Лекція 5
Керуючі конструкції

У цій лекції...

5.1. Блок

При використанні операторів присвоювання й арифметичних виразів можна створювати лише найпростіші програми, що імітують калькулятор. Щоб виконати більш складні обчислення, необхідні засоби керування і способи організації циклів. Оператори, що забезпечують такі можливості, називаються *керуючими*. З цим поняттям тісно зв'язана концепція *керуючої конструкції*, що складається з керуючого оператора і сукупності зв'язаних з ним операторів.

Однією з найпростіших конструкцій, необхідних для організації керування потоком виконання програми, є *складений* оператор, чи *блок*.

Складений оператор — це набір операторів, розглянутий як єдине ціле. У мові C++ для організації блоків використовують фігурні дужки. Ми уже зіштовхувалися з блоками, розглядаючи тему, зв'язану з маскуванням імен змінних.

Блок

```
#include <iostream.h>

int main()
{
    double var = 0; // Локальна змінна
    {
       // Блок
       double var = 1;  // Нова локальна змінна
       cout << "Локальна змінна var у блоці         = " << var << endl;
    }
    cout << "Локальна змінна var у функції main  = " << var << endl;
    return 0;
}
```

Область видимості всіх локальних змінних, оголошених усередині блоку, обмежена його рамками. Тому результати роботи цієї програми виглядають так.

```
Локальна змінна var у блоці         = 1
Локальна змінна var у функції main  = 0
```

5.2. Оператор послідовного обчислення

Близьким родичем блоків є *оператор послідовного обчислення*, що зв'язує декілька виразів в одне ціле. Символом цього оператора є кома. Значенням цього оператора є значення останнього оператора, що стоїть в правій частині. Наприклад, оператор

```
var1 = (var2=100, var1+var2);
```

спочатку привласнює змінної `var2` значення 100, потім обчислює суму змінних `var1` і `var2` і копіює результат у змінну `var1`. Якщо зняти дужки, зміст вираз зовсім зміниться, хоча із синтаксичної точки зору усі залишається правильним.

```
var1 = var2=100, var1+var2;
```

Тепер значення змінної `var1` дорівнює 100, а результат додавання `var1+var2` повисає в повітрі.

Цей оператор корисний, якщо деякі оператори повинні виконуватися обов'язково одночасно. У блоці ж вони виконуються послідовно.

Варто мати на увазі, що коми, що розділяють аргументи функції при її виклику, не є операторами.

Оператор послідовного обчислення

```
#include <iostream.h>
int sumxy1(int);
int sumxy2(int, int);
```

```
int main()
{
   int x=1, y = 2;
   cout << "x + y = " << sumxy1((x,x+y)) << endl; // Один аргумент
   cout << "x + y = " << sumxy2(x,x+y)   << endl; // Два аргументи
   return 0;
}

int sumxy1(int x)
{
   cout << "sumxy1: x = " << x << endl;
   return x;
}

int sumxy2(int x, int y)
{
   cout << "sumxy2: x = " << x << " y = " << y << endl;
   return x+y;
}
```

Результат роботи цієї програми приведений нижче.

```
sumxy1: x = 3
x + y = 3
sumxy2: x = 1 y = 3
x + y = 4
```

5.3. Оператор if

При реалізації складних алгоритмів програміст часто змушений прибігати до різноманітних обчислень, що залежать від умов, що виникають у ході виконання програми. У найпростішому випадку для цього досить конструкції «якщо ..., то ...», що у мові C++ реалізується за допомогою *оператора розгалуження*, чи *умовного оператора*. Він надає програмісту засоби перевірки умови і виконання обчислень у залежності від результату перевірки.

Існує два різновиди операторів розгалуження: двохваріантні і різноманітні. У даному розділі ми розглянемо «вироджений» вид двохваріантного оператора розгалуження — одноваріантний оператор розгалуження `if`. Він має наступний вид.

```
if (вираз) оператор;
```

Як *вираз* може застосовуватися будь-який вираз, результат якого може інтерпретуватися як «істина» чи «хибність». Якщо вираз помилковий, тобто дорівнює нулю, *оператор* не виконується. У противному випадку оператор виконується.

Блок дозволяє виконувати не тільки один, але і відразу декілька операторів. Таким чином, оператор `if` допускає наступний варіант.

```
if (вираз)
{
  оператор_1;
  оператор_2;
   ...
  оператор_N
}
```

Оскільки результат умовного виразу повинний інтерпретуватися як «істина» чи «хибність», найчастіше в його ролі виступають оператори порівняння.

```
if (оператор порівняння)
{
  оператор_1;
  оператор_2;
   ...
  оператор_N
}
```

Зрозуміло, якщо умова достатня є складною, можна удатися до сполучення операторів порівняння і логічних операторів.

```
if ((a>b) && (c==d))
{
  оператор_1;
  оператор_2;
  ...
```

```
   оператор_N;
}
```

Зверніть увагу на три моменти. По-перше, дужки навколо умовного виразу обов'язкові. Ця помилка не дуже страшна, тому що ее легко виловить компілятор. По-друге, при перевірці рівності новачки часто плутають знак присвоювання = і оператора перевірки на рівність ==. Оскільки оператор присвоювання повертає результат rvalue, його можна інтерпретувати в залежності від значення: якщо rvalue дорівнює 0 — «хибність», якщо не 0 — «істина». Виловити таку помилку набагато складніше, хоча компілятор може видати попередження про некоректне присвоювання. По-третє, що саме неприємне, існує помилкова думка, що комп'ютер точно обчислює значення функції. Розглянемо приклад.

Оператор if (порівняння дійсних чисел)

```
 #include <iostream.h>
#include <math.h>

int main()
{
   double var = sin(M_PI);
   cout << "sin(M_PI) = " << endl;
   if (var == 0) cout << "sin(M_PI) == 0";
   return 0;
}
```

На екран виводиться число.

```
1.224606e-16
```

Звичайно, sin(π)=0, але тільки на папері! Комп'ютер працює з раціональними числами, отже, число π апроксимується раціональною константою M_PI, що дорівнює 3.14159265358979323846. У результаті одержуємо, що sin(M_PI)=1.224606e-16, а зовсім не нулю!

Варто мати на увазі, що при порівнянні дійсних чисел необхідно виявляти обережність. Порівняння на рівність практично завжди чревате помилками, а при порівнянні «більше» і «менше» необхідно робити допуск. У даному випадку допуск повинний бути дуже маленьким.

```
if (var <= 1.0e-15) cout << "sin(M_PI) == 0";
```

Помітимо також, що оператор

```
if(вираз != 0) оператор;
```

функціонально еквівалентний наступному оператору.

```
if(вираз) оператор;
```

Хоча ця форма і коротше, але ніякої вигоди вона не дає. Більш того, читабельність програми знижується через такого укороченого способу запису.

Відзначимо також, що на оператор, який повинний виконуватися в результаті перевірки умовного виразу, не накладаються ніякі обмеження. Отже, він може, у свою чергу, бути новим оператором if. Таким чином, оператор if може бути вкладеним. Кількість рівнів вкладення не повинне перевищувати 256. Зрозуміло, на практиці ніхто не створює такі глибокі ієрархії умовних операторів.

Оператор if володіє однією цікавою особливістю — як умовне вираз може виступати оголошення нової змінної.

Оголошення в операторі розгалуження

```
 #include <iostream.h>
int even(int);

int main()
{
   int j=2;
   if(int i = even(j))
    {
      cout << "Ініціалізація парним числом" << endl;
      // Кінець області видимості змінної i
   }
   return 0;
}

int even(int n)
{
   if(n%2 == 0) return 1;
   else return 0;
}
```

Оголошення змінної в умовному операторі дозволяє обмежити її область видимості відповідним блоком.

5.4. Оператор else

Повноцінний двохваріантний оператор розгалуження створюється за допомогою ключового слова `else`.

```
if(вираз)
  оператор_1;
else
  оператор_2;
```

Його семантика полягає в тім, що *оператор_1* виконується, тільки якщо значенням умовного вираз є «істина»; у противному випадку виконується *оператор_2*. Ці оператори є взаємовиключними і ніколи не виконуються одночасно.

Цікава проблема виникає, коли двохваріантні оператори розгалуження вкладаються. Розглянемо наступний фрагмент коду.

```
if (var1 == 0)
  if (var2 == 0)
    var3 = 0;
else
    var3 = 1;
```

У даному випадку порушений баланс операторів `else` і `if`. За замовчуванням оператор `else` завжди відповідає найближчому попередньому оператору `if`. Інакше кажучи, з функціональної точки зору цей фрагмент еквівалентний наступному.

```
if (var1 == 0 && var2 == 0)
    var3 = 0;
else
    var3 = 1;
```

5.5. Тернарная альтернатива

У мові C++ існує скорочений оператор `if-else`, що називається *тернарною альтернативою*.

вираз3: вираз2?вираз3;

Спочатку обчислюється *вираз1*. Потім, якщо він истинний, обчислюється *вираз2*, а якщо хибний — *вираз3*. Тернарний оператор еквівалентний наступній конструкції.

```
if(вираз1)
   вираз2;
else
   вираз3;
```

Результатом цього оператора є значення обчисленого виразу. Крім того, тернарні оператори можуть бути вкладеними. Проілюструємо це наступною програмою.

Тернарний оператор

```
#include <iostream.h>
#include <math.h>

int main()
{
    int var1 = 0, var2 = 1, var3 = 2;

    var2 = (var1 == 0) ? 0 : 1;
    var3 = (var2 == 0) ? ( var2 = (var1 == 1 ? 0 : 1 )) : 2;
    cout << var2 << endl << var3;

    return 0;
}
```

Оскільки початкове значення змінної `var1` дорівнює `0`, у результаті виконання першого тернарного оператора змінна `var2` набуває значення `0`. Отже, у другому операторі обчислюється вираз, що стоїть після знаку питання, — новий тернарний оператор. Його значенням є число `1`. Таким чином, у підсумку змінної `var3` привласнюється значення оператора `var2 = (var1 == 1 ? 0:1)`, тобто число `1`, а змінна `var2` набуває значення `1`.

5.6. Оператор switch

Вкладені оператори двохваріантного розгалуження дозволяють змоделювати будь-яку ситуацію. Уявимо собі, що ми програмуємо реакцію на натискання клавіші і хочемо, щоб у відповідь на натискання різних букв виконувалися різні оператори.

Програма, що реагує на натискання клавіш

```
#include <iostream.h>
#include <conio.h>
```

```
int main()
{
   char press;
   press = getch();
   if(press == 'a' || press == 'b') cout << "Буква a чи b";
   if(press == 'c') ;
   if(press == 'd') cout << "Буква d";
   return 0;
}
```

Розглянемо його недоліки. По-перше, програма виконує кожний з операторів порівняння, хоча після натискання потрібної клавіші інші перевірки стають зайвими. По-друге, не визначена реакція програми на натискання непередбаченої клавіші. Виправити положення дозволяє оператор різноманітного розгалуження `switch`.

```
switch(вираз)
{
 case значення: послідовність операторів; break;
 case значення: послідовність операторів; break;
 ...
 case значення: послідовність операторів; break;
 default: оператор;
}
```

Спочатку обчислюється значення *вираз*. Потім воно порівнюється з кожним зі значень, що відповідають різним варіантам `case`. Значенням *вираз* повинний бути символ чи ціле число. Вираз, результатом якого є число з крапкою, що плаває, не допускаються. Коли виявляється збіг, виконується відповідна послідовність операторів, поки не зустрінеться оператор `break` чи не буде досягнутий кінець оператора `switch`. Якщо в якому-небудь розділі оператор `break` пропущений, виконання автоматично переходить до наступного розділу `case` (фігурально виражаючи, потік керування «провалюється на нижній поверх»). Якщо значення виразу не збігається ни жодною з констант, виконується розділ `default`. Цей розділ оператора `switch` є необов'язковим. Якщо він не передбачений, під час відсутності збігів не буде виконаний жодний оператор.

Стандарт мови C++ допускає до 16384 розділів `case`. Однак важко уявити, щоб програміст вручну передбачив виконання 16384 варіантів. Слід зазначити, що оператор `case` є міткою і не має самостійного значення. Він використовується тільки усередині оператора `switch`.

Оператор `break` відноситься до групи операторів переходу. Він використовується не тільки в операторі `switch`, але й у циклах. Коли потік керування досягає оператора `break`, програма виконує перехід до оператора, що є настпним за оператором `switch`.

Відзначимо основні властивості оператора `switch`.

- На відміну від оператора `if`, оператор `switch` порівнює значення умовного виразу винятково на рівність з константами. Інші види порівняння і логічних виразів не допускаються.
- Усі константи в розділах `case` повинні бути взаємовиключними, за винятком випадку, коли один оператор `switch` вкладений в іншій.
- Символьні константи, що використовуються в операторі `switch`, автоматично перетворюються в целочисельні.

Якщо та сама послідовність операторів відповідає декільком міткам `case`, їх можна об'єднати.

Таким чином, програму, що реагує на натискання клавіш, можна переписати в такий спосіб.

Об'єднані мітки

```
#include <iostream.h>
#include <conio.h>

int main()
{
   char press;
   press = getch();
   switch (press)
   {
      case 'a': case 'b': cout << "Буква a чи b" << endl; break;
      case 'c': break;
      case 'd': cout << "Буква d" << endl; break;
      default: cout << "За замовчуванням ..."; break;
   }
   return 0;
}
```

Оператори `switch` можуть бути вкладеними. При цьому константи розділів `case` зовнішнього і внутрішнього операторів `switch` можуть збігатися.

Вкладені оператори

```
#include <iostream.h>
#include <conio.h>

int main()
{
    int press1, press2;
   press1 = getch();
   switch(press1)
   {
     case 'a' : case 'b': cout << "Буква a чи b" << endl; break;
     case 'c' : cout << "Буква c" << endl; press2 = getch();
     switch (press2)
     {
        case '1' : cout << "Цифра 1" << endl; break;
        case '2' : cout << "Цифра 2" << endl; break;
        default  : cout << "Цифра digit" << endl; break;
     }
     break;
     case 'd' : cout << "Буква d" << endl; break;
     default  : cout << "За замовчуванням ..." << endl; break;
   }
   return 0;
}
```

Ця програма має цікаву особливість, зв'язану з властивістю оператора `break`. Подумки викинемо з тексту оператор `break`, що стоїть після складеного оператора `switch`. Тоді попередній оператор `break` адресував би потік керування оператору, зв'язаному з константою `'d'`, незалежно від значення змінної `press1`! Щоб цього не трапилось, необхідний додатковий оператор переходу, що перериває виконання зовнішнього оператора `switch`.

До речі, мітка в розділі `case` може бути не тільки литералом `1`, `'a'` і так далі, але і справжньою інтегральною константою або елементом перерахування. Таким чином, цілком припустимі наступні програми.

Застосування целочисленных констант до якості міток

```
#include <iostream.h>
#include <conio.h>

int main()
{
   char press;
    const int a= 97, b = 98, c = 99, d = 100;
   press = getch();
   switch(press)
   {
     case a : cout << "Буква a" << endl; break;
     case b : cout << "Буква b" << endl; break;
     case c : cout << "Буква c" << endl; break;
     case d : cout << "Буква d" << endl; break;
     default  : cout << "За замовчуванням ..." << endl; break;
   }
   return 0;
}
```

Застосування перерахувань як міток

```
 #include <iostream.h>
#include <conio.h>

int main()
{
   char press;
    enum {a= 97, b, c, d};
   press = getch();
   switch(press)
   {
      case a : cout << "Буква a" << endl; break;
      case b : cout << "Буква b" << endl; break;
      case c : cout << "Буква c" << endl; break;
      case d : cout << "Буква d" << endl; break;
      default  : cout << "За замовчуванням ..." << endl; break;
```

```
    }
    return 0;
}
```

5.7. Оператори циклу

Оператори циклу забезпечують повторне виконання обчислень. У мові C++ передбачено три види таких операторів. Вони відрізняються механізмами керування циклом та їх розташуванням. Цикл `for`, кількість повторень якого задано заздалегідь, керується лічильником і логічними виразми, розташованими в його початку. Цикл `while` виконується доти, поки не буде виконана задана логічна умова, сформульована в його початку. На противагу йому цикл `do-while` перевіряє умова виходу з циклу наприкінці.

Тіло циклу — це набір операторів, що повторно виконуються. У залежності від розташування умови виходу з циклу, цикли розділяються на цикли з попередньою перевіркою (`for`, `while-do`) і на цикл із наступною перевіркою (`do-while`).

5.7.1. Цикл із лічильником for

Оператор `for` керується змінною, що називається *лічильником циклу*. Крім того, до складу механізму керування входять оператори *ініціалізації* лічильника, перевірки *умови виходу з циклу* і оператор *зміни лічильника*. Величину зміни лічильника називають *кроком*. Початкове і кінцеве значення лічильника, а також крок циклу називають *параметрами циклу*. Оскільки лічильник циклу в ході його виконання постійно змінюється, його часто оголошують як реєстрову змінну.

Основні питання, зв'язані з циклом `for`, такі.

- Який тип і яку область видимості має лічильник циклу?
- Чому дорівнює лічильник циклу після завершення циклу?
- Чи змінюється значення лічильника і параметрів циклу усередині циклу?
- Чи обчислюються параметри циклу на кожній ітерації?

Загальний вид оператора `for` такий.

`for (`*`ініціалізація`*`; `*`умова`*`; `*`зміна`*`)`

Розглянемо послідовність його виконання. Першим виконується оператор ініціалізації, що задає початкове значення лічильника. Природно, ініціалізація здійснюється за допомогою оператора присвоювання. Потім перевіряється *умова виходу з циклу*. Цикл виконується доти, поки значення цього вираз залишається істинним. Оператор *зміни* змінює значення лічильника циклу *після* чергової ітерації. Він виконується *після* виконання тіла циклу. Зверніть увагу на те, що оператори керування циклом з лічильником відокремлюються крапкою з комою. Якщо умова виходу з циклу стала помилковою, керування передається оператору, розташованому після тіла циклу.

Оператор циклу

```
 #include <stdio.h>

int main()
{
   int i;
   for(i=33;i<=122;i++)
   {
     printf("%c",i);
   }
}
```

У результаті на екран виводяться символи, ASCII-коди яких змінюються в діапазоні від 33 до 122.

```
!"#$%&'()*+,-
./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnop
qrstuvwxyz
```

Лічильник циклу може мати будь-як тип. Зокрема, для того щоб вивести на екран числа 33.0, 34.0, ..., 122.0, можна використовувати лічильник циклу типу `float`.

Лічильник циклу типу float

```
 #include <stdio.h>

int main()
{
   float i;
   for(i=33;i<=122;i=i+1)
   {
     printf("%f",i);
   }
}
```

З огляду на, що тип `char` також відноситься до інтегральних типів, символи можна вивести в такий спосіб.

Лічильник циклу типу char

```
#include <stdio.h>

int main()
{
   for(char i=33;i<=122;i++)
   {
     printf("%c",i);
   }
}
```

Ми проілюстрували одну цікаву особливість мови C++: допускається визначення лічильника циклу усередині його заголовка. Крім того, у C++ область видимості лічильника циклу простирається від його оголошення або до кінця функції (Visual C++), або до кінця тіла циклу (CodeWarrior). При роботі з різними компіляторами необхідно пом'ятати про цю обставинц, інкаше виникнуть помилки, зв'язані або з повторним визначенням ідентифікатора, або з відсутністю його визначення. Природно, при цьому лічильник зберігає своє останнє значення, що він набув усередині циклу. Крім того, значення лічильника циклу можна змінювати усередині його тіла.

В операторі `for` не обов'язково обчислювати усі вирази, що входять у заголовок. Відсутність умови циклу інтерпретується як істинний вираз, тому в цьому випадку оператор циклу стає нескінченним. З іншого боку, відсутність ініціалізації означає, що початкове значення лічильника визначене до циклу.

Відсутність ініціалізації

```
#include <stdio.h>

int main()
{
   char i = 33;
   for(;i<=122;i++)
   {
      printf("%c",i);
      i=122;
   }
}
```

Пропустимо умову `i<=122`. Тепер програма зациклиться, причому, з огляду на те, що переповнення інтегральної змінної також має властивості циклічності, це приведе до повторюваного виводу всіх символів.

Відсутність умови виходу

```
#include <stdio.h>

int main()
{
   char i = 33;
   for(;;i++)
   {
     printf("%c",i);
   }
}
```

Відсутність усіх виразів означає нескінченний цикл.

Нескінченний цикл

```
int main()
{
   for(;;);
}
```

Зверніть увагу на те, що в даному випадку тіло циклу складається з одного-єдиного порожнього оператора `;`.

Крім порожнього і нескінченного оператора `for`, часто використовується його різновид, у якому для одночасної ініціалізації декількох лічильників використовується оператор послідовного виконання («кома»). Як приклад розглянемо цикл, що синхронно виводить на екран значення двох лічильників.

Синхронні лічильники

```
#include <iostream.h>

int main()
{
   int x,y;
   for(x=1, y=5; y-x > 0; x++,y--)
   cout << x << " " << y << endl;
}
```

Результат роботи цієї програми виглядає так.

```
1 5
2 4
```

Як бачимо, лічильник може як збільшуватися, так і зменшуватися. Крім цього, умовний вираз не обов'язково являє собою перевірку значення лічильника. Ця умова може формулюватися за допомогою будь-якого оператора порівняння або логічного оператора. З огляду на те, що нульові і ненульові значення арифметичних виразів також можуть інтерпретуватися як «істина» і «хибність», можна сказати, що як умову виходу з оператора циклу допускається будь-який вираз.

5.7.2. Цикл while

Ще одним найбільш розповсюдженим циклом є конструкція `while`.

`while (`*`умова`*`)` *`оператор`*`;`

Як *умова оператора* може використовуватися будь-який вираз. Цикл виконується, поки його умова не стане помилковою. У цьому випадку програма передасть керування оператору, що стоїть після оператора `while`.

Природним застосуванням цього циклу є ситуація, коли програма повинна виконуватися, поки користувач не введе ознаку ее закінчення. У даному випадку програма вводить і відразу виводить на екран символи і завершує роботу, якщо користувач увів символ `'0'`.

Застосування циклу while: перший приклад

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

int main()
{
   char c;
   while ((c=getch())!='0')printf("%c\n",c);
   return 0;

}
```

Оператор `while` є циклом з попередньою перевіркою умови. Якщо ця умова є помилковою, його тіло не буде виконано ніколи.

Умова виходу з циклу `while` може залежати від значень, що обчислюються усередині його тіла. Отже, має сенс використовувати як умову виходу з циклу змінну, що змінює своє значення усередині циклу — деякий індикатор події. Розглянемо програму, що припиняє свою роботу, як тільки користувач уведе п'ять парних чисел.

Застосування циклу while: другий приклад

```
 #include <iostream.h>

int main()
{
   int i = 5, j;
   while (i)
   {
      cin >> j;
      if(j%2 == 0) i--;
      cout <<j;
   }
   return 0;

}
```

Тіло циклу `while` може бути порожнім. Наприклад, із приведеної раніше програми можна виділити оператор `while ((c=getch())!='0')`, що очікує введення символів `'0'`.

5.7.3. Цикл do–while

Оператор `do-while` є циклом з наступною перевіркою умови. Таким чином, його тіло завжди виконується хоча б один раз. Загальний вид цього оператора виглядає так.

```
   do
  {
    оператор;
  } while(умова);
```

Цикл `do–while` повторюється, поки *умова* є істинною.

Проілюструємо застосування оператора `do–while` наступним прикладом.

Застосування оператора do-while

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

```
int main()
{
   char c = '0';
   do
   {
     printf("%c\n",c);
   } while ((c=getch())!='0');
   return 0;
}
```

Ця програма, на відміну від попередньої, принаймні один раз виводить на екран символ `'0'`. В іншому логіка циклу залишається незмінною.

5.8. Оператори переходу

Мова C++ має чотири оператори безумовного переходу: `return`, `goto`, `break` і `continue`. Застосування операторів `return` і `goto` нічим не обмежене, крім стилістичних вимог. У той же час оператори `break` і `continue` самостійного значення не мають — вони нерозривно зв'язані з оператором `switch` і операторами циклів.

5.8.1. Оператор return

Даний оператор `повертає` результат у точку виклику функції. Ми уже досить часто зустрічали його у функції `main()`. У відповідності зі стандартом мови C++, функція `main()` повинна повертати операційній системі ціле значення, що служить ознакою успішного завершення програми.

Оператор `return` має наступний вид.

`return` *`вираз`*`;`

Кількість операторів `return` усередині функції нічим не обмежена. Варто лише мати на увазі, що їх виконання припиняє роботу функції. Крім того, вираз в операторі `return` може бути відсутнім. У цьому випадку функція повертає керування в точку виклику.

5.8..2. Оператор break

Розглянутий оператор застосовується для припинення виконання оператора `case` усередині оператора `switch`, а також для безумовного виходу з тіла циклу і передачі керування наступному оператору. Перший варіант ми уже розглянули, описуючи оператор `switch`, тому зосередимося тепер на другому варіанті. Розглянемо програму, що повинна припиняти виконання циклу і передавати керування зовнішньому циклу при виконанні деякої умови.

Застосування оператора break

```
 #include <iostream.h>

int main()
{
   for(int i = 0; i<1; i++)
   {
       for(int j = 0; j<2; j++)
       {
           for(int k = 0; k<3; k++)
             {
                if(k>0)break;
                cout << "k" << endl;
           }
           cout << "j" << endl;
       }
       cout << "i" << endl;
   }
   return 0;
}
```

Вона виводить на екран наступні результати.

```
k
j
k
j
i
```

Як тільки лічильник k у внутрішньому циклі стає додатним, програма виконує оператор break, передаючи керування в зовнішній цикл.

5.8.3. Оператор continue

Описуваний оператор аналогічний оператору break, але наслідки його виконання більш «легкі». У той час як оператор break припиняє виконання всього циклу, оператор continue лише зупиняє виконання його поточної ітерації, залишаючи керування усередині циклу. У циклі for при цьому виконується чергова перевірка умови і збільшення лічильника циклу. У циклах while і do–while керування передається операторам, що виконують перевірку умови.

Повернемося до попередньої програми. Якщо нам потрібно припинити поточне виконання і перейти на наступну ітерацію, досить виконати оператор continue.

Застосування оператора continue

```
#include <iostream.h>

int main()
{
   for(int i = 0; i<1; i++)
   {
       for(int j = 0; j<2; j++)
       {
           for(int k = 0; k<3; k++)
           {
               if(k==0)continue;
               cout << "k" << endl;
           }
           cout << "j" << endl;
       }
       cout << "i" << endl;
   }
   return 0;
}
```

Ця програма виводить на екран наступні результати.

```
k
k
j
k
k
j
i
```

Як бачимо, оператор continue заблокував одне виконання оператора висновку змінної k.

5.8.4. Оператор goto

Цей багатостраждальний оператор уже багато років є предметом гонінь з боку занадто академично налаштованих програмістів.

Загальний вид оператора goto такий.

goto *мітка*;

...

мітка: *оператор*;

Зрозуміло, великий вибір керуючих конструкцій, передбачених у мові C++, дозволяє уникнути його застосування. І все ж таки, як не дивно, основною претензією до цього оператора є зниження наочності програми. Що на це сказати? Якщо цей оператор такий поганий, навіщо його залишати в стандарті мови? Або існують ситуації, коли без нього не обійтися, або в деяких випадках він більш ефективний, чим інші оператори безумовного переходу. Розглянемо наступну програму.

Застосування оператора goto

```
#include <iostream.h>

int main()
{
   for(int i = 0; i<1; i++)
```

```
    {
        for(int j = 0; j<2; j++)
        {
            for(int k = 0; k<3; k++)
            {
                cout << "k" << endl;
            }
            cout << "j" << endl;
        }
        cout << "i" << endl;
    }
    return 0;
}
```

Вона виводить на екран наступні результати.

```
k
k
k
j
k
k
k
j
i
```

Якщо ж необхідно вийти в зовнішній цикл, ми можемо скористатися оператором `break`.

Застосування оператора break для виходу в зовнішній цикл

```
#include <iostream.h>

int main()
{

    for(int i = 0; i<1; i++)
    {
        for(int j = 0; j<2; j++)
        {
            for(int k = 0; k<3; k++)
              {
                 cout << "k" << endl;
                 break;
            }
            cout << "j" << endl;
        }
        cout << "i" << endl;
    }
    return 0;
}
```

Ця програма виводить на екран наступні результати.

```
k
j
k
j
i
```

Тепер тіло найбільш глибоко вкладеного циклу виконується не більше одного разу. Якщо потрібно вийти в зовнішній цикл, доведеться вставити ще один оператор `break`. А щоб вийти «на волю» за межі усіх вкладених циклів, потрібно виконати оператор `break` ще раз.

Застосування оператора break для виходу з внутрішніх циклів

```
 #include <iostream.h>

int main()
{
    for(int i = 0; i<1; i++)
    {
        for(int j = 0; j<2; j++)
        {
            for(int k = 0; k<3; k++)
            {
                 cout << "k" << endl;
```

```
                break;
            }
            cout << "j" << endl;
            break;
        }
        cout << "i" << endl;
        break;
    }
    return 0;
}
```

Програма виводить на екран наступні результати.

```
 k
j
i
```

Однак при цьому кожен цикл виконується хоча б один раз. У той же час за допомогою оператора `goto` можна було б вийти за межі всіх циклів і перейти до виконання оператора, що є наступним за ними.

Застосування оператора goto для виходу з внутрішніх циклів

```
#include <iostream.h>

int main()
{

    for(int i = 0; i<1; i++)
    {
        for(int j = 0; j<2; j++)
        {
            for(int k = 0; k<3; k++)
            {
                  cout << "k" << endl;
                  if(k==0) goto a;
            }
            cout << "j" << endl;
          }
        cout << "i" << endl;
    }
    a:;
    return 0;
}
```

Тепер на екран виводиться одна-єдина буква k. Як бачимо, логика програми зовсім змінилася.

Для оператора `goto` необхідна *мітка*, що представляє собою ідентифікатор із двокрапкою. Слід зазначити, що мітка повинна знаходитися де завгодно в межах тієї ж функції, що й оператор `goto`, оскільки переходи з функції у функцію заборонені.

5.8. Функція exit()

У деяких випадках виконання програми продовжувати небезпечно. Якщо відбулася подія, що може привести до катастрофи, розумніше припинити виконання програми взагалі. Якщо це відбулося у функції `main()`, варто виконати оператор `return`, задавши як значення, що повертається, ознаку помилки. Якщо ж помилка відбулася в іншій функції, передача керування в точку виклику може виявитися нездійсненною. У цьому випадку варто виконати функцію `exit()` і передати керування операційній системі.

Прототип функції `exit()` визначений у заголовному файлі `<stdlib.h>`.

```
void exit(int ознака);
```

Значення змінної *ознака* передається операційній системі. Якщо ця змінна дорівнює нулю, виходить, програма завершена успешно. Інші значення позначають помилку. У мові C++ як ознаку помилки можна застосовувати макроси `EXIT_SUCCESS` і `EXIT_FAILURE`. Функція `exit()` використовує заголовний файл `stdlib.h` чи заголовок `<cstdlib>`.

Виклик функції exit( )

```
 #include <iostream.h>

int main()
{
   int x, y;
```

```
   cin >> x;  // Уведення змінної x
   cin >> y;  // Уведення змінної y
   if(y == 0) { cout << "exit"; exit(EXIT_FAILURE); }

   return 0;
}
```

5.8.6. Функція abort()

Виконання програми можна припинити за допомогою функції `abort`. Ніяких додаткових умов для цього не потрібно. Прототип функції `abort` у заголовному файлі `<stdlib.h>` виглядає в такий спосіб.

```
void abort(void);
```

Виклик функції abort()

```
#include <iostream.h>

int main()
{
   int x, y;
   cin >> x;  // Уведення змінної x
   cin >> y;  // Уведення змінної y
   if(y == 0) { cout << "abort"; abort(); }

   return 0;
}
```

5.8..7. Функція atexit()

Програміст може передбачити дії, що програма повинна виконати при нормальному завершенні своєї роботи. Для цього в стандарті мови C++ передбачена функція `atexit()`. Її прототип у заголовному файлі `<stdlib.h>` виглядає в такий спосіб.

```
int atexit(void (*p)(void));
```

Функція `atexit()` одержує вказівник *p* на функцію, яку необхідно виконати при нормальному завершенні програми. Якщо при виході необхідно виконати декілька функцій, варто передбачити декілька викликів функції `atexit()`. У цьому випадку зареєстровані функції будуть викликатися в зворотному порядку. Кількість викликів таких функцій не повинне перевищувати число 32. Зверніть увагу на те, що функції, викликувані при виході з програми, не повинні мати аргументів і не повинні нічого повертати.

Виклик функції atexit()

```
 #include <iostream.h>
#include <stdlib.h>

void message1();
void message2();

int main()
{

   atexit(message1);
   atexit(message2);

   return 0;
}

void message1()
{
   cout << "До побачення: 1" << endl;
}

void message2()
{
   cout << "До побачення: 2" << endl;
}
```

При нормальному завершенні програми на екрані з'являться наступні рядки.

```
 До побачення: 2
 До побачення: 1
```

Підкреслимо ще раз, при аварійному завершенні програми (за допомогою функцій `abort()` чи `exit()`), функції, зареєстровані функцією `atexit()`, не викликаються.

5.9. Резюме

- Щоб виконувати складні обчислення, необхідні засоби керування і способи організації циклів. Оператори, що забезпечують такі можливості, називаються *керуючими*. З цим поняттям тісно зв'язана концепція *керуючої конструкції*, що складається з керуючого оператора і сукупності зв'язаних з ним операторів.
- Однією з найпростіших конструкцій, необхідних для організації керування потоком виконання програми, є *складений* оператор, чи *блок*. Складений оператор — це набір операторів, розглянутий як єдине ціле. У мові C++ для організації блоків використовують фігурні дужки.
- Область видимості всіх локальних змінних, оголошених усередині блоку, обмежена його рамками.
- *Оператор послідовного обчислення* зв'язує декілька виразів в одне ціле. Символом цього оператора є кома. Значенням цього оператора є значення останнього оператора, що стоїть в правій частині. Наприклад, оператор `var1 = (var2=100, var1+var2);` спочатку привласнює змінної `var2` значення 100, потім обчислює суму змінних `var1` і `var2` і копіює результат у змінну `var1`.
- Конструкція «якщо ..., то ...» у мові C++ реалізується за допомогою *оператора розгалуження*, чи *умовного оператора*. Він надає програмісту засоби перевірки умови і виконання обчислень у залежності від результату перевірки. Існує два різновиди операторів розгалуження: двохваріантні і різноманітні.
- Одноваріантний оператор розгалуження `if`. Він має наступний вид.
  `if (вираз) оператор;`
- Як *вираз* може застосовуватися будь-який вираз, результат якого може інтерпретуватися як «істина» чи «хибність». Якщо вираз помилковий, тобто дорівнює нулю, *оператор* не виконується. У противному випадку оператор виконується.
- Блок дозволяє виконувати не тільки один, але і відразу декілька операторів. Таким чином, оператор `if` допускає наступний варіант.

```
if (вираз)
{
  оператор_1;
  оператор_2;
   ...
  оператор_N
 }
```

- Повноцінний двохваріантний оператор розгалуження створюється за допомогою ключового слова `else`.

```
if(вираз)
   оператор_1;
else
   оператор_2;
```

Його семантика полягає в тім, що *оператор_1* виконується, тільки якщо значенням умовного вираз є «істина»; у противному випадку виконується *оператор_2*. Ці оператори є взаємовиключними і ніколи не виконуються одночасно.

- У мові C++ існує скорочений оператор `if-else`, що називається *тернарною альтернативою*.
  вираз3: вираз2?вираз3;
  Спочатку обчислюється *вираз1*. Потім, якщо він истинний, обчислюється *вираз2*, а якщо хибний — *вираз3*. Тернарний оператор еквівалентний наступній конструкції.

```
if(вираз1)
   вираз2;
else
   вираз3;
```

- Оператор розгалуження `switch` має вигляд

```
switch(вираз)
{
case значення: послідовність операторів; break;
case значення: послідовність операторів; break;
...
case значення: послідовність операторів; break;
default: оператор;
}
```

Спочатку обчислюється значення *вираз*. Потім воно порівнюється з кожним зі значень, що відповідають різним варіантам `case`. Значенням *вираз* повинний бути символ чи ціле число. Вираз,

результатом якого є число з крапкою, що плаває, не допускаються. Коли виявляється збіг, виконується відповідна послідовність операторів, поки не зустрінеться оператор `break` чи не буде досягнутий кінець оператора `switch`. Якщо в якому-небудь розділі оператор `break` пропущений, виконання автоматично переходить до наступного розділу `case` (фігурально виражаючи, потік керування «провалюється на нижній поверх»). Якщо значення виразу не збігається ни жодною з констант, виконується розділ `default`. Цей розділ оператора `switch` є необов'язковим. Якщо він не передбачений, під час відсутності збігів не буде виконаний жодний оператор.

- Оператор `break` відноситься до групи операторів переходу. Він використовується не тільки в операторі `switch`, але й у циклах. Коли потік керування досягає оператора `break`, програма виконує перехід до оператора, що є настпним за оператором `switch`.
- На відміну від оператора `if`, оператор `switch` порівнює значення умовного виразу винятково на рівність з константами. Інші види порівняння і логічних виразів не допускаються.
- Усі константи в розділах `case` повинні бути взаємовиключними, за винятком випадку, коли один оператор `switch` вкладений в іншій.
- Символьні константи, що використовуються в операторі `switch`, автоматично перетворюються в целочисельні.
- Якщо та сама послідовність операторів відповідає декільком міткам `case`, їх можна об'єднати.
- Оператори `switch` можуть бути вкладеними. При цьому константи розділів `case` зовнішнього і внутрішнього операторів `switch` можуть збігатися.
- *Оператори циклу* забезпечують повторне виконання обчислень. У мові C++ передбачено три види таких операторів. Вони відрізняються механізмами керування циклом та їх розташуванням. Цикл `for`, кількість повторень якого задано заздалегідь, керується лічильником і логічними виразми, розташованими в його початку. Цикл `while` виконується доти, поки не буде виконана задана логічна умова, сформульована в його початку. На противагу йому цикл `do-while` перевіряє умова виходу з циклу наприкінці.
- *Тіло* циклу — це набір операторів, що повторно виконуються. У залежності від розташування умови виходу з циклу, цикли розділяються на цикли з попередньою перевіркою (`for`, `while-do`) і на цикл із наступною перевіркою (`do-while`).
- Оператор `for` керується змінною, що називається *лічильником циклу*. Крім того, до складу механізму керування входять оператори *ініціалізації* лічильника, перевірки *умови виходу з циклу* і оператор *зміни лічильника*. Величину зміни лічильника називають *кроком*. Початкове і кінцеве значення лічильника, а також крок циклу називають *параметрами циклу*. Оскільки лічильник циклу в ході його виконання постійно змінюється, його часто оголошують як реєстрову змінну.
- Основні питання, зв'язані з циклом `for`, такі.
  1. Який тип і яку область видимості має лічильник циклу?
  2. Чому дорівнює лічильник циклу після завершення циклу?
  3. Чи змінюється значення лічильника і параметрів циклу усередині циклу?
  4. Чи обчислюються параметри циклу на кожній ітерації
- Загальний вид оператора `for` такий.

```
for (ініціалізація; умова; зміна)
```

- Першим виконується оператор ініціалізації, що задає початкове значення лічильника. Природно, ініціалізація здійснюється за допомогою оператора присвоювання. Потім перевіряється *умова виходу з циклу*. Цикл виконується доти, поки значення цього вираз залишається істинним. Оператор *зміни* змінює значення лічильника циклу *після* чергової ітерації. Він виконується *після* виконання тіла циклу. Зверніть увагу на те, що оператори керування циклом з лічильником відокремлюються крапкою з комою. Якщо умова виходу з циклу стала помилковою, керування передається оператору, розташованому після тіла циклу.
- Ще одним найбільш розповсюдженим циклом є конструкція `while`.

```
while (умова) оператор;
```

- Як *умова оператора* може використовуватися будь-який вираз. Цикл виконується, поки його умова не стане помилковою. У цьому випадку програма передасть керування оператору, що стоїть після оператора `while`. Оператор `while` є циклом з попередньою перевіркою умови. Якщо ця умова є помилковою, його тіло не буде виконано ніколи.
- Оператор `do-while` є циклом з наступною перевіркою умови. Таким чином, його тіло завжди виконується хоча б один раз. Загальний вид цього оператора виглядає так.

```
do
{
  оператор;
} while(умова);
```

Цикл `do-while` повторюється, поки *умова* є істинною.

- Мова C++ має чотири оператори безумовного переходу: `return`, `goto`, `break` і `continue`. Застосування операторів `return` і `goto` нічим не обмежене, крім стилістичних вимог. У той же час оператори `break` і `continue` самостійного значення не мають — вони нерозривно зв'язані з оператором `switch` і операторами циклів.
- Оператор `return` повертає результат у точку виклику функції.

  Оператор `return` має наступний вид.

  `return` *`вираз`*;
- Кількість операторів `return` усередині функції нічим не обмежена. Варто лише мати на увазі, що їх виконання припиняє роботу функції. Крім того, вираз в операторі `return` може бути відсутнім. У цьому випадку функція повертає керування в точку виклику.
- Оператор `break` застосовується для припинення виконання оператора `case` усередині оператора `switch`, а також для безумовного виходу з тіла циклу і передачі керування наступному оператору.
- Оператор `continue` є аналогічним оператору `break`, але наслідки його виконання більш «легкі». У той час як оператор `break` припиняє виконання всього циклу, оператор `continue` лише зупиняє виконання його поточної ітерації, залишаючи керування усередині циклу. У циклі `for` при цьому виконується чергова перевірка умови і збільшення лічильника циклу. У циклах `while` і `do-while` керування передається операторам, що виконують перевірку умови.
- Загальний вид оператора `goto` такий.

  `goto` *`мітка`*;

  ...

  `мітка`: *`оператор`*;
- У деяких випадках виконання програми продовжувати небезпечно. Якщо відбулася подія, що може привести до катастрофи, розумніше припинити виконання програми взагалі. Якщо це відбулося у функції `main()`, варто виконати оператор `return`, задавши як значення, що повертається, ознаку помилки. Якщо ж помилка відбулася в іншій функції, передача керування в точку виклику може виявитися нездійсненною. У цьому випадку варто виконати функцію `exit()` і передати керування операційній системі.
- Прототип функції `exit()` визначений у заголовному файлі `<stdlib.h>`.

  `void exit(int` *`ознака`*`);`
- Значення змінної *ознака* передається операційній системі. Якщо ця змінна дорівнює нулю, виходить, програма завершена успешно. Інші значення позначають помилку. У мові C++ як ознаку помилки можна застосовувати макроси `EXIT_SUCCESS` і `EXIT_FAILURE`. Функція `exit()` використовує заголовний файл `stdlib.h` чи заголовок `<cstdlib>`.
- Виконання програми можна припинити за допомогою функції `abort`. Ніяких додаткових умов для цього не потрібно. Прототип функції `abort` у заголовному файлі `<stdlib.h>` виглядає в такий спосіб.

  `void abort(void);`
- Програміст може передбачити дії, що програма повинна виконати при нормальному завершенні своєї роботи. Для цього в стандарті мови C++ передбачена функція `atexit()`. Її прототип у заголовному файлі `<stdlib.h>` виглядає в такий спосіб.

  `int atexit(void (*`*`p`*`)(void));`
- Функція `atexit()` одержує вказівник *p* на функцію, яку необхідно виконати при нормальному завершенні програми. Якщо при виході необхідно виконати декілька функцій, варто передбачити декілька викликів функції `atexit()`. У цьому випадку зареєстровані функції будуть викликатися в зворотному порядку. Кількість викликів таких функцій не повинне перевищувати число 32. Функції, що викликаються при виході з програми, не повинні мати аргументів і не повинні нічого повертати.

5.10. Резюме

1. Що таке керуючий оператор і керуюча конструкція?
2. Що таке складений оператор, чи блок?
3. Як діє оператор послідовного обчислення?
4. Якими операторами реалізується розгалуження в програмі?
5. Опишіть одноваріантний оператор розгалуження `if`.
6. Опишіть двохваріантний оператор розгалуження.

```
if(вираз)
  оператор_1;
else
  оператор_2;
```

7. Опишіть тернарну альтернативу.
8. Опишіть оператор розгалуження `switch`.

9. Навіщо потрібен оператор `break` в конструкції `switch`?
10. Що може слугувати мітками в розділах `case`?
11. Чи можна об'єднувати розділи `case`?
12. Чи можна вкладати оператори `switch`?
13. Назвіть оператори циклу у мові C++.
14. Що таке тіло циклу?
15. Що таке лічильник циклу?
16. Опишіть механізм дії оператора `for`.
17. Які питання пов'язані з оператором `for`?
18. Опишіть механізм дії оператору `while`.
19. Що може бути використане як умова оператора?
20. Опишіть механізм дії оператор `do-while`
21. Назвіть чотири оператори безумовного переходу.
22. Для чого призначений оператор `return`?
23. Для чого потрібен оператор `break`?
24. Опишіть відмінність оператора `continue` від оператора `break`.
25. Опишіть механізм дії оператора `goto`.
26. Опишіть функцію `exit()`. Як вона функціонує?
27. Опишіть функцію `abort()`. Як вона функціонує?
28. Опишіть функцію `atexit()`. Як вона функціонує?