Робота з функціями в С++ Класи пам’яті

Проф. Куссуль Н.М.

Функція main()

- Кожна програма у своєму складі повинна мати функцію `main()`. Саме функція `main()` забезпечує створення **точки входу** в об'єктний модуль.
- Крім функції `main()` у програму може входити будь-яка кількість функцій.
- Кожна функція по відношенню до іншої є зовнішньою.
- Для того щоб функція була доступна, необхідно, щоб **до її виклику** про неї було **відомо компілятору**.

Використання функцій

- З поняттям функції в мові C++ пов'язано три наступних компоненти:

  1) визначення (опис) функції
  2) прототип функції
  3) виклик функції

Визначення функції 1/3

- **Визначення функції** складається з двох частин
  - заголовка функції
  - тіла функції
- Визначення функції має наступну форму запису:

```
/* заголовок функції */
тип_результату <ім'я>([параметри])
{
   /* оголошення й оператори */
   тіло_функції
}
```

Визначення функції 2/3

- **Тип результату** — це тип значення, яке повертається.
- Якщо функція не повертає ніякого значення, то на місці типу записується специфікатор **`void`**.
- У **списку параметрів** для кожного параметра повинен бути зазначений **тип**.
- При відсутності параметрів список може бути порожнім або мати специфікатор `void`.

Визначення функції 3/3

- **Тіло функції** являє собою послідовність оголошень і операторів, які описують визначений алгоритм.
- Важливим оператором є оператор повернення в місце виклику: **`return (вираз)`**.
- Оператор `return` має подвійне призначення.
  - Він забезпечує негайне повернення у викликаючу функцію і може використовуватися для передачі обчисленого значення функції.
- У тілі функції може бути декілька операторів `return`, а може не бути жодного. В останньому випадку повернення у викликаючу програму відбувається після виконання останнього оператора тіла функції.

Прототип функції 1/2

- **Прототип функції** – це її **оголошення**, але не визначення
- Прототип функції задається до її виклику замість опису функції для того, щоб компілятор міг виконати перевірку відповідності типів аргументів і параметрів.
- Прототип функції за формою такий же, як і заголовок функції, в кінці його ставитися «**;**».
- Параметри функції в прототипі можуть мати імена, але для компілятора не є обов'язковими.

Приклад

```
#include<iostream.h>
#include "d:\max.cpp"
//включення файлу max.cpp
//із функцією max(,)

int max(int, int);  /* прототип функції */

int main()
{
   int x, y, z;
   cout << "\nпочергово введіть x та y\n";
   cin >> x; cin >> y;
   z = max(x, y);
   cout <<"z=" << z;
   return 0;
}
```

Прототип функції 2/2

- Компілятор використовує прототип функції для порівняння типів аргументів з типами параметрів.
- Мова C++ не передбачає автоматичного перетворення типів у випадках, коли аргументи не співпадають за типами з відповідними їм параметрами, тобто мова C++ забезпечує строгий контроль типів.
- При наявності прототипу функція, яка викликається, не обов'язково розміщається в одному файлі з функцією, що її викликає.

Виклик функції 1/2

- Виклик функції може бути оформлений у вигляді оператора, якщо відсутнє значення, що повертається, або у вигляді виразу, якщо існує значення, що повертається.

```
ім'я_функції(список_аргументів);
```

```
f(x);   //не повертає значення
h=f(x);  //повертає значення
```

- Значення обчисленого виразу є значенням функції, що повертається.
- Значення, що повертається, передається в місце виклику функції і є результатом її роботи.

Виклик функція 2/2

- Параметри, які описуються у визначенні функції, називаються **формальними**, а параметри, які використовуються при виклику функції, називаються **фактичними**.
- **Типи фактичних** параметрів повинні **відповідати типам формальних** параметрів.

Приклад 1/2

- Програма, яка звертається до функції підрахунку максимуму двох чисел
- Файл `max.cpp` знаходиться в кореневому каталозі диска `d:` і має наступний вигляд:

```
int max(int a, int b)
{
   int c; /* робоча змінна */
   if (a>=b) c=a;
   else c=b;
   return c;
}
```

Приклад 2/2

```
#include<iostream.h>
#include "d:\max.cpp" // непотрібно
//включення файлу max.cpp
//із функцією max(,)

int max(int, int);  /* прототип функції */

int main()
{
   int x, y, z;
   cout << "\nпочергово введіть x та y\n";
   cin >> x; cin >> y;
   z = max(x, y); // виклик функції
   cout <<"z=" << z;
   return 0;
}
```

Аргументи за замовчуванням

- Значення формальних параметрів можуть бути задані за замовчуванням
- Зазвичай це константа, яка часто зустрічається при виклику функції
- Значення аргументів за замовчуванням задаються **справа наліво**, починаючи від самого правого
- Приклади
  - `void foo(int i, int j = 7); // коректно`
  - `void foo(int i = 3, int j); // некоректно`
  - `void foo(int i, int j = 7, int k = 8); // коректно`
  - `void foo(int i = 1, int j = 7, int k = 8); // коректно`
  - `void foo(int i, int j = 7, int k); // некоректно`

Перевантаження функцій

- Зазвичай назва функції відображає її основне призначення.
- **Перевантаження** - використання однієї й тієї ж назви для декількох операторів або функцій.
- Вибір варіанту залежить від типу аргументів оператора або функції.
- **Приклад**
  - `int max(int, int);`
  - `double max(double, double);`

Вбудовані функції (inline)

- В C++ забезпечується ключовим словом `inline`.
- Воно вказується перед оголошенням функції, коли необхідно, щоб код в тілі функції **вбудовувався в місце виклику функції**.

```
inline double cube (double x)
{
   return (x*x*x);
}
```

- Обмеження компілятора не дозволяє вбудовувати складні функції

Рекурсивні функції

- Для реалізації рекурсивних алгоритмів у C++ передбачена можливість створення рекурсивних функцій.
- **Рекурсивна функція** являє собою функцію, у тілі якої здійснюється **виклик цієї ж функції**.
- В рекурсивних функціях зазвичай використовується **кілька** операторів **return**
- Така необхідність виникає при реалізації динамічних структур даних, таких як стеки, дерева, черги.

Приклад

- Обчислення факторіалу

```
int fact(int n)
  {
     int a;
     if (n < 0) return 0;
     if (n == 0) return 1;
     a = n*fact(n-1);
     return a;
  }
```

Область видимості та клас пам’яті

- Два види області видимості
  - локальна: область видимості – блок
    - відноситься до блоку
    - приклад: тіло функції, тіло циклу
  - глобальна: область видимості - файл
- Основне правило
  - ідентифікатори доступні лише всередині блоку, в якому вони оголошені

Приклад

```
{
    int  a = 2;   //зовнішній блок
    cout << a << endl; // виведе 2
    { // внутрішній блок
      int  a = 7;// інша змінна a
      cout << a << endl;// виведе 7
    } // exit inner block
    cout << ++a << endl;
    //виведе ???
}
```

Класи пам’яті

- Кожна змінна і функція в C++ має два атрибути
  - тип
  - клас пам’яті
- Існує чотири класи пам’яті
  - **автоматичний** – `auto`
  - **зовнішній** – `extern`
  - **регістровий** – `register`
  - **статичний** – `static`

Автоматичний

- Змінна, оголошена **всередині функції**, за замовчуванням є **автоматичною**
- Оголошення змінних всередині блоку **неявно** отримують **автоматичний клас** пам'яті
- Приклад (явного використання)
  - `auto int a, b, c;`
  - `auto float d = 5.38`
- Система виділяє пам'ять для автоматичних змінних при вході в блок **в стеку**

Клас пам'яті `register`

- Відповідні змінні буде розміщено **у швидких регістрах процесора** (при наявності такої можливості)
- Якщо компілятор не може виділити фізичний регістр – клас пам'яті стає автоматичним
-
```
for(register i = 0; i < LIMIT; i++)
{
    …
}
```
- Оголошення `register i = 0;` рівнозначно `register int i = 0;`

Клас пам’яті `extern`

- Спосіб **передачі інформації між блоками** та функціями – використання зовнішніх змінних
- Якщо змінна оголошена **ззовні функції** – її клас пам’яті `extern`

Приклад

- circle3.cpp

```
const double  pi = 3.14159; //зовнішня змінна до
                                    //circle.cpp
double circle(double radius)
  {
     return (pi * radius * radius);
  }
```

- cir_main.cpp

```
#include  <iostream.h>
double circle(double);  //функція автоматично є
                                    //зовнішньою
int main()
{
   double  x;
   x = 3.5;
   cout << endl;
   cout << circle(x) << “площа кола радіусу" <<
x;
}
```

Клас пам’яті `static` 1/3

- Два важливих застосування
  - локальна змінна **зберігає значення** при повторному вході в блок
  - забезпечення механізму **закритості** (privacy)
    - **обмеження видимості** змінних або функцій
    - важливе для модульності програм

Клас пам’яті static 2/3

- Приклад зберігання значення змінної

```
#include <iostream.h>

int f()
{
   static int  called = 0;
   ++called;
   return called;
}

main()
{
   int call_times;

   for (int i = 0; i < 5; ++i)
      call_times = f();

   cout << "\nThe function is called " <<
call_times << " times\n";
}
```

Клас пам’яті static 3/3

- Приклад забезпечення закритості
  - ```
    static int goo(int a)
    {
       ...
    }

    int foo(int a)
    {
       ...
       b = goo(a); //доступно лише в цьому файлі; в інших – не доступно
       ...
    }
    ```
    ```