~balance() {
cout « "Знищення об'єкта ";
cout « name « "\n"; }
void set(double n, char *s) {
cur_bal = n;
strcpy(name, s); }
void get_bal(double &n, char *s) {
n = cur_bal;
strcpy(s, name); } };
int main() {
balance *p;
char s[80];
double n;
int i ;
try {
p = new balance [3]; // Розміщення масиву } catch (bad_alloc xa) {
cout « "Виняткова ситуація\n";
return 1; }
// Використається оператор ".", а не ->
p[0].set(12387.87, "Ральф Уилсон");
p[l].set(144.00, "А. С. Коннерс"),
p[2].set(-11.23, "И. М. Овердроун");
for(i=0; i<3; i++) { p[i].get_bal(n, s);
cout << s « ": сума = " « n; cout « "\n"; }
delete [] p;
 return 0; }
```

У результаті роботи цієї програми на екран будуть виведені наступні рядки.

```
Ральф Уилсон: сума = 12387.9
А. С. Коннерс: сума = 144
И. М. Овердроун: сума = -11.23
Знищення об'єкта И. М. Овердроун
Знищення об'єкта А. С. Коннерс
Знищення об'єкта Ральф Уилсон
```

Для знищення масиву динамічних об'єктів варто застосовувати оператор **delete [ ]**, щоб можна було викликати деструктор для кожного об'єкта окремо.

Альтернатива nothrow

У випадку недостачі пам'яті оператор **new** може не генерувати виняткову ситуацію, а повертати нульовий вказівник. Цей вид оператора **new** виявляється досить корисним при роботі зі старими компіляторами. Особливо корисно, що виклики функції malloc () можна замінити оператором new. (Це часто доводиться робити при перекладі програм з мови С на мову С++.) Для цього застосовується наступна форма оператора new.

вказівник = new (nothrow) *тип;*

Ця форма оператора **new** нагадує його ранні версії. Оскільки у випадку недостачі пам'яті він не генерує виняткову ситуацію, а повертає нульовий вказівник, його можна впроваджувати в старі програми. Однак у нових програмах краще використати новий варіант оператора new, що генерує виняткову ситуацію **bad_alloc.** Щоб мати можливість використати опцію **nothrow,** варто включити в програму заголовок <new>.

Розглянемо програму, що демонструє альтернативу **new(nothrow).**

```
// Демонстрація альтернативи new(nothrow)
#include <iostream>
#include <new>
using namespace std;
```

```
int main() {
int   *p,    i;
p = new(nothrow) int[32]; // Застосування опції nothrow
if(!p) {
cout << "Виняткова ситуація.\n";
return 1; }
for(i=0; i<32; i++) p[i] = i;
for(i=0; i<32; i++) cout « p[i] « "
delete [] p; // Звільняємо пам'ять
return 0; }
```

Як показує ця програма, при використанні альтернативи **nothrow** варто перевіряти вказівник, що повертає оператором new.

Буферизований оператор new

У мові C++ існує особлива форма оператора **new**, яку можна застосовувати для вказівки альтернативного способу розподілу динамічної пам'яті. Ця форма називається *буферизованим оператором* **new** (replacement form of new). Найбільш корисний цей засіб виявляється при перевантаженні оператора **new** в особливих ситуаціях. Його загальний вид такий.

вказівник = new (*список_аргументів*) *тип;*

Тут *список аргументів* являє собою перерахування значень, розділених комами, які передаються перевантаженому оператору new.

Функції

Функція — це послідовність стейтментів для виконання певного завдання. Часто ваші програми переривають виконання одних функцій заради виконання інших. Ви це постійно робите в реальному житті, наприклад, ви читаєте книгу і згадали, що повинні були зробити телефонний дзвінок. Ви залишаєте *закладку* в своїй книзі, берете телефон і набираєте номер. Після того, як ви вже поговорили, ви повертаєтеся до тієї сторінки в книзі, на якій ви зупинилися.

Програми в C++ працюють схожим чином. Іноді, коли програма виконує код, вона може зіткнутися з викликом функції. **Виклик функції** — це **вираз**, який вказує процесору перервати виконання поточної функції і приступити до виконання іншої функції. Процесор "залишає закладку" в поточній точці виконання, а потім виконує функцію, що викликається. Коли виконання функції, що викликається, — завершено, то процесор повертається до "закладки" і відновлює виконання перерваної функції.

Функція, в якій знаходиться виклик, називається **викликаючою функцією** (англ. *"caller"*). Наприклад:

```
#include <iostream> // для std::cout i std::endl

// Оголошення функції doPrint(), яку ми будемо викликати
void doPrint() {
	std::cout << "In doPrint()" << std::endl;
}

// Оголошення функції main()
int main()
{
	std::cout << "Starting main()" << std::endl;
	doPrint(); // перериваємо виконання main() викликом функції doPrint(). Функція main() в
даному випадку є викликаючою функцією
	std::cout << "Ending main()" << std::endl;
	return 0;
  }
```

Результат виконання програми:

```
Starting main()
In doPrint()
Ending main()
```

Ця програма починає своє виконання з першого рядка функції main(), в якому виводиться на екран Starting main(). Другий рядок функції main() викликає функцію doPrint(). На цьому етапі виконання стейтментів у функції main() призупиняється і процесор переходить до виконання стейтментів всередині функції doPrint(). Перший (і єдиний) рядок в doPrint() виводить текст In doPrint().

Коли процесор завершує виконання doPrint(), він повертається назад в функцію main() до тієї точки, на якій зупинився. Отже, наступним стейтментом є вивід рядка Ending main() на екран.

Зверніть увагу, для виклику функції потрібно вказати її ім'я і список параметрів в круглих дужках (). У вищенаведеному прикладі параметри не використовуються, тому круглі дужки порожні. Ми поговоримо детально про параметри функцій на відповідному уроці.

Правило: Не забувайте вказувати круглі дужки () при виклику функцій.

Значення, що повертаються

Коли функція main() завершує своє виконання, вона повертає цілочисельне значення назад в операційну систему, використовуючи **оператор return**.

Функції, які ми пишемо, також можуть повертати значення. Для цього потрібно вказати **тип повернення значення**. Він вказується при оголошенні функції, перед її ім'ям. Зверніть увагу, що тип повернення не вказує, яке саме значення повертатиметься. Він вказує тільки тип цього значення.

Після цього всередині функції, що викликається, ми використовуємо оператор return, щоб вказати фактичне **значення, що повертається**.

Розглянемо просту функцію, яка повертає цілочисельне значення:

```
#include <iostream>

// int означає, що функція повертає цілочисельне значення у викликаючу функцію
int return7()
{
	// Ця функція повертає цілочисельне значення, тому ми повинні використовувати оператор return
	return 7; // повертаємо число 7 у викликаючу функцію
}

int main()
{
	std::cout << return7() << std::endl; // на екран виведеться 7
	std::cout << return7() + 3 << std::endl; // на екран виведеться 10

	return7(); // значення 7, що повертається, - ігнорується, тому що main() з ним нічого не робить

	return 0;
}
```

Результат виконання програми:

```
7
10
```

Тепер давайте розберемося детально:

- Перший виклик функції return7() повертає 7 у викличаючу функцію, де воно потім передається в **std::cout** для виводу на екран.
- Другий виклик функції return7() знову повертає 7 у викликаючу функцію. Вираз 7 + 3 генерує результат 10, який виводиться на екран.
- Третій виклик функції return7() знову повертає 7 у викликаючу функцію. Проте main() нічого з ним не робить, тому нічого і не відбувається (значення, що повертається, просто ігнорується).

Примітка*:* Значення, що повертаються, не виводяться на екран, якщо їх не передавати об'єкту std::cout. В останньому виклику функції return7() значення не надсилається в std::cout, тому нічого і не відбувається.

Тип повернення void

Функції можуть і не повертати значення. Щоб повідомити компілятору, що функція не повертає значення, потрібно використати **тип повернення void**. Погляньмо ще раз на функцію doPrint() з вищенаведеного прикладу:

```
void doPrint() // void - це тип повернення
{
	std::cout << "In doPrint()" << std::endl;
	// Ця функція не повертає значення, тому і оператор return тут не потрібен
```

```
}
```

Ця функція має тип повернення void, який означає, що функція не повертає значення. Оскільки значення не повертається, то і оператор return тут не потрібен.

Ось ще один приклад використання функції типу void:

```
#include <iostream>

// void означає, що функція не повертає значення
void returnNothing()
{
	std::cout << "Hi!" << std::endl;
	// Ця функція не повертає значення, тому і оператор return тут не потрібен
}

int main()
{
	returnNothing(); // функція returnNothing() викликається, але в main() нічого не повертається

	std::cout << returnNothing(); // помилка: цей рядок не скомпілюється (вам потрібно буде його закоментувати)
	return 0;
}
```

У першому виклику функції returnNothing () виводиться Hi!, але нічого не повертається назад у викликаючу функцію. Точка виконання повертається назад в main(), де програма продовжує своє виконання.

Другий виклик функції returnNothing () навіть не скомпілюється. Функція returnNothing() має тип повернення void, який означає, що ця функція не повертає значення. Однак main() намагається відправити це значення (яке не повертається) в std::cout для виведення на екран. std::cout не може опрацювати цей випадок, оскільки не було надано значення для виводу. Отже, компілятор видасть помилку. Вам потрібно буде **закоментувати** цей рядок, щоб компіляція пройшла успішно.

Повернення значень функцією main()

Тепер у вас є розуміння того, як працює функція main(). Коли програма виконується, операційна система викликає функцію main() і починається її виконання. Стейтменти в main() виконуються послідовно. В кінці функція main() повертає цілочисельне значення (зазвичай 0) назад в операційну систему. Тому main() оголошується як int main().

Чому потрібно повертати значення назад в операційну систему? Справа в тому, що значення, що повертається функцією main(), є **кодом стану**, який повідомляє операційній системі про те, чи успішно було виконання програми. Зазвичай, значення 0 (нуль) означає що все пройшло успішно, тоді як будь-яке інше значення означає невдачу/помилку.

Зверніть увагу, за стандартами мови C++ функція main() повинна повертати цілочисельне значення. Однак, якщо ви не вкажете return в кінці функції main(), то компілятор поверне 0 автоматично, якщо ніяких помилок не буде. Проте все ж рекомендується вказувати оператор return в кінці функції main() і використовувати тип повернення int для функції main().

Детальніше про значення, що повертаються

По-перше, якщо тип повернення функції не є void, то вона повинна повертати значення зазначеного типу (використовуючи оператор return). Єдиний виняток — це функція main(), яка повертає 0, якщо не надано інше значення.

По-друге, коли процесор зустрічає в функції оператор return, він негайно виконує повернення значення назад у викликаючу функцію і точка виконання також переходить у викликаючу функцію. Будь-який код, який знаходиться за оператором return у функції — ігнорується.

Функція може повертати тільки одне значення через оператор return у викликаючу функцію. Це може бути або число (наприклад, 7), або значення змінної, або вираз (який генерує результат), або певне значення з набору можливих значень.

Є способи обійти правило повернення одного значення, але про це трішки пізніше.

Нарешті, автор функції вирішує, що означатиме значення, яке повертає функція. Деякі функції використовують повернені значення в якості кодів стану для надання інформації щодо результату

виконання функції (успішне виконання чи ні). Інші функції повертають певне значення з набору можливих значень, ще інші функції взагалі нічого не повертають.

Повторне використання функцій

Одну і ту ж функцію можна викликати декілька разів, навіть в різних програмах, що дуже корисно:

```
#include <iostream>

// Функція getValueFromUser() отримує значення від користувача, а потім повертає його назад у викликаючу функцію
int getValueFromUser()
{
	std::cout << "Enter an integer: ";
	int x;
	std::cin >> x;
	return x;
}

int main()
{
	int a = getValueFromUser(); // перший виклик функції getValueFromUser()
	int b = getValueFromUser(); // другий виклик функції getValueFromUser()

	std::cout << a << " + " << b << " = " << a + b << std::endl;

	return 0;
  }
```

Результат виконання програми:

Enter an integer: 4
Enter an integer: 9
4 + 9 = 13

Тут виконання функції main() переривається 2 рази. Зверніть увагу, в обох випадках отримане значення від користувача зберігається в змінній x, а потім передається назад в функцію main() за допомогою оператора return, де присвоюється змінній a або b!

Також main() не є єдиною функцією, яка може викликати інші функції. Будь-яка функція може викликати будь-яку іншу функцію!

```
#include <iostream>

void printO()
{
	std::cout << "O" << std::endl;
}

void printK()
{
	std::cout << "K" << std::endl;
}

// Функція printOK() викликає як printO(), так і printK()
void printOK()
{
	printO();
	printK();
}

// Оголошення функції main()
int main()
{
	std::cout << "Starting main()" << std::endl;
	printOK();
	std::cout << "Ending main()" << std::endl;
	return 0;
  }
```

Результат виконання програми:

```
Starting main()
OK
Ending main()
```

Вкладені функції

В мові C++ одні функції не можуть бути оголошені всередині інших функцій (тобто бути вкладеними). Наступний код викличе помилку компіляції:

```
#include <iostream>

int main()
{
	int boo() // ця функція знаходиться всередині функції main(), що є заборонено
	{
		std::cout << "boo!";
		return 0;
	}

	boo();
	return 0;
  }
```

Правильно ось так:

```
#include <iostream>

int boo() // тепер вже не в функції main()
{
	std::cout << "boo!";
	return 0;
}

int main()
{
	boo();
	return 0;
  }
```

Параметри і аргументи функції

У багатьох випадках нам потрібно буде передавати дані в функцію, що викликається, щоб вона могла з ними якось взаємодіяти. Наприклад, якщо ми хочемо написати функцію множення двох чисел, то нам потрібно якимось чином повідомити функцію про те, що це будуть за числа. В іншому випадку, як вона дізнається, що на що множити? Тут нам на допомогу приходять параметри і аргументи.

Параметр функції — це змінна, яка використовується в функції і значення якої надає викликаюча функція. Параметри вказуються при оголошенні функції в круглих дужках. Якщо їх багато, то вони перераховуються через кому, наприклад:

```
// Ця функція не має параметрів
void doPrint()
{
	std::cout << "In doPrint()" << std::endl;
}

// Ця функція має один параметр цілочисельного типу - a
void printValue(int a)
{
	std::cout << a << std::endl;
}

// Ця функція має два параметри цілочисельного типу - a і b
int add(int a, int b)
{
	return a + b;
```

```
}
```

Параметри кожної функції дійсні тільки всередині цієї функції. Тому, якщо і printValue(), і add() мають параметр a, то це не означає, що відбудеться конфлікт імен. Ці параметри вважаються окремими і ніяк не взаємодіють один з одним.

Аргумент функції — це значення, яке передається з викликаючої функції в функцію, що викликається, і яке вказується в круглих дужках при виклику:

```
printValue(7); // 7 – це аргумент функції printValue()
  add(4, 5); // 4 і 5 – це аргументи функції add()
```

Зверніть увагу, аргументи також перераховуються через кому. Кількість аргументів має збігатися з кількістю параметрів, інакше компілятор видасть повідомлення про помилку.

Як працюють параметри і аргументи функцій?

При виклику функції, всі її параметри створюються як локальні змінні, а значення кожного з аргументів копіюється у відповідний параметр (локальну змінну). Цей процес називається *"передачею по значенню"*, наприклад:

```
#include <iostream>

// Ця функція має два параметри типу int: a і b.
// Значення змінних a і b визначає викликаюча функція
void printValues(int a, int b)
{
	std::cout << a << std::endl;
	std::cout << b << std::endl;
}

int main()
{
	printValues(8, 9); // тут два аргументи: 8 і 9

	return 0;
  }
```

При виклику функції printValues() аргументи 8 і 9 копіюються в параметри a і b. Параметру a присвоюється значення 8, а параметру b — значення 9.

Результат:

```
8
9
```

Як працюють параметри і значення, що повертаються?

Використовуючи параметри і значення, що повертаються, ми можемо створювати функції, які можуть приймати і опрацьовувати дані, а потім повертати результат назад у викликаючу функцію.

Наприклад, ось проста функція, яка приймає два цілих числа і повертає їх суму:

```
#include <iostream>

// Функція add() приймає два цілих числа в якості параметрів і повертає їх суму.
// Значення a і b визначає викликаюча функція
int add(int a, int b)
{
	return a + b;
}

// Функція main() не має параметрів
int main()
{
	std::cout << add(7, 8) << std::endl; // аргументи 7 і 8 передаються в функцію add()
	return 0;
  }
```

При виклику функції add(), параметру a присвоюється значення 7, а параметру b — значення 8. Після цього функція add() обчислює їх суму і повертає результат назад у функцію main(), де він вже виводиться на екран.

Результат виконання програми:

15

# include <iostream> 	
using namespace std;
 	
void func(int *a, int *b)
{
int c=*a;
*a = *b; *b =c ;
}
void main()
{
int a = 10; int b = 20;
func(&a,&b);
cout <<"a="<<a<<endl;
cout <<"b="<<b<<endl;
}
	# include <iostream>
	using namespace std;
	
	void func(int &a, int &b)
	{
	int c=a;
	a = b; b =c ;
	}
	void main()
	{
	int a = 10; int b = 20;
	func(a,b);
	cout <<"a="<<a<<endl;
	cout <<"b="<<b<<endl;
	}	