Лекція 6

Функції

У цій главі...

6.1. Прототипи і визначення

Будь-яка програма на C++ складається з функцій. Навіть найпростіша програма містить функцію `main()`. Це цілком відповідає модульному принципу процедурного програмування, відповідно до якого вихідна задача повинна бути розбита на декілька більш спеціалізованих підзадач. Рішення цих конкретних задач покладається на функції. Розглянемо спочатку основні поняття, зв'язані з функціями.

6.1..1. Основні поняття

Кожна функція повинна мати оголошення і визначення. Оголошення функції називається її прототипом. Загальний вид прототипу виглядає в такий спосіб.

```
тип_значення_що_повертається ім'я_функції(тип_параметра1,
                                          ... ,
                                          тип_параметра N);
```

Будь-яку функцію можна подати у вигляді “чорної шухляди”, у який передається список фактичних аргументів, можливо порожній, а у відповідь повертається обчислене значення зазначеного типу. У мові C++ функція може повернути тільки одне значення.

Список типів, зазначений у прототипі функції, являє собою перерахування типів її формальних параметрів, розділених комами. Якщо функція нічого не одержує ззовні, цей список порожній. Імена параметрів у прототипі вказувати необов'язково. Крім того, типи параметрів можна модифікувати ключовим словом `const`. Іноді ім'я змінної допомагає швидше знайти помилку, що виникла при компіляції програми. Зверніть увагу на те, що кожний формальний параметр функції повинний бути оголошений окремо. Кількість, порядок і типи формальних параметрів утворять *профіль* параметрів функції. У свою чергу профіль і тип значення, що повертається, є *протоколом* функції. Не забувайте: прототип завжди повинний завершуватися точкою з комою! Ім’я і профіль функції утворюють її *сігнатуру*.

Для реалізації функції необхідно виконати її визначення, що має наступний вид.

```
тип_значення_що_повертається ім'я_функції(
        тип_параметра1 ім'я_параметра1,
        ... ,
        тип_параметра ім'я_параметра)
{
    тіло функції
}
```

Тип значення, що повертається, ім'я функції і список формальних параметрів *утворюють заголовок* функції. Прототип і заголовок функції повинні збігатися.

Фігурні дужки обмежують *тіло функції*, що складає з операторів, що розв’язують поставлену задачу. Крім того, фігурні дужки є межами області видимості всіх змінних, оголошених усередині тіла функції (такі змінні називаються *локальними*). Ця область закрита від зовнішнього світу, і доступ до неї з інших функцій неможливий. Вся інформація може надходити ззовні лише по двох каналах: через фактичні параметри, чи *аргументи*, що представляють собою значення формальних параметрів, і через глобальні змінні, які є видимими в будь-якій точці програми.

Локальні змінні існують лише під час виконання функції. Вони створюються при її виклику і знищуються при поверненні керування в модуль, звідки зроблено виклик. Однак існує можливість зберегти значення локальних змінних між викликами функцій — оголосити такі змінні статичними (див. лекцію 3, розділ 3.3.1).

У мові C++, на відміну від мови Pascal, вкладені функції не допускаються.

Виклик функції виконується оператором `()`. Для цього необхідно перед дужками вказати ім'я функції, а усередині дужок — список аргументів.

Як уже відзначалося вище, змінні, оголошені в заголовку функції, називаються *параметрами* функції. Вони призначені для того, щоб приймати значення аргументів. Як і локальні змінні, вони створюються при виклику функції і знищуються при виході з неї. Розглянемо приклад, що ілюструє поняття виклику, параметрів і аргументів.

Виклик функції

```
#include <iostream.h>
int maxint(int, int);  // Прототип

int main()
{
   int var1, var2;
   cin >> var1; // Введення аргументу var1
   cin >> var2; // Введення аргументу var2
   int maxvar = maxint(var1,var2); // Виклик функції maxint()
   cout << maxvar; // Висновок значення, повернутого функцією maxint()
   return 0;
}

int maxint(int var1, int var2) // Заголовок функції
{
   return var1>=var2 ? var1 : var2;    // Тіло функції
}
```

Функція `maxint()` має два параметри: `var1` і `var2`. Вона повертає максимальне з двох отриманих значень. Як бачимо, дана функція не створює власних локальних змінних. Замість цього вона оперує формальними параметрами. Помітимо також, що в даному випадку функція повертає цілочисельне значення, отже, вона може знаходитися там, де очікується ціле число: у правій частині оператора присвоювання, як у даному варіанті, чи в оголошенні цілочисельної змінної для її ініціалізації. Наприклад, програму можна було б спростити, виключивши проміжну змінну `maxvar`. Замість цього повернуте значення можна було б відразу вивести на екран оператором `<<`.

```
cout << maxint(var1, var2);
```

6.1..2. Функція main()

Розглянемо тепер дуже важливу функцію `main()`, що служить диспетчером і організує взаємодію між усіма функціями. Будучи головною, ця функція має усі властивості звичайних функцій. Зокрема, вона має заголовок, список формальних параметрів, тіло, значення, що повертається, — практично всі атрибути функції, за винятком прототипу.

Список формальних параметрів функції `main()` фіксований. Він може складатися лише з зазначених параметрів, кількість яких обмежується можливостями операційної системи (як правило, це число дорівнює 32767). Аргументи функції `main()` повинні вказуватися в командному рядку слідом за ім'ям модуля, що виконується.

Для витягу аргументів командного рядка використовуються убудовані параметри `argv`, `argc` і `env`. (У деяких компіляторах останній параметр називається `envp`.) Параметр `argc` є цілочисельним. Він задає кількість аргументів, зазначених у командному рядку. Ім'я програми вважається першим аргументом. Отже, якщо користувач не задав жодного власного аргументу, значення змінної `argc` буде дорівнювати 1. Параметр `argv` є вказівником на масив символьних вказівників, що посилаються на аргументи командного рядка. Варто мати на увазі, що всі аргументи командного рядка є рядками і за необхідності повинні перетворюватися в числа за допомогою відповідних функцій. Спробуємо переробити попередню програму, використовуючи аргументи командного рядка.

Аргументи командного рядка

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

int maxint(int, int);
int main(int argc, char* argv[], char* env[])
{

   if(argc != 3) { cout << "Помилка"; return 1;}
   cout << maxint(atoi(argv[1]),atoi(argv[2]));
   return 0;
}
```

```
int maxint(int var1, int var2)
{
   return var1>=var2 ? var1 : var2;
}
```

Тепер для обчислення максимального з двох цілих чисел досить набрати в командному рядку ім'я програми `maxint` і вказати два цілочисельних параметри, розділених пробілами чи знаками табуляції (інші роздільники не допускаються).

Зверніть увагу на те, що `argv` допускає два види оголошення: `char *argv[];` чи `char** argv;`. Обидва способи еквівалентні, оскільки змінна `argv` є вказівником на масив вказівників.

На закінчення зауважимо, що передача параметрів з командного рядка використовується досить рідко. Як правило, функція `main()` має порожній список параметрів: `main()`.

Параметр `env` являє собою масив вказівників на рядки оточення операційної системи. Природно, його реалізація залежить від конкретного середовища. Одержати вказівники на рядки оточення можна за допомогою функції `getenv()`, а задати їх — за допомогою функції `putenv()`. Їхні оголошення знаходяться в заголовному файлі `stdlib.h`.

6.2. Передача аргументів за значенням

У мові C++ є два способи передачі аргументів у функцію: *за значенням* і *за посиланням*. За замовчуванням, якщо аргументом є не масив, застосовується перший спосіб. Для цього створюється копія значення аргументу, що привласнюється формальному параметру. Всі операції, виконані усередині функції, стосуються лише копії аргументу і не впливають на оригінал, що існує в модулі, що здійснює виклик.

Проілюструємо сказане наступною програмою.

Передача аргументів за значенням

```
#include <iostream.h>

int twice(int);

int main()
{
   int actual=1, result;
   result = twice(actual);
   cout << actual;
   return 0;
}

int twice(int formal)
{
   return 2*formal;
}
```

Функція `twice()` подвоює свій формальний параметр `formal`, не змінюючи аргумент `actual`, що після виклику зберігає своє колишнє значення.

6.3. Передача аргументів за посиланням

Другий спосіб, *передача аргументів за посиланням*, означає, що у функцію передається не копія аргументу, а його адреса. У цьому випадку функція одержує безпосередній доступ до аргументу, а формальний параметр збігається з фактичним.

Для передачі аргументів за посиланням передбачено два механізми.

6.3..1. Вказівник як параметр функції

Функції можна передати вказівник на аргумент. Передача вказівника відбувається традиційно, тобто у функції створюється копія вказівника на аргумент. Розглянемо приклад.

Передача вказівника на аргумент: перший варіант

```
#include <iostream.h>

int twice(int*);

int main()
{
   int actual=1, result;
   int* pActual=&actual;
   cout << "У функції main : " << pActual << endl;
```

```
    result = twice(pActual);
    cout << "Аргумент: " << actual;
    return 0;
}

int twice(int* pFormal)
{
  *pFormal = 2*(*pFormal);
   cout << "У функції twice: " << pFormal << endl;
   return *pFormal;
}
```

Функція `twice()` тепер одержує копію вказівника на аргумент, розіменовує цей вказівник, подвоює і повертає його значення. У цьому легко переконатися, проконтролювавши значення вказівника. Крім того, первісне значення аргументу змінилося.

```
    У функції main:  0x0065FDF4
 В функції twice: 0x0065FDF4
 Аргумент: 2
```

У принципі вказівник на аргумент створювати не обязательно. Необходимо лише указати відповідний параметр у прототипі.

Передача вказівника на аргумент: другий варіант

```
#include <iostream.h>

int twice(int*);

int main()
{
   int actual=1, result;
   cout << "У функції main : " << pActual << endl;
   result = twice(&actual);
   cout << "Аргумент: " << actual;
   return 0;
}

int twice(int* pFormal)
{
   *pFormal = 2*(*pFormal);
   cout << "У функції twice: " << pFormal << endl;
   return *pFormal;
}
```

6.3..2. Посилання як параметр функції

Ще один механізм заснований на застосуванні посилань. Для цього перед ім'ям аргументу в прототипі функції і її заголовку (вони завжди повинні збігатися!) слід поставити оператор узяття адреси. Такий прототип буде означати, що аргумент передається за посиланням, хоча визначення функції зовсім не відрізняється від передачі за значенням (ніяких розіменувань не потрібно).

Передача параметра за посиланням

```
#include <iostream.h>

int twice(int&);

int main()
{
   int actual=1, result;
   result = twice(actual);
   cout << actual;
   return 0;
}

int twice(int& formal)
{
   return 2*formal;
}
```

6.3..3. Одномірний масив як параметр функції

Вище ми уже відзначали, що передача масиву у функцію є виключенням із загальноприйнятих правил. За замовчуванням масив передається за посиланням. Функція, параметром якої є масив, може модифікувати будь-які його елементи.

Передача одномірного масиву

```
#include <iostream.h>

void twiceArray(int [],int);

int main()
{
   int actual[]={1,2,3,4,5};
   int size= sizeof(actual)/sizeof(actual[0]);
   cout << size << endl;
   twiceArray(actual,size);
   for(int i=0; i<size; i++) cout << actual[i] << " ";
   return 0;
}

void twiceArray(int formal[], int n)
{
   for (int i = 0; i < n; i++) formal[i]*=2;
}
```

Ця програма ілюструє ряд особливостей, зв'язаних з масивами. По-перше, зверніть увагу на те, як у прототипі зазначений масив: `int []`. (Пробіл між словом `int` і дужками не обов'язковий.) По-друге, розмір масиву задається окремим цілочисельним параметром. Його значення задається не явно, а обчислюється за допомогою виразу `size=sizeof(actual)/sizeof(actual[0])`. Значення `sizeof(actual)` дорівнює кількості байтів, займаних масивом, а значення `sizeof(actual[0])` — кількості байтів, займаних його елементами. Отже, частка `sizeof(actual)/sizeof(actual[0])` дорівнює кількості елементів масиву. Спроба обійтися без цього й обчислити розмір масиву, що є формальним параметром, у тілі функції приречена на невдачу. Це зв'язано з тим, що у функцію передається не масив, а вказівник на його перший елемент. Отже, вираз `size=sizeof(formal)/sizeof(formal[0])` завжди дорівнює 1, оскільки значення `sizeof(formal)` дорівнює значенню `sizeof(formal[0])`.

6.3..4. Багатомірний масив як параметр функції

На жаль, у мові C++ немає способу, що дозволив би уникнути явного завдання розміру масиву, переданого функції. Тому передавати масив без завдання кількості рядків і стовпців неможливо. Наприклад, при передачі функції двомірного масиву необхідно визначити хоча б кількість стовпців, що задається як глобальна змінна, у той час як кількість рядків передається як параметр. Утім, якщо програміст візьме на себе обов'язок представити двомірний масив у вигляді одномірного, можна скористатися способом, описаним у попередньому розділі.

Передача двовимірного масиву

```
#include <iostream.h>

const int COLUMNS = 2;
void twiceArray(int [][COLUMNS], int rows);

int main()
{
   int actual[][COLUMNS]={1,2,3,4};
   int rows = 2;
   twiceArray(actual, rows);
   for(int i=0; i<COLUMNS; i++)
   {
     for(int j=0; j<rows; j++) cout << actual[i][j] << " ";
     cout << endl;
   }
   return 0;
}

void twiceArray(int formal[][COLUMNS], int rows)
{
```

```
    for (int i = 0; i < COLUMNS; i++)
        for(int j = 0; j < rows; j++)
            formal[i][j]*=2;
}
```

6.4. Значення, що повертається

Функції розподіляються на дві категорії. Вони можуть або повертати деяке значення, або не повертати нічого. Для повернення обчисленого значення існує два механізми. По-перше, функція може повернути значення, виконавши оператор `return`, що негайно передасть керування в модуль виклику одночасно з тимчасовою змінною, що містить результат обчислень. По-друге, функція може привласнити обчислене значення параметру, переданому за посиланням. У цьому випадку в модулі виклику можна скористатися результатами побічного ефекту.

Якщо функція не повертає ніяких значень, перед її ім'ям у прототипі і заголовку необхідно поставити ключове слово `void`. Необхідно підкреслити наступне: незважаючи на те що ключове слово `void` стоїть на місці типу значення, що повертається, це зовсім не означає, що повертається деяке значення “порожнього типу”. Навпаки, така функція нічого не повертає, і її виконання припиняється, як тільки потік керування досягає фігурної дужки, що закриває її тіло. Утім, з такої функції можна просто “катапультуватися” за допомогою оператора `return`, не вказавши ніякого значення, що повертається. По суті, така функція є процедурою. Розглянемо декілька важливих ситуацій.

6.4..1. Повернення змінних простих типів

Якщо функція обчислює значення змінної простого типу, її можна віднести до розряду обчислювальних. Як правило, це — математичні функції.

Обчислювальні функції

```
#include <iostream.h>

int signum(double);

int main()
{
   double x;
   cin >> x;
   cout << signum(x);
   return 0;
}

int signum(double x)
{
   if (x < 0) return -1;
   else if (x >1) return 1;
   return 0;
}
```

Функція `signum()` обчислює знак числа з точкою, що плаває, і повертає 0, якщо `x` дорівнює 0; -1 — якщо значення змінної x є від’ємним і +1 — якщо число `x` є додатним.

Оскільки вказівник належить до простих типів, він також може повертатися функцією. Однак тут варто виявляти обережність. Вказівник коректно повертається функцією лише в двох випадках: 1) якщо функція одержує його як аргумент і модифікує змінну, на яку він посилається; і 2) якщо функція створює динамічний масив і заповнює його даними. Грубою помилкою було б повертати вказівник на локальну змінну, створену у функції, — адже вона зникає відразу після виходу з неї!

Розглянемо наступні приклади.

Функція, що створює динамічний масив

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

int* Dynamic(int);
int main()
{
   int size;
   cin >> size;
   int* Array =Dynamic(size);
   for(int i=0;i<size; i++) cout << Array[i] << endl;
```

```
   return 0;
}

int* Dynamic(int size)
{
   int* Array = (int*) malloc(size*sizeof(int));
   for(int i=0;i<size;i++)Array[i]=i;
   return Array;
}
```

У першій програмі функція `Dynamic()` виділяє в купі масив заданого розміру і повертає в головний модуль вказівник на перший осередок виділеної пам'яті. Це дозволяє функції `main()` вивести на екран зміст цього масиву.

Друга програма одержує на вхід вказівник, що посилається на змінну, подвоює її і повертає вказівник назад.

Функція, що одержує і повертає вказівник

```
#include <iostream.h>

int* modify(int*);

int main()
{
   int var = 7;
   int* pVar = &var;
   pVar = modify(pVar);
   cout << *pVar;
   return 0;
}

int* modify(int* p)
{
   (*p)*=2;
   return p;
}
```

6.4..2. Повернення змінних складних типів

Функція може повернути будь-яку змінну складного типу чи структуру об'єднання. Повернення масиву не має сенсу, оскільки він передається у функцію за посиланням і, отже, безпосередньо модифікується нею.

Розглянемо як приклад функцію, що заповнює об'єднання структур.

Функція, що повертає об'єднання

```
#include <iostream.h>

union myUnion
{
   int ident;
   struct
   {
      int i1;
      int i;
   }First;
   struct
   {
      int i2;
      double f;
   }Second;
}Actual;

myUnion FillUnion(int, int, double);

int main()
{
   Actual = FillUnion(1,5,0);

   if(Actual.ident == 1)
      cout << Actual.First.i << endl;
   else
         cout << Actual.Second.f << endl;
```

```
    Actual = FillUnion(2,0,10.5);

    if(Actual.ident == 1)
        cout << Actual.First.i << endl;
    else
        cout << Actual.Second.f << endl;

    return 0;
}

myUnion FillUnion(int one, int two, double three)
{
    myUnion Result;
    Result.ident = one;
    if(one == 1)Result.First.i = two; else Result.Second.f = three;
      return Result;
}
```

Об'єднання `myUnion` складається з цілочисельного поля і двох структур, одна з яких містить два цілочисельних поля, а інша — цілочисельне поле і дійсне число. Функція `FillUnion()` заповнює локальну змінну типу `myUnion`, керуючись наступною логікою: якщо змінна `one` дорівнює 1, заповнюється целочисельна структура, якщо 2 — структура, що містить дійсне число. Оскільки структури перекривають ту саму область пам'яті, один з формальних параметрів стає фіктивним: у першому випадку — параметр `three`, а в другому — `two`.

Точно так само функції працюють зі структурами.

6.5. Виклик функції

Ми уже не раз демонстрували стандартний виклик функції. Спробуємо тепер розібратися в його механізмі і розглянути деякі різновиди викликів.

6.5..1. Механізм активації

Виклик функції виконується в такий спосіб: спочатку зберігається інформація, що дозволяє відновити виконання програми після повернення з функції, потім виділяється пам'ять для формальних параметрів викликуваної функції і всіх локальних змінних, котрі в ній з'являються. Після цього аргументи копіюються і привласнюються формальним параметрам, після чого починається виконання коду функції.

Якщо усередині коду виконуваної функції виявляється виклик іншої функції, виконання припиняється і повторюється процедура, описана вище. Для цього програма виділяє додаткову пам'ять. Після виконання другої функції програма повертає керування в точку виклику усередині першої функції і відновляє її виконання.

Цей механізм не залежить від виду функцій. Зокрема, він дозволяє функції викликати саму себе — *рекурсивний виклик*.

Розглянемо декілька прикладів, що ілюструють виклик функцій. Почнемо з найпростішої лінійної ієрархії викликів.

Ланцюжок викликів

```
#include <iostream.h>

int f1(int);
int f2(int);

int main()
{
    int n=1;
    n=f1(n);
    cout << n;
    return 0;
}

int f1(int n)
{
    return 2*f2(n);
}
```

```
  int f2(int n)
{
   return 3*n;
}
```

Ця програма обчислює факторіал числа 3, тобто 1*2*3. Спочатку функція `main()` викликає функцію `f1()`, що множить свій параметр на 2, а потім, у свою чергу, — функцію `f2()`, що множить параметр на 3.

Легко бачити, що обчислення факторіала довільного цілого числа `n` за допомогою такого способу перетворюється в практично нерозв'язну задачу — для цього потрібні `n` функцій. З цієї причини обчислення факторіала виконують або в циклі, або за допомогою рекурсивних функцій. Оскільки `n!=(n- 1)!*n`, функцію для обчислення факторіала варто написати в такий спосіб.

Рекурсивне обчислення факторіала

```
#include <iostream.h>

long factorial(long);

int main()
{
   long n;
   cout << factorial(5);
   return 0;
}

long factorial (long n)
{
   if(n==0)return 1; else return n*factorial(n-1);
}
```

Як бачимо, рекурсивне рішення набагато простіше і наочніше. Однак тут є одна тонкість. Зверніть увагу на умовний оператор, що необхідний для зупинки рекурсивних викликів. Він задає *базис рекурсії*. Якби його не було, виклики продовжувалися б нескінченно.

Функції можуть бути не тільки рекурсивними, але і взаємно рекурсивними, тобто викликати один одного. Наприклад, що випливає програма обчислює найближче просте число, що перевищує задане.

Обчислення найближчого простого числа

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

void NextPrime(int);
void Increment(int);

int main()
 {
   NextPrime(20);
   return 0;
}

 void NextPrime(int n)
 {
     for(int i = 2; i < n; i++)
         if(n % i == 0) Increment(n);
     cout << n << " - наступне простої число" << endl;
     exit(0);
 }

 void Increment(int n)
 {
     NextPrime(++n);
 }
```

Функція `main()` викликає функцію `NextPrime()`, передаючи їй аргумент, що дорівнює `20`. У свою чергу, функція `NextPrime()` виконує перевірку заданого аргументу. Якщо він виявився не простим, вона викликає функцію `Increment()`, що збільшує лічильник на одиницю і повертає керування назад — функції `NextPrimer()`. Цей “пінг-понг” продовжується доти, поки чергове число не виявиться простим. У цьому випадку функція `NextPrime()` викликає функцію `exit()`, що припиняє роботу програми.

6.5..2. Виклик функції за допомогою вказівників

Функція, як і дані, зберігається в пам'яті. Отже, вона має адресу. Таким чином, на функцію можна установити вказівник. Для цього необхідно оголосити особливу конструкцію.

тип_ щоповертається_значення (* *ім'я_вказівника*)(*список_параметрів*);

Круглі дужки, у які укладений вказівник, означають, що він посилається на функцію. Якби їх не було, зміст конструкції був би зовсім іншим.

```
int (*pFunc)(int); // Вказівник на функцію, що повертає ціле число
int *pFunc(int);   // Функція pFunc(), що повертає вказівник на ціле число
```

Для ініціалізації вказівнику досить привласнити ім'я функції.

```
pFunc = Func;
```

Звідси випливає, що ім'я функції є вказівником на першу комірку області пам'яті, у якій вона записана. Оператор узяття адреси & для цього застосовувати не слід, хоча помилкою це не є. Розглянемо приклад.

Застосування вказівника на функцію

```
#include <iostream.h>

int Func(int);

int main()
{
   int n = 1;
   int (*pFunc)(int); // Оголошення вказівника на функцію
   pFunc = Func;
   n=(*pFunc)(n);
   cout << n;
   return 0;
}

int Func(int n)
{
   return 2*n;
}
```

Вказівники на функції надають програмістам можливість передавати їх як параметри інших функцій. Уявіть собі функцію, що може викликати різні функції в залежності від деяких умов. Замість операторів розгалуження в цій функції можна реалізувати єдиний алгоритм, що залежить від одного вказівника на функцію. Проілюструємо сказане наступним прикладом.

Диспетчер функцій

```
#include <iostream.h>

typedef int (*pFunc)(int,int); // Оголошення вказівника на функцію
int max(int, int);
int min(int, int);
int maxmin(pFunc, int, int);

int main()
{
   pFunc p;
   int ind,n,m;
   cin >> ind;
   cin >> n >> m;
   if(ind == 1) p = max; else p = min;
   ind=maxmin(p,n,m);
   cout << ind;
   return 0;
}

int maxmin(pFunc p, int n, int m)
{
   n=(*p)(n,m);
   return n;
}

int max(int n, int m)
```

```
{
   return n>=m?n:m;
}

int min(int n, int m)
{
   return n>=m?m:n;
}
```

Як бачимо, функція `main()` викликає функцію `maxmin()`, що, у залежності від вказівника `p`, обчислює або мінімальне, або максимальне з двох цілих чисел.

Запам'ятати, як з'являється вказівник на функцію, досить легко. Уявіть собі, що ви визначаєте звичайну функцію, але її ім'я заміняєте розіменованим вказівником, взятим у дужки. Крім того, оператор `typedef`, що визначає синонім для вказівника на функцію, виглядає незвичайно: перший операнд є фіктивним. Синонімом базового типу вказівника на функцію автоматично стає ім'я самого вказівника, у даному випадку — `pFunc`.

6.6. Еліпсис

Іноді кількість параметрів функції заздалегідь не регламентується. У цьому випадку для оголошення функції використовується еліпсис, тобто еліпсис (“`...`”), що заміняє інші параметри.

Існують дві форми еліпсиса.

`тип_значення_що_повертається` `f(`*`параметр_1, параметр_2`*`, ...);`

`тип_ значення_що_повертається` `f(...);`

Перший варіант еліпсиса дозволяє ігнорувати останню кому і ставити еліпсис відразу після останнього заздалегідь відомого параметра.

`тип_`*`значення_що_повертається`* `f(параметр_1, параметр_2 ...);`

Класичний приклад еліпсиса являє собою функція `printf()`. Її прототип виглядає в такий спосіб.

```
int printf(const char * ...);
```

Нікому не відомо заздалегідь, скільки параметрів буде потрібно вивести на екран: 1, 2 чи 122. Отже, мова повинна допускати змінний список параметрів функції `printf()`. Єдиний факт, що відомий заздалегідь, полягає в тім, що функція `printf()` повинна мати хоча б один параметр, що має тип `const char *`, що являє собою *форматний рядок*; кількість інших параметрів нічим не регламентується. Таким чином, еліпсис забезпечує гнучкий висновок даних.

```
printf("Вивід");
printf("Вивід цілого числа: %d", i);
printf("Вивід цілого  і дійсного числа: %d %f", i, f);
...
```

Розглянемо наступний приклад.

Використання еліпсиса

```
#include <stdio.h>
#include <stdarg.h>
#include <iostream.h>

double multiply(char *str, ...)
{
    double product = 1, term;
    va_list arg_list;
    va_start(arg_list, str);
    while ((term = va_arg(arg_list,double)) > 0) {
       product *= term;
    }
    printf(str, product);
    va_end(arg_list);
    return product;
}

int main()
{
    double x = multiply("Ланцюгове множення = %f\n", 2.0,3.0,5.0,-1.0);
    return 0;
}
```

Варто розрізняти ситуації, коли функція зовсім не має параметрів і має змінну кількість параметрів. Наприклад, прототип

```
void f(...);
```

означає, що функцію `f()` можна викликати з будь-якою кількістю параметрів.

Функція з довільною кількістю параметрів

```
#include <iostream.h>

void f(...);

int main()
{
    int i=0;
    f(i);
    return 0;
}

void f(...)
 {
    cout << "f ..." << endl;
 }
```

На екран виводиться наступна рядок.

```
f ...
```

Навпроти, прототип

```
void f();
```

повідомляє функцію, що не має жодного параметра. Викликати таку функцію з аргументом не можна.

6.7. Параметри, задані за замовчуванням

Як бачимо, еліпсис дозволяє задати більшу кількість параметрів, ніж передбачено прототипом функції. У мові C++ є й інша можливість: при виклику функції можна задавати меншу кількість аргументів, ніж зазначено в прототипі. Цей механізм заснований на застосуванні *параметрів за замовчуванням*. Для цього в списку параметрів необхідно вказати, яке значення повинне приймати аргумент, якщо він не зазначений явно.

```
тип_ щоповертається_значення f(параметр_1, параметр_2=значення);
```

Параметри за замовчуванням можна задати як для окремого параметра, так і для всіх параметрів. Розглянемо наступну функцію.

Функція з параметрами, заданими за замовчуванням

```
#include <iostream.h>

void f(int i=10, int k=20, int l=30);

int main()
{
   int i=11, k=21, l=31;
   f(i);
   f(i,k);
   f(i,k,l);
   return 0;
}

void f(int i, int k, int l)
{
   cout << i << " " << k << " " << l << endl;
}
```

Необхідно твердо запам'ятати наступне: 1) параметри за замовчуванням задаються в *прототипі*, а не в заголовку функції; і 2) параметри за замовчуванням повинні завершувати список параметрів — розриви між цими параметрами не допускаються і починати список вони не можуть.

6.8. Перевантаження функцій

Припустимо, нам потрібно знайти максимальне з двох заданих чисел. Виникає питання, про який тип чисел мова йде: `int`, `short`, `long`, `unsigned int`, `float` чи `double`? Хоча в кожнім з цих випадків порівняння виконується зовсім однаково, для обчислення максимального значення нам довелося б написати шість різних функцій і викликати їх у залежності від типу аргументів. У мові C++ є можливість уникнути цієї незручності — механізм *перевантажених функцій*. У його основі лежить здатність компілятора розрізняти однойменні функції, що мають різні чи типи різна кількість аргументів. Це явище іноді називають *найпростішою формою поліморфізму*. Розглянемо описаний вище приклад.

Перевантаження функцій, засноване на типах параметрів

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

```
int fmax(int, int);
double fmax(double, double);

int main()
{
   int i,j;
   double di, dj;

   cin >> i;
   cin >> j;
   cout << "Int:        "<< fmax(i,j)<< endl;

   cin >> di;
   cin >> dj;
   cout << "Double      : "<< fmax(di,dj)<< endl;

   return 0;
}

int fmax(int i, int j)
{
   return i>=j ? i : j;
}

double fmax(double i, double j)
{
   return i>=j ? i : j;
}
```

Як бачимо, робота модуля виклику стала наочніше. Спочатку він викликає функцію, що має цілочисельні параметри, а потім функцію з дійсними параметрами.

Ще одна ознака, що дозволяє розрізняти функції, — кількість параметрів. З цією властивістю зв'язана одна цікава особливість. Якщо деякі параметри функції задані за замовчуванням, при перевантаженні може виникнути неоднозначність. Розглянемо приклад.

Приклад неоднозначності

```
#include <iostream.h>
void print(int);
void print(int, int = 0);

int main()
{
   int i, j;
   cin >> i;
   cin >> j;
   print(i);   // Неоднозначність: print(i) чи print(i,0)?
   print(i,j);
   return 0;
}


void print(int a)
{
   cout << a << endl;
}

void print(int a, int b)
{
   cout << a << " " << b << endl;
}
```

Тут при виклику функції `print(i)` виникає неоднозначність, оскільки компілятор не знає, яку функцію викликати: з одним чи параметром із двома, другий з який дорівнює 0.

Крім того, варто пам'ятати, що перевантажені функції розрізняються тільки типами чи кількістю параметрів. Тип значення, що повертається, не враховується. Отже, дві функції, оголошені нижче, розрізнятися не будуть, і при компіляції виникне помилка.

```
void print(int);
int print(int);
```

Відзначимо також той дуже важливий факт, що деякі конструкції, різні зовні, по суті, ідентичні і тому не розрізняються компілятором. Наприклад, якщо параметром функції є масив, насправді у функцію передається вказівник на його перший елемент. Отже, оголошення

```
int sort(int *Array);
int sort(int Array[]);
```

означають те саме, хоча і розрізняються по зовнішньому вигляді.

Аналогічна ситуація спостерігається при використанні оператора `typedef`.

Повторне визначення перевантаженої функції

```
#include <iostream.h>
typedef int DInt;
void print(DInt);
void print(int);

int main()
{
   int i;
   DInt j;
   cin >> i;
   cin >> j;
   print(i);
   print(j);
   return 0;
}

void print(int a)
{
   cout << a << endl;
}

void print(DInt a) // Повторне визначення функції print()
{
   cout << a << endl;
}
```

Тут ми спробували обдурити компілятор, перейменувавши тип `int`. Виявляється, тепер неоднозначність виникає на рівні тіла функції. При компіляції тіла функції `print(Dint)` компілятор повідомляє, що тіло цієї функції уже існує. Отже, хоча компілятор пропускає неоднозначність на етапі аналізу прототипів, ідентичність функцій `print()` виявляється при їхній реалізації.

6.9. Функції, що підставляються

Повернемося до нашої функції.

```
int fmax(int i, int j)
{
   return i>=j ? i : j;
}
```

Вона настільки проста, що витрати машинного часу на копіювання її аргументів, обчислення адреси і точки повернення, передачу керування у функцію і назад займають більше часу, ніж її безпосереднє виконання. Як уникнути цих непотрібних витрат? Для цього в мові C++ є два способи: *макроси* і *функції, що* підставляються.

6.9..1. Макроси

Макрос — це підстановка, що заміняє собою зазначену лексему в будь-якім місці, де вона зустрінеться. Макрос визначається і виконується препроцесором. Щоб задати макрос, використовується директива `#define`. Скористаємося нею для визначення макросу, що реалізує функцію `fmax()`.

```
#define fmax(a,b) (a)>=(b) ? (a) : (b)
```

Будь-який макрос має два параметри, що є рядками. Препроцесор перед компіляцією аналізує текст програми, виявляє перший параметр макросу і заміняє його другим. Підстановка відбувається в самому тексті і не вимагає ніяких витрат часу, зв'язаних з передачею керування. Просто кожне згадування `fmax(a,b)` заміняється рядком `(a)<=(b) ? (a) : (b)`.

На перший погляд — зовсім непогано. Однак у цього підходу є ряд недоліків. По-перше, у ході підстановки препроцесор цілком ігнорує синтаксичні правила мови. Йому абсолютно неважливо, чи правильний вираз виникне в результаті заміни першого рядка другий, чи відповідають типи параметрів один одному і так далі; відповідальність за виявлення цих помилок, якщо вони виникнуть, перекладається на плечі компілятора. По-друге, макроси можуть породити зовсім несподіваний текст, тому що вони ігнорують пріоритети операторів.

Припустимо, ми пропустили дужки навколо ідентифікаторів a і b (до речі, ви не задавалися питанням, навіщо така вигадлива форма запису?).

```
((a)<=(b) ? (a) : (b)).
```

Приклад невірного макросу

```
#include <iostream.h>
#define fmax(a,b) a>=b?a:b

int main()
{
   int i = 5, j = 10, k;
   k=2*fmax(i,j);
   cout << k;
   return 0;
}
```

Тепер замість очікуваного числа 20, що повинне було б вийти (тобто 2*10), на екран виводиться число 5, оскільки насправді виконується `2*i>=j?i:j`. При недостатньо акуратному звертанні з макросами легко зробити помилку, зв'язану з неправильним обліком пріоритетів операторів. Оскільки множення має більш високий пріоритет, ніж тернарна альтернатива, спочатку виконується подвоєння першого параметра, а потім він порівнюється з другим. Результат у наявності!

Як би там ні було, макроси залишаються могутнім інструментом, що дозволяє програмісту ефективно експлуатувати компілятор. Однак для цього потрібно неабияка майстерність і точність. Спростити цю задачу дозволяють *функції, що* підставляються.

6.9..2. Функції, що підставляються

Функція, що *підставляється*, як і макрос, підставляється в текст програми, а не викликається при її виконанні. Для цього досить поставити перед її прототипом (але не заголовком!) ключове слово `inline`. Утім, коли оголошення і визначення функції збігаються, це обмеження знімається.

```
inline int fmin(int a, int b)
{
   return (a >= b ) ? a : b;
}
```

Ключове слово `inline` компілятор розглядає як рекомендацію, а не директиву. Наприклад, якщо компілятор вважає, що функція занадто складна (розмір її тіла перевищує три рядки, містить оператор циклу, чи порівняння інші небажані дії), вона зробить функцію звичайною. Крім того, варто врахувати, що функція, що підставляється, в очах компілятора виглядає як один нерозривний рядок. Знайти помилку шляхом покрокового налагодження в такій функції неможливо. Щоб уникнути цих проблем, варто спочатку оголосити звичайні функції, виправити можливі помилки і лише в остаточному варіанті зробити необхідні функції, що підставляються.

6.10. Резюме

- Кожна функція повинна мати оголошення і визначення. Оголошення функції називається її прототипом. Загальний вид прототипу виглядає в такий спосіб.

```
тип_значення_ що_повертається ім'я_функції(тип_параметра1,
                                            ... ,
                                            тип_параметра N);
```

- У мові C++ функція може повернути тільки одне значення.
- Список типів, зазначений у прототипі функції, являє собою перерахування типів її формальних параметрів, розділених комами. Якщо функція нічого не одержує ззовні, цей список порожній. Імена параметрів у прототипі вказувати необов'язково. Крім того, типи параметрів можна модифікувати ключовим словом `const`. Іноді ім'я змінної допомагає швидше знайти помилку, що виникла при компіляції програми. Кожний формальний параметр функції повинний бути оголошений окремо.
- Кількість, порядок і типи формальних параметрів утворять *профіль* параметрів функції.
- У свою чергу профіль і тип значення, що повертається, є *протоколом* функції. Не забувайте: прототип завжди повинний завершуватися точкою з комою!
- Ім'я і профіль функції утворюють її *сігнатуру*.
- Для реалізації функції необхідно виконати її визначення, що має наступний вид.

```
тип_значення_що_повертається ім'я_функції(
     тип_параметра1 ім'я_параметра1,
        ... ,
```

```
      тип_параметра ім'я_параметра)
{
   тіло функції
}
```

- Тип значення, що повертається, ім'я функції і список формальних параметрів *утворюють заголовок* функції. Прототип і заголовок функції повинні збігатися.
- Фігурні дужки обмежують *тіло функції*, що складає з операторів, що розв'язують поставлену задачу. Крім того, фігурні дужки є межами області видимості всіх змінних, оголошених усередині тіла функції (такі змінні називаються *локальними*). Ця область закрита від зовнішнього світу, і доступ до неї з інших функцій неможливий. Вся інформація може надходити ззовні лише по двох каналах: через фактичні параметри, чи *аргументи*, що представляють собою значення формальних параметрів, і через глобальні змінні, які є видимими в будь-якій точці програми.
- Локальні змінні існують лише під час виконання функції. Вони створюються при її виклику і знищуються при поверненні керування в модуль, звідки зроблено виклик. Однак існує можливість зберегти значення локальних змінних між викликами функцій — оголосити такі змінні статичними (див. лекцію 3, розділ 3.3.1).
- Виклик функції виконується оператором `()`. Для цього необхідно перед дужками вказати ім'я функції, а усередині дужок — список аргументів.
- Змінні, оголошені в заголовку функції, називаються *параметрами* функції. Вони призначені для того, щоб приймати значення аргументів. Як і локальні змінні, вони створюються при виклику функції і знищуються при виході з неї.
- Функція `main()` служить диспетчером і організує взаємодію між усіма функціями. Будучи головною, ця функція має усі властивості звичайних функцій. Зокрема, вона має заголовок, список формальних параметрів, тіло, значення, що повертається, — практично всі атрибути функції, за винятком прототипу.
- Список формальних параметрів функції `main()` фіксований. Він може складатися лише з зазначених параметрів, кількість яких обмежується можливостями операційної системи (як правило, це число дорівнює 32767). Аргументи функції `main()` повинні вказуватися в командному рядку слідом за ім'ям модуля, що виконується.
- Для витягу аргументів командного рядка використовуються убудовані параметри `argv`, `argc` і `env`. (У деяких компіляторах останній параметр називається `envp`.) Параметр `argc` є цілочисельним. Він задає кількість аргументів, зазначених у командному рядку. Ім'я програми вважається першим аргументом. Отже, якщо користувач не задав жодного власного аргументу, значення змінної `argc` буде дорівнювати 1. Параметр `argv` є вказівником на масив символьних вказівників, що посилаються на аргументи командного рядка.
- У мові C++ є два способи передачі аргументів у функцію: *за значенням* і *за посиланням*. За замовчуванням, якщо аргументом є не масив, застосовується перший спосіб. Для цього створюється копія значення аргументу, що привласнюється формальному параметру. Всі операції, виконані усередині функції, стосуються лише копії аргументу і не впливають на оригінал, що існує в модулі, що здійснює виклик.
- *Передача аргументів за посиланням*, означає, що у функцію передається не копія аргументу, а його адреса. У цьому випадку функція одержує безпосередній доступ до аргументу, а формальний параметр збігається з фактичним.
- Функції можна передати вказівник на аргумент. Передача вказівника відбувається традиційно, тобто у функції створюється копія вказівника на аргумент.
- Ще один механізм заснований на застосуванні посилань. Для цього перед ім'ям аргументу в прототипі функції і її заголовку (вони завжди повинні збігатися!) слід поставити оператор узяття адреси. Такий прототип буде означати, що аргумент передається за посиланням, хоча визначення функції зовсім не відрізняється від передачі за значенням (ніяких розіменувань не потрібно).
- Передача масиву у функцію є виключенням із загальноприйнятих правил. За замовчуванням масив передається за посиланням. Функція, параметром якої є масив, може модифікувати будь-які його елементи.
- У мові C++ немає способу, що дозволив би уникнути явного завдання розміру масиву, переданого функції. Тому передавати масив без завдання кількості рядків і стовпців неможливо.
- Функції розподіляються на дві категорії. Вони можуть або повертати деяке значення, або не повертати нічого. Для повернення обчисленого значення існує два механізми. По-перше, функція може повернути значення, виконавши оператор `return`, що негайно передасть керування в модуль виклику одночасно з тимчасовою змінною, що містить результат обчислень. По-друге, функція може привласнити обчислене значення параметру, переданому за посиланням. У цьому випадку в модулі виклику можна скористатися результатами побічного ефекту.

- Якщо функція не повертає ніяких значень, перед її ім'ям у прототипі і заголовку необхідно поставити ключове слово `void`. Необхідно підкреслити наступне: незважаючи на те що ключове слово `void` стоїть на місці типу значення, що повертається, це зовсім не означає, що повертається деяке значення "порожнього типу". Навпаки, така функція нічого не повертає, і її виконання припиняється, як тільки потік керування досягає фігурної дужки, що закриває її тіло. Утім, з такої функції можна просто "катапультуватися" за допомогою оператора `return`, не вказавши ніякого значення, що повертається. По суті, така функція є процедурою.
- Виклик функції виконується в такий спосіб: спочатку зберігається інформація, що дозволяє відновити виконання програми після повернення з функції, потім виділяється пам'ять для формальних параметрів викликуваної функції і всіх локальних змінних, котрі в ній з'являються. Після цього аргументи копіюються і привласнюються формальним параметрам, після чого починається виконання коду функції.
- Якщо усередині коду виконуваної функції виявляється виклик іншої функції, виконання припиняється і повторюється процедура, описана вище. Для цього програма виділяє додаткову пам'ять. Після виконання другої функції програма повертає керування в точку виклику усередині першої функції і відновляє її виконання. Цей механізм не залежить від виду функцій. Зокрема, він дозволяє функції викликати саму себе — *рекурсивний виклик*.
- Для зупинки рекурсивних викликів необхідний *базис рекурсії*. Якби його не було, виклики продовжувалися б нескінченно.
- Функції можуть бути не тільки рекурсивними, але і взаємно рекурсивними, тобто викликати один одного.
- Функція, як і дані, зберігається в пам'яті. Отже, вона має адресу. Таким чином, на функцію можна установити вказівник. Для цього необхідно оголосити особливу конструкцію.
  ```
  тип_значення_ що_повертається (* ім'я_вказівника)(список_параметрів);
  ```
- Круглі дужки, у які укладений вказівник, означають, що він посилається на функцію. Якби їх не було, зміст конструкції був би зовсім іншим.
  ```
  int (*pFunc)(int); // Вказівник на функцію, що повертає ціле число
  int *pFunc(int); // Функція pFunc(), що повертає вказівник на ціле число
  ```
- Для ініціалізації вказівнику досить привласнити ім'я функції.
  ```
  pFunc = Func;
  ```
  Звідси випливає, що ім'я функції є вказівником на першу комірку області пам'яті, у якій вона записана. Оператор узяття адреси & для цього застосовувати не слід, хоча помилкою це не є.
- Вказівники на функції надають програмістам можливість передавати їх як параметри інших функцій. Уявіть собі функцію, що може викликати різні функції в залежності від деяких умов. Замість операторів розгалуження в цій функції можна реалізувати єдиний алгоритм, що залежить від одного вказівника на функцію. Проілюструємо сказане наступним прикладом.
- Запам'ятати, як з'являється вказівник на функцію, досить легко. Уявіть собі, що ви визначаєте звичайну функцію, але її ім'я заміняєте розіменованим вказівником, взятим у дужки. Крім того, оператор `typedef`, що визначає синонім для вказівника на функцію, виглядає незвичайно: перший операнд є фіктивним. Синонімом базового типу вказівника на функцію автоматично стає ім'я самого вказівника, у даному випадку — `pFunc`.
- Іноді кількість параметрів функції заздалегідь не регламентується. У цьому випадку для оголошення функції використовується еліпсис, тобто еліпсис ("`...`"), що заміняє інші параметри.
- Існують дві форми еліпсиса.
  ```
  тип_значення_що_повертається_значення f(параметр_1, параметр_2, ...);
  тип_значення_що_повертається f(...);
  ```
- Перший варіант еліпсиса дозволяє ігнорувати останню кому і ставити еліпсис відразу після останнього заздалегідь відомого параметра.
  ```
  тип_ щоповертається_значення f(параметр_1, параметр_2 ...);
  ```
  Класичний приклад еліпсиса являє собою функція `printf()`. Її прототип виглядає в такий спосіб.
  ```
  int printf(const char * ...);
  ```
- Еліпсис дозволяє задати більшу кількість параметрів, ніж передбачено прототипом функції. У мові C++ є й інша можливість: при виклику функції можна задавати меншу кількість аргументів, ніж зазначено в прототипі. Цей механізм заснований на застосуванні *параметрів за замовчуванням*. Для цього в списку параметрів необхідно вказати, яке значення повинне приймати аргумент, якщо він не зазначений явно.
  ```
  тип_ значення_що_повертається f(параметр_1, параметр_2=значення);
  ```
  Параметри за замовчуванням можна задати як для окремого параметра, так і для всіх параметрів.
- Необхідно твердо запам'ятати наступне: 1) параметри за замовчуванням задаються в *прототипі*, а не в заголовку функції; і 2) параметри за замовчуванням повинні завершувати список параметрів — розриви між цими параметрами не допускаються і починати список вони не можуть.

- У мові C++ є можливість використати механізм *перевантажених функцій*. У його основі лежить здатність компілятора розрізняти однойменні функції, що мають різні типи чи різну кількість аргументів. Це явище іноді називають *найпростішою (статичною) формою поліморфізму*.
- Перевантажені функції розрізняються тільки типами чи кількістю параметрів. Тип значення, що повертається, не враховується.
- Витрати машинного часу на копіювання її аргументів, обчислення адреси і точки повернення, передачу керування у функцію і назад займають більше часу, ніж її безпосереднє виконання. Як уникнути цих непотрібних витрат? Для цього в мові C++ є два способи: *макроси* і *функції, що* підставляються.
- *Макрос* — це підстановка, що заміняє собою зазначену лексему в будь-якім місці, де вона зустрінеться. Макрос визначається і виконується препроцесором. Щоб задати макрос, використовується директива `#define`. Наприклад,
  ```
  #define fmax(a,b) (a)>=(b) ? (a) : (b)
  ```
- Будь-який макрос має два параметри, що є рядками. Препроцесор перед компіляцією аналізує текст програми, виявляє перший параметр макросу і заміняє його другим. Підстановка відбувається в самому тексті і не вимагає ніяких витрат часу, зв'язаних з передачею керування. Просто кожне згадування `fmax(a,b)` заміняється рядком `(a)<=(b) ? (a) : (b)`.
- У цього підходу є ряд недоліків. По-перше, у ході підстановки препроцесор цілком ігнорує синтаксичні правила мови. Йому абсолютно неважливо, чи правильний вираз виникне в результаті заміни першого рядка другий, чи відповідають типи параметрів один одному і так далі; відповідальність за виявлення цих помилок, якщо вони виникнуть, перекладається на плечі компілятора. По-друге, макроси можуть породити зовсім несподіваний текст, тому вони ігнорують пріоритети операторів.
- Функція, що *підставляється*, як і макрос, підставляється в текст програми, а не викликається при її виконанні. Для цього досить поставити перед її прототипом (але не заголовком!) ключове слово `inline`. Утім, коли оголошення і визначення функції збігаються, це обмеження знімається.
- Ключове слово `inline` компілятор розглядає як рекомендацію, а не директиву. Наприклад, якщо компілятор вважає, що функція занадто складна, вона зробить функцію звичайною. Крім того, варто врахувати, що функція, що підставляється, в очах компілятора виглядає як один нерозривний рядок. Знайти помилку шляхом покрокового налагодження в такій функції неможливо.

6.11. Контрольні питання

1. Як виглядають оголошення і визначення функції?
2. Скільки значень може повертати функція у мові C++?
3. Як виглядає список типів, зазначений у прототипі функції?
4. Що таке *профіль* параметрів функції?
5. Що таке *протокол* функції?
6. Що таке *сигнатура* функції?
7. Що таке заголовок функції?
8. Що таке *тіло функції*?
9. Які змінні називаються *параметрами* функції?
10. Назвіть вхідні аргументи функції `main`.
11. Назвіть два способи передачі аргументів у функцію у мові C++ .
12. Опишіть механізм *передачі аргументів за значенням*.
13. Опишіть механізм *передачі аргументів за посиланням*
14. Назвіть особливості передачі масивів як аргументів функції.
15. Назвіть дві категорії функцій відносно значень, що повертаються.
16. Опишіть механізм виклику функції.
17. Опишіть механізм рекурсивного виклику функції.
18. Опишіть конструкцію вказівника на функцію;
19. Опишіть дві форми еліпсиса.
20. Опишіть перший варіант еліпсиса і його призначення.
21. Опишіть другий варіант еліпсиса і його призначення.
22. Як задаються параметри функції за замовчуванням.
23. Опишіть механізм *перевантажених функцій*.
24. Що таке *макрос*?
25. Опишіть механізм використання функції, що *підставляється*.