Лекція 12. Особливі випадки оголошення функцій. Простори імен. Класи пам'яті

1. *Перевантажені функції. Статичний поліморфізм.*
2. *Аргументи за замовчуванням функцій.*
3. *Вбудовані функції.*
4. *Функції зі змінною (невідомою) кількістю аргументів.*
5. *Статичні локальні змінні функцій.*
6. *Простори імен: призначення, оголошення, використання.*
7. *Класи пам'яті, діапазони доступу, зв'язність імен.*

Перевантаження функцій

Зазвичай різним функціям дають різні імена. Якщо ж декілька функцій виконують одну і ту ж дію над об'єктами різних типів, то зручніше дати однакові імена всім цим функціям. Перевантаженням імені називається його використання для позначення різних функцій (операторів), що працюють з різними типами аргументів. Так створюють сім'ю функцій, що виконують одне перетворення для різних типів даних. Наприклад, функції відшукання найбільшого значення для різних наборів параметрів можуть називатися однаково:

```
int maxVal(int a, int b, int c)
{
	return a > b ? (a > c ? a : c) : (b > c ? b : c);
}

double maxVal(double * A, unsigned n)
{
	double max = A[0];
	for (unsigned i = 1; i < n; ++i)
		if (A[i] > max) max = A[i];
	return max;
}
```

Перевантаження широко застосовується у C++. Дійсно, для операцій додавання є тільки одне ім'я +, але воно використовується для додавання і цілих чисел, і чисел з плаваючою крапкою, і вказівників. Такий підхід легко можна поширити на операції, визначені користувачем, тобто на функції. Наприклад, у лекції 8 ми перевантажили оператори +, >, << для типу *Time*.

Список параметрів функції називають *сигнатурою функції*. Сигнатуру визначає кількість параметрів, їхні типи і порядок. Тип результату на сигнатуру не впливає. Перевантажувати можна імена функцій з різними сигнатурами. Під час трансляції функції компілятор створює для власних потреб кодове ім'я функції. У ньому він поєднує власне ім'я та сигнатуру, тому компілятор легко розрізняє однойменні функції з різними сигнатурами.

Перевантаження імен функцій ще називають *статичним поліморфізмом функцій*. Поліморфізм – різноманітність форм – термін, що прийшов з біології, означає, що за одним іменем можуть ховатися різні сутності. Статичний, бо за списком аргументів у точці виклику перевантаженої функції компілятор може ще на етапі трансляції визначити, якого саме представника сім'ї функцій викликають. Згодом ми познайомимося з динамічним поліморфізмом, за якого сутність виклику з'ясовується лише на етапі виконання.

Пригадаємо ще приклади перевантажених імен, які вже зустрічалися в цьому курсі. Наприкінці лекції 9 йшла мова про об'єднання впорядкованих послідовностей.

```
// об'єднання двох впорядкованих масивів у новий
int * merge(int*A, size_t n, int*B, size_t m);

// об'єднує два впорядковані списки (копіює елементи в новий)
List_t merge(List_t a, List_t b)
```

Обидві функції *merge* виконують однакову роботу (про що свідчить їхня назва), проте з різними об'єктами (про що свідчать їхні сигнатури).

У лекції 8 було оголошено тип *Time* з трьома конструкторами:

```
// Тип моделює тривалість у годинах і хвилинах
struct Time
{
	unsigned hours;
	unsigned minutes;
	Time() :hours(0), minutes(0){} // конструктор за замовчуванням
	Time(unsigned h, unsigned m);  // конструктор з параметрами
	Time(unsigned t);           // конструктор з одним параметром перетворює ціле на Time
};
```

Декілька конструкторів одного типу діють, як перевантажені функції. Про перевантаження операторів для типу *Time* ми вже згадували.

Для транслятора в перевантажених функціях спільне тільки одне – ім'я. Очевидно, за змістом такі функції подібні, але для транслятора це не має жодного значення. Мова не сприяє і не перешкоджає створенню перевантажених функцій. Їхнє визначення служить, перш за все, для зручності запису. Адже для функцій з такими традиційними іменами, як *sqrt*, *print* або *open*, не можна цією зручністю нехтувати.

Якщо саме ім'я грає важливу семантичну роль, наприклад, в таких операціях, як +, * і <<, або для конструктора класу, то така зручність стає суттєвим фактором. При виконанні функції з іменем *f* транслятор повинен розібратися, яку саме функцію *f* слід викликати. Для цього порівнюються типи фактичних параметрів, зазначені у виклику, з типами формальних параметрів всіх описів функцій з іменами *f*. У результаті викликається та функція, у якої формальні параметри найкращим чином зіставлені з параметрами виклику, або видається помилка, якщо такої функції не знайшлося.

Правила співставлення параметрів (за спаданням пріоритету)

1. Точний збіг: співставлення відбулося без всяких перетворень типу або тільки з неминучими перетвореннями (наприклад, імені масиву на вказівник, імені функції на вказівник на функцію, або типу *T* на *const T*).
2. Співставлення з використанням стандартного підняття типу від меншого до більшого (тобто *char* в *int*, *short* в *int* та їх беззнакових двійників в *int*, *int* в *long long*, а також перетворень *float* в *double*).
3. Співставлення з використанням стандартних перетворень (наприклад, *int* в *double*, *derived** в *base**, *derived*& в *base*&, *unsigned* в *signed* і навпаки).
4. Співставлення з використанням перетворень, визначених програмістом.
5. Співставлення з використанням трикрапки ... в оголошенні функції.

Якщо знайдено два співставлення за одним найбільш пріоритетним правилом, то виклик вважається неоднозначним, а, значить, помилковим. Зазначимо також, що порядок оголошення перевантажених функцій не впливає на вибір їх компілятором.

Значення за замовчуванням аргументів функції

Функції з значеннями аргументів за замовчуванням, використовують тоді, коли ми наперед знаємо, що найчастіше дану функцію будемо викликати саме з таким значенням даного параметра. Наприклад, для розмежування тексту, виведеного на консоль, можна використати функцію, що друкує горизонтальні лінії.

```
void writeLine(char c = '-', unsigned k = 80)
{
	for (unsigned i = 0; i < k; ++i) std::cout << c;
	std::cout << '\n';
}
```

Тут перший параметр задає вигляд символа-розділювача, що імітує лінію, другий – довжину лінії, значення 80 використано тому, що рядок консолі містить 80 позицій. Оскільки параметри мають значення за замовчуванням, то при виклику функції аргументи можна не вказувати. Значення аргумента записують, якщо хочуть скасувати значення за замовчуванням. Виклики *writeLine*(); *writeLine*('='); *writeLine*('_', 40); виведуть на консоль три різні рядки: вісімдесят мінусів, вісімдесят знаків дорівнює та сорок підкреслень відповідно.

Оголошена функція *writeLine* у використанні подібна до сім'ї трьох перевантажених функцій: з двома параметрами, з одним і без параметрів.

Зазвичай функцію оголошують у файлі заголовків, а визначають у файлі коду. Якщо потрібно задати значення за замовчуванням, то їх оголошують у прототипі функції у файлі заголовків:

```
// оголошення прототипу у header.h
void writeLine(char c = '-', unsigned k = 80);

// визначення функції у source.cpp
void writeLine(char c, unsigned k)
{
     for (unsigned i = 0; i < k; ++i) std::cout << c;
     std::cout << '\n';
}
```

У мові C++ співставлення аргументів з параметрами відбувається за місцем, тому параметри зі значеннями за замовчуванням розташовують наприкінці списку параметрів. Неможливо, наприклад, пропустити перший параметр і задати значення другого.

```
writeLine();          // правильно, означає writeLine('-', 80);
writeLine('=');       // правильно, означає writeLine('=', 80);
writeLine('_', 40);   // правильно, вказані всі аогументи
writeLine( , 40);     // неправильно, не можна пропустити аргумент
writeLine(40);        // синтаксично правильно, означає writeLine(char(40), 80);
                      // але, чи цього хотів користувач?
```

Ще один приклад: функція, яка повертає *n* перших символів з масиву символів. За замовчуванням дана функція повертає один символ.

Прототип функції:

```
char * left(const char * str, int n = 1);
```

Оголошення:

```
char * left(const char * str, int n)
{
     if (n < 0) n = 0;
     char * p = new char[n + 1];
     int i;
     for (i = 0; i < n && str[i]; ++i) p[i] = str[i];
     while (i <= n) p[i++] = '\0';
     return p;
}
```

Застосування:

```
cout << left("Hello World!", 5);                    // Hello
cout << left("Hello World!");                       // H
```

У лекції 9 ми використовували значення за замовчуванням у конструкторі типу *Node*:

```
struct Node; // попереднє оголошення типу
typedef Node* List_t;

struct Node  // власне оголошення
```

```
{
    int value;      // елемент даних
    List_t link;    // поле зв'язку
    Node(int x, List_t p = nullptr) :value(x), link(p) {}
};
```

Вбудовані функції

Вбудовані функції є засобом економії часу виконання. Замість стандартного виклику функції, при якому запам'ятовується адреса повернення з функції, створюються локальні змінні, копіюються аргументи, передається керування у функцію, компілятор вбудовує код функції в місце її виклику. Таке використання функцій оптимізує час виконання програми, але в свою чергу збільшує розмір виконуваного файла. Найбільш актуально, коли функція коротка і часто викликається.

Вбудованою може бути функція, яка:

- приймає аргументи-значення;
- визначення функції вміщається в один рядок;
- не рекурсивна.

Вбудовану функцію визначають у заголовковому файлі. Наприклад

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

Компілятор сам вирішує, вбудовувати, чи ні дану функцію. Специфікатор *inline* є лише рекомендацією, підказкою програміста. Справжнє вбудовування відбудеться після того, як компілятор перевірить функцію за закладеними в нього критеріями. Наведений вище приклад має всі шанси на вбудовування, тому виклик *double b = sqr(sin(a));* компілятор перетворить на *double x = sin(a); double b = x*x;* чи щось подібне.

Функції з невідомою кількістю аргументів

Існують функції, в описі яких неможливо вказати число і типи всіх допустимих параметрів. Тоді список формальних параметрів завершується трикрапкою, що означає: «і, можливо, ще декілька аргументів», але хоча б один параметр мусить бути. Приклад з офіційної документації, процедура *ShowVar* розпізнає і друкує свої аргументи:

```
#include <stdarg.h>
//  ShowVar takes a format string of the form "ifcs", where each character specifies the
//  type of the argument in that position.
//      i = int
//      f = float
//      c = char
//      s = string (char *)
//  Following the format specification is a variable list of arguments. Each argument
//  corresponds to a format character in the format string to which the szTypes parameter
//  points
void ShowVar(char *szTypes, ...)
{
      va_list vl;
      int i;
      //  szTypes is the last argument specified; you must access
      //  all others using the variable-argument macros.
      va_start(vl, szTypes);
      // Step through the list.
      for (i = 0; szTypes[i] != '\0'; ++i)
      {
            union Printable_t
            {
                  int     i;
                  double  f;
                  char    c;
```

```
                char   *s;
            } Printable;

            switch (szTypes[i])   // Type to expect.
            {
            case 'i':
                Printable.i = va_arg(vl, int);
                cout << Printable.i << '\n';
                break;
            case 'f':
                Printable.f = va_arg(vl, double);
                cout << Printable.f << '\n';
                break;
            case 'c':
                Printable.c = va_arg(vl, char);
                cout << Printable.c << '\n';
                break;
            case 's':
                Printable.s = va_arg(vl, char *);
                cout << Printable.s << '\n';
                break;
            default:
                break;
            }
        }
        va_end(vl);
}
```

Дещо полегшений варіант тієї ж процедури:

```
void PrintVar(char *szTypes, ...)
{
        va_list vl;
        va_start(vl, szTypes);
        for (int i = 0; szTypes[i] != '\0'; ++i)
        {
            switch (szTypes[i])   // Type to expect.
            {
            case 'i':
                cout << (int)va_arg(vl, int) << '\n';
                break;
            case 'f':
                cout << (double)va_arg(vl, double) << '\n';
                break;
            case 'c':
                cout << (char)va_arg(vl, char) << '\n';
                break;
            case 's':
                cout << (char*)va_arg(vl, char *) << '\n';
                break;
            default:
                break;
            }
        }
        va_end(vl);
}
```

Щоб мати змогу працювати зі змінною кількістю параметрів, потрібно приєднати заголовковий файл *stdarg.h* або *cstdarg*. Звідти ми отримуємо тип *va_list* і макроси (або макровизначення) *va_start*, *va_arg*, *va_copy*, *va_end*. Ми не будемо зараз обговорювати, що таке *макрос*, достатньо знати, що його використання подібне до виклику функції. Отже, об'єкт *va_list vl* міститиме всю інформацію про список аргументів, виклик *va_start* пов'язує його з першим переданим аргументом. Виклик *va_arg* повертає черговий аргумент зі списку. За

макросом не можна визначити тип результату, тому застосовують приведення типу. Виклик *va_end* звільняє список аргументів. Виклик *va_copy* можна використати, щоб скопіювати список аргументів, якщо це потрібно.

При виклику *PrintVar* обов'язково потрібно вказати параметр типу *char** та можуть бути (а можуть і не бути) ще інші параметри, наприклад

```
PrintVar("fcsi", 32.1, 'a', "Test string", 4);
```

Параметр *szTypes* відіграє роль дескриптора, який описує кількість і тип наступних аргументів. Аналіз його вмісту керує роботою процедури.

При передаванні змінної кількості аргументів можна використати інший прийом: передавати спеціальне термінальне значення, яке позначає закінчення списку аргументів.

```
// процедура обчислює і друкує суму переданих їй аргументів
// відомо, що всі вони цілі, список закінчується нулем
void writeSum(int d ...)
{
	int s = 0;
	va_list ap;
	va_start(ap, d);
	cout << "SUM = ";
	while (d != 0){
		if (d > 0) cout << '+'; cout << d;
		s += d;
		d = va_arg(ap, int);
	} va_end(ap);
	cout << " = " << s << '\n';
}
```

Після викликів

```
writeSum(0);
writeSum(-1,2,3,0);
```

Отримано результати

```
SUM =  = 0
SUM = -1+2+3 = 4
```

Описані функції користуються для розпізнавання своїх фактичних параметрів недоступною транслятору інформацією. У перших двох перший параметр є рядком, що специфікує формат виведення. Він містить спеціальні символи, які дозволяють правильно сприйняти наступні параметри. Процедура *writeSum* покладається на добросовісний виклик: всі аргументи мають бути цілими, останнє значення – нулем.

У добре продуманій програмі може знадобитися, як виняток, лише декілька функцій, у яких вказані не всі типи параметрів. Щоб обійти контроль типів параметрів, краще використовувати перевантаження функцій або стандартні значення параметрів, ніж параметри, типи яких не були описані. Трикрапка стає необхідною тільки тоді, коли можуть змінюватися не тільки типи, але і кількість параметрів.

Статичні локальні змінні функцій

У C++ є змога оголосити *статичну локальну змінну* (чи змінні) в блоці функції. Для цього використовують специфікатор *static*. До статичної змінної, як і до інших локальних, можна доступитися лише з блока функції, але пам'ять вона отримує не в стеку, як інші, а в статичній частині, тому існує весь час виконання програми (а не лише під час виконання функції). Ініціалізувати її можна виключно константним виразом. Статичні змінні дають змогу функції «пам'ятати про її попередні життя».

У прикладі нижче статичну змінну використано для контролю кількості викликів.

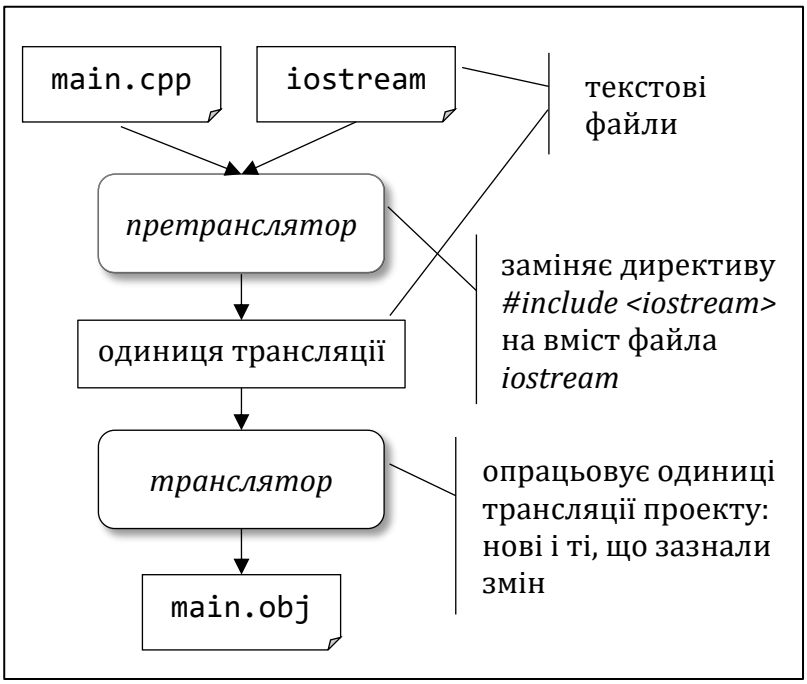
```
unsigned printChars(char c, int k)
{
      static unsigned counter = 0;
      for (int i = 0; i < k; ++i) cout << c;
      return ++counter;
}
```

Змінну *counter* буде створено один раз під час першого виклику *printChars*. Під час усіх наступних вона існуватиме зі значенням, яке збережеться від попереднього виклику.

```
printChars('A', 3);   // counter == 1;
// рядки "*****" буде надруковано не більше семи разів,
// перехід на новий рядок – не більше шести разів.
while (printChars('*', 5) < 7) std::cout << '\n';
```

One Definition Rule

Програма мовою C++ складається з декількох файлів: головної програми (файла, що містить функцію *main*), файлів заголовків стандартних і користувача, *cpp*-файлів з визначеннями функцій, методів тощо. Мова надає засоби та заохочує програміста розташовувати функції, що складають програму, в різних файлах. Для цього є декілька причин: окрема компіляція файлів пришвидшує виготовлення програми, полегшується повторне використання коду, організація колективної роботи.



Яким чином об'єктам одного файла стає відомо про об'єкти іншого? Пригадаємо, як відбувається компіляція (див. схему).

Кожне ім'я, що використовується в одиниці трансляції, треба оголосити. Не буває «чарівних» імен, імен за замовчуванням тощо. Кожне треба один раз *оголосити* перед використанням. Одне з завдань заголовкового файла – постачання таких оголошень. У цілому проекті оголошень імені (наприклад, типу чи функції) може бути декілька – по одному на одиницю трансляції. Десь в проекті (у котромусь файлі) має бути *визначення* імені.

Правило одного визначення (One Definition Rule, ODR) – важлива концепція в мові програмування C++, що визначена в ISO C++ Standard(ISO/IEC 14882) 2003, в розділі 3.2. Якщо коротко, то ODR стверджує:

В окремій одиниці трансляції тип, функція, змінна, або шаблон можуть мати не більше одного визначення. Хоча деякі можуть мати яку завгодно кількість оголошень у проекті.

В програмі об'єкт або невбудована функція не можуть мати більш ніж одне визначення. Якщо об'єкт чи функція не використовуються, тоді достатньо оголошення без визначення. У випадку використання вони повинні мати рівно одне визначення.

Деякі сутності, наприклад, типи, шаблони або вбудовані функції можуть мати більше ніж одне визначення тільки якщо:

- вони знаходяться в різних одиницях трансляції;
- вони ідентичні лексема за лексемою;
- значення лексем однакове в обох одиницях трансляції.

Усі оголошення та визначення мають бути узгоджені. Відсутність оголошення чи повторне оголошення виявляє компілятор, неузгодженість виявляє компонувальник (редактор зв'язків).

Уявімо, що до складу деякого проекту входять такі файли:

Source1.cpp

```
int x = 1;       // визначення
double f(int);   // оголошення
```

Source2.cpp

```
extern int x;    // оголошення
double f(int n)  // визначення
{ return n*.5; }
```

Правило ODR не порушено, *f* – звичайна функція, *x* – глобальна змінна, визначена у файлі, поза будь-яким блоком.

Як ми вже сказали, файл заголовків – постачальник оголошень. До його складу традиційно включають:

- визначення символьних констант;
- прототипи функцій;
- визначення вбудованих функцій;
- оголошення переліків, структур, класів, шаблонів.

Заголовковий файл не повинен містити

- визначення змінних;
- визначення функцій;
- неіменовані простори імен.

Простори імен

Імена локалізуються в тому блоці, в якому оголошені. Функції оголошують поза блоками, тому їхні імена є глобальними. Типи зазвичай також оголошують поза блоками, і їхні імена стають глобальними. У великому проекті може виникнути проблема супроводу великої кількості неструктурованих імен. Для наведення порядку з приналежністю глобального імені до певної зони відповідальності використовують *простори імен*.

Простір імен – *namespace* – новий інструмент локалізації імен. Він задає область видимості імені. Два однакові імені з різних просторів не конфліктують. Схематично оголошення простору імен можна зобразити так:

```
namespace Space
{
     <оголошення констант, змінних, типів, функцій>
}
```

Його розташовують зовні будь-яких блоків (можна, хіба що, всередині іншого простору імен). Оголошення відкрите: різні одиниці трансляції можуть додавати до простору нові імена. До речі, стандартний простір *std* так і влаштований: його імена розділено між різними файлами заголовків.

Щоб звернутися до імені *name* з простору *Space*, потрібно використати кваліфікатор імені: *Space*::*name*. Таке ім'я відрізняється від, наприклад, *Calculation*::*name* чи *Report*::*name*. Використання кваліфікованого імені – найбільш безпечний та інформативний спосіб звертання, проте постійне вживання довгих префіксів (як *Calculation*::) може суттєво затримувати створення тексту програми, тому, для спрощення звертань, використовують *using*-оголошення або директиву *using*.

Оголошення *using Space*::*name*; є розумним компромісом між безпекою та зручністю. Воно вводить в область видимості одне ім'я з простору *Space* як локальне. Надалі для звертання до *name* не потрібно вказувати кваліфікатор.

Директива *using namespace Space*; відкриває цілий простір *Space*, всі його імена стають глобальними (зовнішніми). Програміст може оголосити в блоці свою змінну *name* і тим самим перекрити доступ до *Space*::*name*. Використання директиви найменш безпечне і, здається, найбільш поширене.

Приклад використання просторів імен можна знайти в [4, гл. 9.3].

Ми також могли б скористатися послугами простору імен, наприклад, щоб об'єднати в одну область видимості імена типів і функцій для роботи зі списками, які ми розробляли в лекції 9. Заголовковий файл *ListTools.h* зміниться не дуже: тільки додалося оголошення *namespace*, яке огортає всі оголошення типів і функцій.

```
#pragma once
namespace ListTools
{
	struct Node;
	typedef Node* List_t;
	struct Node
	{
		int value;
		List_t link;
		Node(int x, List_t p = nullptr) :value(x), link(p) {}
	};

	// завантажує з консолі послідовність цілих в звичайному порядку
	List_t readList();
	// виводить елементи списку на консоль
	void writeList(List_t L);
	......
	// вставляє число у впорядкований список
	void insert(int x, List_t& L);
	// об'єднує два впорядковані списки (копіює елементи в новий)
	List_t merge(List_t a, List_t b);
}
```

Тепер ім'я *ListTools*::*Node* означає тип ланка списку, а звичайне ім'я *Node* можна використати для інших потреб, наприклад, для моделювання вузла дерева. Ім'я функції об'єднання двох списків *ListTools*::*merge* відрізняється від імені *merge* об'єднання масивів.

Зміни у файлі *ListTools.cpp* можуть бути дещо суттєвіші. Найпростіший шлях – огорнути всі визначення в простір імен:

```
namespace ListTools
{
	List_t readList()
	{
		// кожне прочитане число дописується в кінець списку
		List_t L = nullptr;
		cout << "Input a succession of integers terminated by 'stop':\n";
		int x;
		cin >> x;
		......
		return L;
	}
	void writeList(List_t L)
	{
		if (L == nullptr)
		{
		......
		cout << '\n';
	}
	.........
}
```

Трохи довший – визначати кваліфіковані імена функцій:

```
void ListTools::insert(int x, List_t& L) // у блоці функції кваліфікатори не потрібні
{
	Node phantom(0, L);
	List_t place = &phantom;
	......
}
```

Звертатися до ресурсів простору імен *ListTools* тепер можна так (у файлі *ArrayTools.cpp*):

```
#include "ListTools.h"
#include "ArrayTools.h"
void sortByList(int * A, size_t n)
{     // щоб упорядкувати масив A, помістимо всі його елементи у впорядкований список
      ListTools::List_t List = nullptr;
      for (size_t i = 0; i < n; ++i) ListTools::insert(A[i], List);
      // а тоді повернемо зі списку назад в масив
      ListTools::List_t curr = List;
      for (size_t i = 0; i < n; ++i, curr = curr->link) A[i] = curr->value;
      removeList(List);
}
```

Класи пам'яті, діапазони доступу, зв'язування імен

Програми мовою C++ використовують три види (класи) пам'яті для зберігання даних. *Автоматичний клас* утворюють локальні змінні функцій (включно з параметрами) та блоків. Пам'ять для них виділяється в стеку автоматично в момент входу в блок, а звільняється – в момент виходу з нього. Часом існування автоматичних змінних керує компілятор і хід виконання програми. *Динамічний клас* складають змінні, які створює програміст за допомогою операторів *new* та *new*[] і звільняє за допомогою *delete* та *delete*[]. Пам'ять для них виділяється в купі (heap), часом життя керує програміст. *Статичний клас* утворюють імена, оголошені поза блоками або зі специфікатором *static*. Існують вони весь час, поки виконується програма.

Діапазон доступу імені або видимість імені (scope) – частина програми, в якій до імені можна звертатися. Діапазон починається в точці оголошення, а закінчується разом з областю, в якій оголошене ім'я: блок, тип (*struct* чи *class*), простір імен, файл. Наприклад, діапазоном доступу параметра циклу *for (int i = ...){...}* є блок інструкції *for*. Діапазоном доступу параметрів *a*, *b*, *c* функції *maxVal*, визначеної на початку цієї лекції, є блок функції *maxVal*. Діапазоном доступу змінної *x* зі сторінки 7 є файл *Source1.cpp*, а функції *ListTools*::*insert* – простір імен *ListTools*.

Імена можуть володіти *зв'язуванням*. Зовнішнє зв'язування мають імена, доступні в межах проекту, в декількох файлах. Внутрішнє зв'язування мають імена, доступні в межах одного файла, можливо, в декількох функціях, оголошених у цьому файлі. Імена без зв'язування «живуть» всередині блока.

Функцію не можна оголосити всередині блока, тому всі функції отримують статичний клас пам'яті і зовнішнє зв'язування. Так само статичний клас пам'яті і зовнішнє зв'язування мають змінні, оголошені поза блоками функцій. Змінні, оголошені у функції, належать до автоматичного класу, мають локальний доступ і не мають зв'язування. Статичні локальні змінні – статичний клас, без зв'язування.

Іменовані константи дуже часто оголошують поза блоками функцій, але модифікатор *const* змінює визначення класів пам'яті. Константа отримує статичний клас і внутрішнє зв'язування. Внутрішнє зв'язування потрібне для того, щоб константи можна було визначати в файлах заголовків і включати до багатьох файлів проекту.

Специфікатор *static* також впливає на зв'язування імен. Глобальна змінна та функція, оголошені з ним, отримають статичний клас пам'яті та внутрішнє зв'язування.

```
int global = 1000;           // статичний клас, зовнішнє зв'язування

static int file_handler = 7; // статичний клас, внутрішнє зв'язування

void funct()                 // статичний клас, зовнішнє зв'язування
{
     static int count = 0;   // статичний клас, без зв'язування
     ......
}
```

```
cons tint fingers = 10;      // статичний клас, внутрішнє зв’язування

static int internal()        // статичний клас, внутрішнє зв’язування
{
     ......
}
```

Додаткову інформацію за цією темою можна знайти в [2, гл. 9].

Література

1. Бублик В.В. Об'єктно-орієнтоване програмування мовою С++.
2. Стивен Прата Язык программирования С++.
3. Дудзяний І.М. Програмування мовою С++.
4. Бьерн Страуструп Язык программирования С++.