Лекція 18

Нові засоби мови C++

Ядро мови C++ і його бібліотеку звичайно стандартизували паралельно. Завдяки цьому бібліотека могла використовувати переваги, надані новими засобами мови, а мова могла поліпшуватися за рахунок досвіду реалізації бібліотеки. З цієї причини стандартна бібліотека мови C++ завжди використовує конкретні засоби мови, яких могло не бути в попередніх версіях стандарту.

Таким чином, мова C++11 відрізняється від мови C++98/C++03, а та, у свою чергу, відрізняється від мови C++, що існувала до початку стандартизації. Якщо не знати про цю еволюцію, то можна здивуватися новим мовним засобам, використовуваним у бібліотеці. У дійсній главі наведений короткий огляд нових засобів мови C++11, що грають важливу роль у проектуванні, розумінні і застосуванні стандартної бібліотеки C++11.

18.1. Нові мовні засоби стандарту C++11

18.1.1. Невеликі, але важливі синтаксичні уточнення

По-перше, варто згадати про двох нових засобах мови C++11, що є невеликими, але важливими для повсякденного програмування.

Пробіли у виразах із шаблонами

Вимога поміщати пробіл між двома закриваючими кутовими дужками в шаблонних виразх скасовано:

```
vector<list<int> >; // OK у кожній версії C++
vector<list<int>>;  // OK, починаючи з версії C++11
```

У книзі (і реальних програмах) можна зустріти обидві ці форми.

Ключове слово nullptr і тип std::nullptr_t

Стандарт C++11 дозволяє використовувати ключове слово `nullptr` замість `0` чи `NULL`, щоб відзначити той факт, що вказівник не посилається на жодний об'єкт (на відміну від ситуації, коли вказівник не має визначеного значення). Це новий засіб мови дозволяє уникнути помилок, що виникають, коли нульовий вказівник інтерпретується як цілочисельне значення. Наприклад:

```
void f(int);
void f(void*);

f(0);       // виклик f(int)
f(NULL);    // виклик f(int), якщо NULL == 0,
            // і неоднозначність у противному випадку
f(nullptr); // виклик f(void*)
```

Слово `nullptr` — це нове ключове слово. Константа, задана за допомогою ключового слова `nullptr`, автоматично конвертується в будь-який тип вказівника, але не в

цілочисельний тип. Вона має тип `std::nullptr_t`, визначений у заголовку `<cstddef>`, так що тепер існує можливість навіть перевантажувати оператори для ситуацій, коли їм передається нульовий вказівник. Відзначимо, що тип `std::nullptr_t` належить до елементарних типів данних.

18.1.2. Автоматичне виведення типу за допомогою ключового слова auto

У мові C++11 можна оголосити змінну чи об'єкт без указівки їх конкретного типу, використовуючи ключове слово `auto`. Розглянемо приклад:

```
auto i = 42;  // Змінна i має тип int
double f();
auto d = f(); // Змінна d має тип double
```

Тип змінної, оголошеної за допомогою ключового слова `auto`, виводиться з її ініціалізатора. Наприклад, у наступному коді потрібен ініціалізатор:

```
auto i; // ПОМИЛКА: неможливо вивести тип змінної i
```

Дозволяється використання додаткових кваліфікаторов. Наприклад:

```
static auto vat = 0.19;
```

Використання ключового слова `auto` особливо корисно в ситуаціях, коли тип є дуже довгим, а вираз складним. Наприклад:

```
vector<string> v;
...
 auto pos = v.begin(); // змінна pos має тип
                       // vector<string>::iterator
 auto l = [] (int x) -> bool { // Змінна l має тип лямбда-функції,
              ...,             // приймаючої int і що повертає bool
         };
```

Змінна `l` є об'єктом, що являє собою лямбда-функцію.

18.1.3. Універсальна ініціалізація і списки ініціалізації

До появи стандарту C++11 програмісти, особливо початківці, легко могли заплутатися в питаннях ініціалізації змінної чи об'єкта. Ініціалізація могла здійснюватися за допомогою круглих чи фігурних дужок, а також операторів присвоювання.

З цієї причини в стандарт C++11 включена концепція універсальної ініціалізації, що означає, що будь-яка ініціалізація здійснюється за допомогою однакового синтаксису. Ця конструкція використовує фігурні дужки. Наприклад:

```
int values[] { 1, 2, 3 };
std::vector<int> v { 2, 3, 5, 7, 11, 13, 17 };
std::vector<std::string> cities {
   "Berlin", "New York", "London", "Braunschweig", "Cairo", "Cologne"
};
std::complex<double> c{4.0,3.0}; // еквівалентно c(4.0,3.0)
```

Список ініціалізації здійснює так називану *ініціалізацію значеннями* (*value initialization*), що припускає, що кожна, навіть локальна змінна елементарного типу, що звичайно має

невизначене початкове значення, ініціалізується нулем (чи константою `nullptr`, якщо змінна є вказівником):

```
int i;     // Змінна i має невизначене значення
int j{};   // Змінна j ініціалізується нулем
int* p;    // Змінна p має невизначене значення
int* q{};  // Змінна q ініціалізується константою nullptr
```

Однак *звужуючі* ініціалізації, тобто ті, що зменшують точність чи модифікують передане значення, у фігурних дужках заборонені. Наприклад:

```
int x1(5.3);      // OK, але x1 стає рівним 5
int x2 = 5.3;     // OK, але x2 стає рівним 5
int x3{5.0};      // ПОМИЛКА: звуження
int x4 = {5.3};   // ПОМИЛКА: звуження
char c1{7};       // OK: незважаючи на те що 7 — ціле значення,
                  // це не звуження
char c2{99999};   // ПОМИЛКА: звуження (якщо 99999 виходить
                  // за діапазон char)
std::vector<int> v1 { 1, 2, 4, 5 };     // OK
std::vector<int> v2 { 1, 2.3, 4, 5.6 }; // ПОМИЛКА: звуження double у int
```

Для підтримки концепції списків ініціалізації для користувальницьких типів стандарт C++11 передбачає клас `std::initializer_list<>`. Його можна використовувати для ініціалізації за допомогою списку чи значень у будь-якім іншім місці, де потрібно список значень. Наприклад:

```
void print (std::initializer_list<int> vals)
 {
   for (auto p=vals.begin(); p!=vals.end(); ++p)
   {    //обробка списку значень
        std::cout << *p << "\n";
   }
 }

 print ({12,3,5,7,11,13,17}); // передача списку значень функції print()
```

При наявності конструкторів як з конкретною кількістю аргументів, так і зі списком ініціалізації, перевага віддається версії зі списком ініціалізації.

```
class P
 {
   public:
    P(int,int);
    P(std::initializer_list<int>);
 };

 P p(77,5);    // виклик P::P(int,int)
 P q{77,5};    // виклик P::P(initializer_list)
 P r{77,5,42}; // виклик P::P(initializer_list)
 P s = {77,5}; // виклик P::P(initializer_list)
```

Якби конструктора зі списком ініціалізації не було, то для ініціалізації об'єктів `q` і `s` був би викликаний конструктор, що одержує два аргументи типу `int`, а ініціалізація об'єкта `r` була б некоректною.

Завдяки спискам ініціалізації тепер ключове слово `explicit` стосується і конструкторів, що отримують більше одного аргументу. Отже, тепер можна заборонити автоматичні перетворення типів декількох значень, що використовувалися також при ініціалізації за допомогою синтаксису присвоювання.

```
class P
{
   public:
     P(int a, int b) {
     ...
     }
     explicit P(int a, int b, int c) {
     ...
     }
 };

 P x(77,5);           // OK
 P y{77,5};           // OK
 P z {77,5,42};       // OK
 P v = {77,5};        // OK (неявне перетворення типу допускається)
 P w = {77,5,42};     // ПОМИЛКА через ключове слово explicit
                     // (неявне перетворення типу не допускається)
 void fp(const P&);
 fp({47,11});         // OK, неявне перетворення {47,11} у P
 fp({47,11,3});       // ПОМИЛКА через ключове слово explicit
 fp(P{47,11});        // OK, явне перетворення {47,11} у P
 fp(P{47,11,3});      // OK, явне перетворення {47,11,3} у P
```

Аналогічно в конструкторах із ключовим словом `explicit`, що одержують список ініціалізації, заборонені неявні перетворення списків ініціалізації — як порожніх, так і утримуючих одне чи несколько значень.

18.1.4. Діапазонні цикли for

Стандарт C++11 увів нову форму циклу `for`, що перебирає всі елементи в заданому діапазоні, чи масиві колекції. В інших мовах програмування така форма циклу називається *foreach*. Загальна синтаксична конструкція цього циклу має наступний вид:

```
for ( decl : coll ) {
    операторы
 }
```

де *decl* — оголошення кожного елемента колекції, що *перебирається,* coll, і до кожного елемента застосовуються зазначені оператори. Наприклад, наступний цикл застосовує до кожного значення переданого списку ініціалізації оператор, що виводить це значення в стандартний потік висновку `cout`:

```
for ( int i : { 2, 3, 5, 7, 9, 13, 17, 19 } ) {
    std::cout << i << std::endl;
 }
```

Для множення кожного елемента `elem` вектора `vec` на `3` можна написати наступний код:

```
std::vector<double> vec;
...
 for ( auto& elem : vec ) {
   elem *= 3;
 }
```

Тут важливо підкреслити, що змінна `elem` оголошена як посилання, тому що інакше оператори в тілі циклу `for` застосовуються до локальних копій елементів у векторі (що іноді буває корисним).

Це значить, що, для того щоб уникнути виклику конструктора копій і деструктора для кожного елемента, необхідно оголосити константне посилання на поточний елемент. Таким чином, узагальнена функція для виводу всіх елементів колекції може бути реалізована в такий спосіб:

```
template <typename T>
 void printElements (const T& coll)
 {
   for (const auto& elem : coll) {
       std::cout << elem << std::endl;
   }
 }
```

Тут діапазонний цикл `for` еквівалентний наступній конструкції:

```
{
   for (auto _pos=coll.begin(); _pos != coll.end(); ++_pos ) {
       const auto& elem = *_pos;
       std::cout << elem << std::endl;
   }
 }
```

У загальному випадку діапазонний цикл `for`, оголошений як

```
for ( decl : coll ) {
   операторu
 }
```

еквівалентний наступній конструкції (якщо об'єкт *coll* має члени `begin()` і `end()`):

```
{
   for (auto _pos=coll.begin(), _end=coll.end(); _pos!=_end; ++_pos ) {
        decl = *_pos;
        операторu
   }
 }
```

чи, якщо таких членів немає, наступній конструкції, у якій використовуються глобальні функції `begin()` і `end()`, що приймають об'єкт *coll як* аргумент:

```
{
   for (auto _pos=begin(coll), _end=end(coll); _pos!=_end; ++_pos ) {
   decl = *_pos;
   операторu
   }
 }
```

У результаті з'являється можливість використовувати діапазонний цикл `for` навіть у випадку списку ініціалізації, оскільки шаблонний клас `std::initializer_list<>` містить члени `begin()` і `end()`.

Крім того, існує правило, що дозволяє використовувати масиви в стилі мови C, що мають відомий розмір. Наприклад, програма

```
int array[] = { 1, 2, 3, 4, 5 };

long sum=0; // накопичує суму всіх елементів
for (int x : array) {
  sum += x;
}

for (auto elem : { sum, sum*2, sum*4 } ) { // виводимо на екран
                                           // декілька добутків суми
                                           // на число
  std::cout << elem << std::endl;
}
```

виводить наступні результати:

```
15
30
60
```

Зверніть увагу на те, що при ініціалізації елементів за допомогою конструкції *decl* у циклі `for` явне перетворення типу стає неможливим. Таким чином, наступний код скомпільований не буде:

```
class C
 {
   public:
   explicit C(const std::string& s); // явне(!) перетворення типу
                                     // із класу string
   ...
};

std::vector<std::string> vs;
for (const C& elem : vs) { // ПОМИЛКА, перетворення класу string
                           // у клас C не визначене
    std::cout << elem << std::endl;
}
```

18.1.5. Семантика переміщення і rvalue-посилання

Одним з найбільш важливих нововведень у стандарті C++11 є підтримка семантики переміщення. Цей засіб дозволяє ще більше наблизитися до основної мети мови C++ і запобігти створення непотрібних копій і тимчасових об'єктів.

Розглянемо наступний приклад:

```
void createAndInsert (std::multiset<X>& coll)
{
  X x;              // створюємо об'єкт типу X
  ...
```

```
   coll.insert(x);  // вставляємо його в передану колекцію
 }
```

Тут ми вставляємо в колекцію новий об'єкт, що має функцію-член, що створює внутрішню копію переданого елемента.

```
namespace std {
   template <typename T, …> class multiset {
      public:
        ... insert (const T& v); // копія об'єкта v

        ...
      };
 }
```

Це корисно, оскільки колекція має семантику значень і передбачає можливість вставляти тимчасові об'єкти чи об'єкти, що використовуються або модифікуються після вставки.

```
X x;
coll.insert(x);   // вставляє копію об'єкта x
...
coll.insert(x+x); // вставляє копію тимчасового rvalue-значення
 ...
coll.insert(x);   // вставляє копію об'єкта x
                  // (хоча об'єкт x більше не використовується)
```

Однак для двох останніх вставок було б добре, щоб передані значення (`x+x` і `x`) більше не використовувалися об'єктом `coll` і він міг уникнути створення их копій і замість цього якимсь образом *перенести* їхній вміст у нові елементи. Це особливо корисно, якщо копіювання об'єкта `x` зв'язано з великими витратами ресурсів — наприклад, якщо він являє собою велику колекцію рядків. У цьому випадку швидкодія програми значно підвищилося б.

У мові C++11 це стало реальністю. Однак програміст зобов'язаний указати, що переміщення можливо, інакше будуть використовуватися тимчасові копії об'єктів. Незважаючи на те що в тривіальних випадках компілятор може самостійно вирішувати такі задачі, новий механізм дозволяє програмісту застосовувати цей засіб у будь-яких ситуаціях, де це вимагає логика. Попередній приклад можна змінити в такий спосіб:

```
X x;
coll.insert(x);            // вставляє копію x
                           // (OK, x як і раніше використовується)
 ...
coll.insert(x+x);          // переміщає (чи копіює) вміст
                           // тимчасового rvalue-значення
 ...
coll.insert(std::move(x)); // переміщає (чи копіює) вміст x
                           // в об'єкт coll
```

За допомогою функції `std::move()`, оголошеної в заголовку `<utility>`, об'єкт `x` можна *перенести*, а не копіювати. Однак сама функція `std::move()` не виконує переміщення, вона просто перетворює свій аргумент у так називане rvalue-посилання (rvalue reference), що представляє собою тип, оголошений із двома амперсандами: `X&&`. Цей новий тип представляє rvalue-значення (анонімні тимчасові об'єкти, що можуть з'являтися тільки в

правій частині оператора присвоювання), що допускають модифікацію. За згодою об'єкт `x` у цьому випадку вважається тимчасовим об'єктом, що уже не потрібний, тому его вміст і/чи ресурси можна захопити.

Тепер колекція може використовувати перевантажені версії функції `insert()`, що працює з rvalue-посиланнями.

```
namespace std {
   template <typename T, ...> class multiset {
     public:
       ... insert (const T& x); // для lvalue-значень: копіює значення
       ... insert (T&& x);      // for rvalue-значень: переміщає значення
       ...
   };
 }
```

Версія для rvalue-значень тепер може бути оптимізована, щоб її реалізація *захоплювала* (steals) вміст об'єкта `x`. Однак для цього потрібна допомога з боку типу об'єкта `x`, оскільки только члени об'єкта `x` мають доступ до його внутрішніх компонентів. Так, наприклад, для ініціалізації вставленого елемента можна використовувати внутрішні масиви і вказівники з об'єкта `x`. Якщо клас об'єкта `x` є складним, це може значно підвищити швидкодію програми. У противному випадку нам довелося б копіювати елемент за елементом. Для ініціалізації нового внутрішнього елемента ми просто викликаємо так називаний конструктор переміщення із класу `X`, що захоплює значення переданого аргументу для ініціалізації нового об'єкта. Усі складні типи повинні — як це зроблено в стандартній бібліотеці C++ — мати такий спеціальний конструктор, що здійснює переміщення вмісту існуючого елемента в новий елемент.

```
class X {
   public:
     X (const X& lvalue); // конструктор, що
     копіює, X (X&& rvalue); // конструктор, що
     перемется ...
};
```

Наприклад, що конструктор переміщення для рядків зазвичай просто привласнює існуючий внутрішній масив символів новому об'єкту, а не створює новий масив і не копіює всі елементи. Те ж саме відноситься і до всіх класів колекцій: замість створення копий всіх елементів ми просто привласнюємо внутрішню пам'ять новому об'єкту. Якщо в класі немає конструктора переміщення, використовується конструктор копій.

Крім того, варто зробити так, щоб будь-яка модифікація — особливо знищення — переданого об'єкта, уміст якого було *захоплено*, не робило впливу на стан нового об'єкта, що став власником значення. Для цього звичайно досить стерти вміст переданого аргументу (наприклад, привласнивши константу `nullptr` его внутрішньому члену, що посилається на його елементи).

Стирання вмісту об'єкта, до якого застосовується семантика переміщення, узагалі говорячи, не є необхідним, але якщо цього не зробити, то весь механізм стає практично

марним. Усі класи стандартної бібліотеки C++ гарантують, що після переміщення об'єкт перебуває в *коректному, але невизначеному стані*. Інакше кажучи, ви можете згодом привласнювати йому нові значення, але його поточне стан не визначений. Контейнери з бібліотеки STL гарантують, що після переміщення значень вони залишаються порожніми.

Точно так само будь-який нетривіальний клас повинний містити оператор присвоювання, що копіює, і оператор присвоювання, що переміщає.

```
class X {
   public:
     X& operator= (const X& lvalue);  // оператор присвоювання, що копіює
     X& operator= (X&& rvalue);       // оператор присвоювання, що переміщає ...
};
```

Для рядків і колекцій ці оператори можна було б реалізувати так, щоб вони просто змінювали местами внутрішній вміст і ресурси. Однак у цьому випадку також варто стерти вміст об'єкта `*this`, тому що він може володіти ресурсами, наприклад блокуваннями, які необходимо як звільнити. Як і в попередніх випадках, з формальної точки зору семантика переміщення не вимагає цього, але, наприклад, контейнерні класи в стандартній бібліотеці C++ роблять це в обов'язковому порядку.

На закінчення зробимо два зауваження про нові засоби мови:: 1) правила перевантаження для rvalue- і lvalue-посилань і 2) повернення rvalue-посилань.

Правила перевантаження для rvalue- and lvalue-посилань

Правила перевантаження для rvalue- і lvalue-значень формулюються в такий спосіб.

- Якщо реалізується тільки функція

```
void foo(X&);
```

без `foo(X&&)`, то її поводження відповідає стандарту C++98: функцію `foo()` можна викликати для lvalue-значень, але не для rvalue-значень.

- Якщо реалізується функція

```
void foo(const X&);
```

без функції `void foo(X&&)`, то її поводження відповідає стандарту C++98: функцію `foo()` можна викликати як для rvalue-значень, так і для lvalue-значень.

- Якщо реалізуються функції

```
void foo(X&);
void foo(X&&);
```

чи

```
void foo(const X&);
void foo(X&&);
```

то rvalue- і lvalue-значення використовуються по-різному. Версія функції для rvalue-значень може і зобов'язана використовувати семантику переміщення. Таким чином, вона може *захоплювати* внутрішній стан і ресурси переданого аргументу.

- Якщо реалізується функція

```
void foo(X&&);
```

але ані `void foo(X&)`, ані `void foo(const X&)` не реалізовані, то функція `foo()` може викликатися для rvalues-значень, але спроба викликати її для lvalue-значення викликає помилку компіляції. Таким чином, тут передбачається тільки семантика переміщення. Ця можливість використовується в бібліотеці: наприклад, унікальними вказівниками, файловими потоками чи строковими потоками.

Це значить, що якщо клас не передбачає семантику переміщення і містить тільки звичайний конструктор копій і оператор присвоювання, що копіює, то він може використовуватися тільки для rvalue-посилань. Таким чином, функція `std::move()` реалізує семантику переміщення, якщо вона передбачена в класі, чи семантику копіювання в іншому випадку.

Повернення rvalue-посилань

Функцію `move()` не обов'язково застосовувати до значень, що повертаються. Відповідно до правил мови в стандарті визначено, що для коду

```
X foo ()
{
  X x;
  ...
  return x;
}
```

гарантується наступне поводження:

- Якщо клас `X` має доступний конструктор копій чи конструктор переміщення, компілятор може ігнорувати копію. Це так називана (*іменована*) *оптимізація значення, що* повертається, ((named) return value optimization — (N)RVO). Вона була визначена ще до появи стандарту C++11 і підтримувалася більшістю компіляторів.
- У противному випадку, якщо в класі `X` є конструктор, що переміщує, то об’єкт x переміщується.
- У противному випадку, якщо в класі `X` є конструктор, що копіює, об’єкт x копіюється.
- У противному випадку виникає помилка компіляції.

Відзначимо, що повернення rvalue-посилання є помилкою, якщо об'єкт, що повертається, є локальним і нестатичним.

```
X&& foo ()
{
   X x;
   ...
   return x; // ПОМИЛКА: повертає посилання на неіснуючий об'єкт
 }
```

Rvalue-посилання — це різновид посилання, і її повернення в момент, коли вона посилається на локальний об'єкт, означає, що ви повертаєте посилання на об'єкт, якого більше немає. Застосування функції `std::move()` на це ніяк не впливає.

18.1.6. Ключове слово noexcept

Стандарт C++11 передбачає ключове слово `noexcept`. Його можна використовувати для того, щоб указати, що функція не може генерувати чи виключення не призначена для цього. Наприклад, код

```
void foo () noexcept;
```

повідомляє, що функція `foo()` не може генерувати виключення. Якщо виключення не було оброблено локально у функції `foo()`, тобто якщо функція `foo()` усе-таки згенерувала виключення, програма припиняє роботу, викликаючи функцію `std::terminate()`, що за замовчуванням викликає функцію `std::abort()`.

18.1.7. Ключове слово constexpr

Починаючи з версії C++11 за допомогою ключового слова `constexpr` можна відзначати вираз, що обчислюються на етапі компіляції. Наприклад:

```
constexpr int square (int x)
 {
   return x * x;
 }
 float a[square(9)]; // OK у версії C++11:
                     // масив a містить 81 елементів
```

Це ключове слово вирішує проблему, що виникла в стандарті C++98 при використанні граничних числових значень. До стандарту C++11 такий вираз, як

```
std::numeric_limits<short>::max()
```

не міг використовуватися як цілочисельна константа, хоча і був функціонально еквівалентним макросу `INT_MAX`. З появою стандарту C++11 такий вираз з'являється з ключовим словом `constexpr`. Наприклад, його можна використовувати для оголошення масиву й обчислень на етапі компіляції (метапрограмування):

```
std::array<float,std::numeric_limits<short>::max()> a;
```

18.1.8. Нові можливості шаблонів

Варіативні шаблони

У стандарті C++11 шаблони можуть мати параметри, що приймають змінну кількість шаблонних аргументів. Ця можливість називається *варіативними шаблонами* (variadic templates). У наступному прикладі их можна використовувати для виклику функції `print()` з аргументами різних типів.

```
void print ()
 {
 }

 template <typename T, typename... Types>
 void print (const T& firstArg, const Types&... args)
 {
   std::cout << firstArg << std::endl; // виводимо на екран перший аргумент
   print(args...);                     // викликаємо print() для
```

```
                                        // інших аргументів
}
```

При передачі одного чи декількох аргументів використовується шаблонна функція, у якій окреме визначення першого аргументу дозволяє виводити його на екран, а потім рекурсивно викликати функцію `print()` з аргументами, що залишилися. Для завершення рекурсії дається нешаблонна перевантажена версія функції `print()`.

Наприклад, виклик функції

```
print (7.5, "hello", std::bitset<16>(377), 42);
```

приводить до наступного результату:

```
7.5
hello
0000000101111001
42
```

Відзначимо, що коректність цього прикладу в даний час також є предметом обговорення. Причина полягає в тім, що варіативна форма з одним аргументом формально збігається з неваріативною формою з одним аргументом; однак компілятори звичайно допускають наступний код:

```
template <typename T>
void print (const T& arg)
{
  std::cout << arg << std::endl;
}

template <typename T, typename... Types>
void print (const T& firstArg, const Types&... args)
{
  std::cout << firstArg << std::endl; // виводимо на екран перший аргумент
  print(args...);                     // викликаємо print() для
                                      // інших аргументів
}
```

У варіативних шаблонах оператор `sizeof...(args)` повертає кількість аргументів.

Цю функціональну можливість інтенсивно використовує клас `std::tuple<>`.

Шаблонний псевдонім (шаблонна оператор typedef)

У стандарті C++11 підтримуються визначення шаблонних (часткових) типів. Однак, оскільки всі підходи, засновані на використанні ключового слова `typename`, з деякої причини виявилися непрацездатними, було введено ключове слово `using` і термін *псевдонім шаблона* (alias template). Наприклад, після операторів

```
template <typename T>
using Vec = std::vector<T,MyAlloc<T>>; // стандартний вектор, що щовикористовує,
                                       // власний механізм розподілу пам'яті
```

вираз

```
Vec<int> coll;
```

стає еквівалентним виразу

```
std::vector<int,MyAlloc<int>> coll;
```

Інші нові шаблонні засоби

У стандарті C++11 шаблонні функції можуть мати шаблонні аргументи, задані за замовчуванням. Крім того, локальні типи тепер можна використовувати як шаблонні аргументи, а функції з внутрішнім зв'язуванням тепер можна використовувати як аргументи для нетипізованих шаблонів чи вказівників посилань на функції.

18.1.9 Лямбда-вирази і лямбда-функції

У стандарті C++11 з'явилися *лямбда-вирази* і *лямбда-функції*, що дозволяють створювати визначення функцій, що підставляються, які можна використовувати як параметр чи локальний об'єкт.

Лямбда-вирази і лямбда-функції змінюють спосіб використання стандартної бібліотеки C++. Наприклад, можна використовувати лямбда-вирази і лямбда-функції з алгоритмами і контейнерами з бібліотеки STL.

Синтаксис лямбда-виразів і лямбда-функцій

Лямбда-функція — це опис функціональної можливості, яку можна визначити в операторі і виразі. Таким чином, лямбда-функцію можна використовуватияк функцію, що підставляється.

Мінімальна лямбда-функція не має параметрів і просто робить щось. Наприклад:

```
[] {
   std::cout << "hello lambda" << std::endl;
 }
```

Цю функцію можна викликати безпосередньо

```
[] {
   std::cout << "hello lambda" << std::endl;
 } (); // виводить на екран "hello lambda"
```

чи передавати об'єктам

```
auto l = [] {
   std::cout << "hello lambda" << std::endl;
};
...
 l(); // виводимо на екран "hello lambda"
```

Як бачимо, лямбда-функції завжди передує так називаний *ініціатор лямбда-функції* (lambda introducer): квадратні дужки, усередині яких можна визначити так називане захоплення для доступу до нестатичних зовнішніх об'єктів у лямбда-функції. Якщо доступ до зовнішніх даних не потрібний, квадратні дужки залишаються порожніми, як у даному випадку. У лямбда-функціях можна використовувати статичні об'єкти, наприклад `std::cout`.

Між ініціатором лямбда-функції і тілом лямбда-функції можна вказати параметри, ключове слово `mutable`, специфікацію винятку, специфікатори атрибутів і тип значення, що повертається. Усе це не обов’язково, але якщо один з перерахованих пунктів присутній,

круглі дужки для параметрів стають обов'язковими. Таким чином, синтаксис лямбда-функції виглядає або так:

```
[...] {...}
```

або так:

```
[...] (...) mutable_opt throwSpec_opt ->retType_opt {...}
```

Лямбда-функція може мати параметри, зазначені в дужках, як звичайна функція.

```
auto l = [] (const std::string& s) {
             std::cout << s << std::endl;
         };
 l("hello lambda"); // виводить на екран "hello lambda"
```

Однак варто підкреслити, що лямбда-функції не можуть бути шаблонними. У них завжди необхідно указувати всі типи.

Лямбда-функція може повертати який-небудь об'єкт. Якщо тип об'єкта, що повертається, не зазначений, він виводиться з його значення. Наприклад, наступна функція повертає значення типу `int`:

```
[] {
   return 42;
 }
```

Для вказівки типу значення, що повертається, можна використовувати нову синтаксичну конструкцію мови С++, що застосовується і для звичайних функцій. Наприклад, наступна лямбда-функція повертає число `42.0`:

```
[] () -> double {
   return 42;
 }
```

У даному випадку необхідно вказати тип значення, що повертається, після обов'язкових у даному випадку дужок, призначених для аргументів, і символів ->.

Між параметрами і типом значення, що повертається, чи тілом можна також задати специфікацію винятку, як при роботі зі звичайною функцією. Однак для звичайних функцій специфікації винятку оголошені застарілими.

Захоплення (доступ до зовнішньої області видимості)

В ініціаторі лямбда-функції (квадратних дужках перед лямбда-функцією) можна задати *список захоплення* (capture) для доступу до даних із зовнішньої області видимості, що не передаються як аргументи.

- Символи `[=]` означають, що зовнішня область видимості передається в лямбда-функцію за значенням. Таким чином, можна прочитати, але не модифікувати все дані, що були доступними для читання при визначенні лямбда-функції.
- Символи `[&]` означають, що зовнішня область видимості передається в лямбда-функцію по посиланню. Таким чином, при визначенні лямбда-функції можна змінювати дані, що були коректними в момент визначення лямбда-функції, за умови, що їхній можна змінювати в принципі.

Крім того, для кожного об'єкта в лямбда-функції необхідно вказати режим доступу до нього: за значенням чи за посиланням. Це дозволяє обмежувати доступ і змішувати різні режими. Наприклад, оператори

```
int x=0;
int y=42;
auto qqq = [x, &y] {
               std::cout << "x: " << x << std::endl;
               std::cout << "y: " << y << std::endl;
               ++y; // OK
           };
x = y = 77;
qqq();
qqq();
std::cout << "final y: " << y << std::endl;
```

приводять до наступних результатів:

```
x: 0
y: 77
x: 0
y: 78
final y: 79
```

Оскільки об'єкт `x` передається за значенням, його не можна модифікувати в лямбда-функції; оператор `++x` у лямбда-функції не скомпілюється. Оскільки об'єкт `y` передається по посиланню, його можна модифікувати, тому подвійний виклик лямбда-функції двічі збільшить привласнене початкове значення `77`.

Замість списку захоплення `[x, &y]` можна написати список `[=, &y]` і передати об'єкт `y` за посиланням, а всі інші об'єкти — за значенням.

Для того щоб змішати передачу за посиланням і за значенням, лямбда-функцію можна оголосити з ключовим словом `mutable`. У цьому випадку об'єкти передаються за значенням, але у функції-об'єкті, визначеній в лямбда-функції, передане значення можна змінити. Наприклад, код

```
int id = 0;
auto f = [id] () mutable {
          std::cout << "id: " << id << std::endl;
          ++id; // OK
         };
id = 42;
f();
f();
f();
std::cout << id << std::endl;
```

виводить наступні результати:

```
id: 0
id: 1
id: 2
42
```

Лямбда-функцію можна порівняти з функцією-об'єктом.

```
class {
   private:
     int id; // копіювання зовнішнього id
   public:
     void operator() () {
         std::cout << "id: " << id << std::endl;
      ++id; // OK
   }
 };
```

Завдяки ключовому слову `mutable` оператор виклику функції `operator()` визначений як неконстантна функція-член, тобто існує можливість змінювати значення змінної `id`. Отже, завдяки ключовому слову `mutable` лямбда-функція зберігає поточний стан, навіть якщо стан передається за значенням. Без ключового слова `mutable`, що є звичайною ситуацією, оператор виклику функції `operator()` стає константною функцією-членом, і ви маєте право лише читати об'єкти, передані за значенням.

Типи лямбда-функцій

Типом лямбда-функції є анонімна функція-об'єкт (чи функтор), що є унікальною для кожного лямбда-виразу. Таким чином, для оголошення об'єктів такого типу необхідні шаблони чи ключове слово `auto`. Якщо необхідний тип, можна використовувати ключове слово `decltype`, що, наприклад, потрібно при передачі лямбда-функції як хеш-функції чи критерію сортування для асоціативних чи неупорядкованих контейнерів.

Як альтернативу для вказівки загального типу, призначеного для функціонального програмування, можна використовувати шаблонний клас `std::function<>`, наданий стандартною бібліотекою мови C++. Цей шаблонний клас забезпечує єдиний спосіб задати тип значення, що повертається лямбда-функцією:

```
// lang/lambda1.cpp

 #include<functional>
 #include<iostream>

 std::function<int(int,int)> returnLambda ()
 {
   return [] (int x, int y) {
              return x*y;
          };
 }

 int main()
 {
   auto lf = returnLambda();
   std::cout << lf(6,7) << std::endl;
 }
```

Нижче приведений результат роботи цієї програми.

```
42
```

18.1.10. Ключове слово decltype

За допомогою ключового слова `decltype` можна дозволити компілятору самому розпізнати тип виразу. Це реалізація часто використовуваного оператора `typeof`. Оскільки існуючі реалізації оператора `typeof` суперечливі і неповні, стандарт C++11 увів нове ключове слово. Наприклад:

```
std::map<std::string,float> coll;
decltype(coll)::value_type elem;
```

Одне з застосувань ключового слова `decltype` — оголошення типів значень, що повертаються. Інше застосування стосується метапрограммування і передачі типу лямбда-функції.

18.1.11. Новий синтаксис оголошення функцій

Іноді тип значення, що повертається функцією, залежить від виразу, що має аргументи. Однак код на зразок

```
template <typename T1, typename T2>
decltype(x+y) add(T1 x, T2 y);
```

до появи стандарту C++11 був неможливий, оскільки тип значення, що повертається, у цьому випадку використовує об'єкти, що не оголошені чи ще не з'явилися в області видимості.

У стандарті C++11 існує альтернативне оголошення типу значення, що повертається функцією, що розміщається після списку параметрів.

```
template <typename T1, typename T2>
auto add(T1 x, T2 y) -> decltype(x+y);
```

Це оголошення використовує той же синтаксис, що і функції-лямбда-функції при оголошенні типів значень, що повертаються.

18.1.12. Перерахування з обмеженою областю видимості

Стандарт C++11 дозволяє визначати *перерахування з обмеженою областю видимості* (scoped enumerations), відомі також як *строгі перерахування* (strong enumerations) чи *класи перерахувань* (enumeration classes). Ці перерахування є більш точною реалізацією значень, що перелічуються, (перечисельників) у мові C++.

Наприклад:

```
enum class Salutation : char { mr, ms, co, none };
```

Варто підкреслити, що ключове слово `class` після `enum` є обов'язковим.

Перерахування з обмеженою областю видимості мають наступні переваги.

- Неявні перетворення в тип `int` і з типу `int` неможливі.
- Значення накшталт `mr` не є частиною області видимості, у якій оголошене перерахування. Замість нього варто використовувати конструкцію `Salutation::mr`.
- Можна явно визначити базовий тип (у даному випадку `char`) і гарантувати розмір елементів перерахування (якщо пропустити вираз «`: char`», то за замовчуванням буде використовуватися тип `int`).

- Можливі попередні оголошення перелічуваних типів, що виключає необхідність повторної компіляції модулів з новими значеннями перерахувань за умови, що використовується тільки даний тип.

Зверніть увагу на те, що властивість типу `std::underlying_type` дозволяє визначити базовий тип перерахування.

18.1.13. Нові фундаментальні типи даних

У стандарті C++11 визначені нові фундаментальні типи даних.

- `char16_t` і `char32_t`
- `long long` і `unsigned long long`
- `std::nullptr_t`

18.2. Старі “нові” засоби мови

Незважаючи на те що стандарту C++98 уже більше десяти років, програмісти усе ще іноді дивуються деяким засобам мови. Розглянемо деякі з них.

Нетипізовані шаблонні параметри

Крім типізованих параметрів допускаються нетипізовані. Нетипізований параметр розглядається як частина типу. Наприклад, стандартному класу `bitset<>` можна передати кількість бітів у виді шаблонного аргументу. Наступні оператори визначають два бітових поля: одне з 32 битов, інше — з 50:

```
bitset<32> flags32; // бітове поле з 32 битов
bitset<50> flags50; // бітове поле з 50 битов
```

Ці бітові множини мають різні типи, тому що вони використовують різні шаблонні аргументи. Таким чином, їх не можна привласнювати один одному і порівнювати між собою, не передбачивши відповідного перетворення типу.

Шаблонні параметри, задані за замовчуванням

Шаблонні класи можуть мати аргументи, задані за замовчуванням. Наприклад, наступне оголошення дозволяє оголошувати об'єкти класу `MyClass` з одним чи двома шаблонними аргументами:

```
template <typename T, typename container = vector<T>>
 class MyClass;
```

Якщо передати тільки один аргумент, то другим буде використовуватися параметр, заданий за замовчуванням.

```
MyClass<int> x1; // еквівалент MyClass<int,vector<int>>
```

Відзначимо, що шаблонні аргументи, задані за замовчуванням, можна визначати за допомогою попередніх аргументів.

Ключове слово typename

Ключове слово `typename` було введено для того, щоб указати, що наступний за ним ідентифікатор позначає тип. Розглянемо наступний приклад:

```
template <typename T>
class MyClass {
   typename T::SubType * ptr;
   ...
 };
```

де ключове слово `typename` використовується для того, щоб роз'яснити, що `SubType` — це тип, визначений у класі `T`. Таким чином, `ptr` — це вказівник на об'єкт типу `T::SubType`. Без ключового слова `typename` ім'я `SubType` розглядалося б як ім'я статичного члена, а виходить, вираз

```
T::SubType * ptr
```

означало б множення значення об'єкта `SubType` типу `T` на змінну `ptr`.

Оскільки в програмі зазначено, що `SubType` — це тип, то будь-який тип, використовуваний замість параметра `T`, повинний мати внутрішній тип `SubType`. Наприклад, використання типу `Q` як шаблонний аргумент можливо только в тому випадку, якщо він містить визначення внутрішнього типу `SubType`.

```
class Q {
   typedef int SubType;
   ...
 };

 MyClass<Q> x; // OK
```

У такому випадку змінна `ptr`, що є членом класу `MyClass<Q>`, представляла б собою вказівник на тип `int`. Однак підтип також може бути абстрактним типом даних, таким, як клас:

```
class Q {
   class SubType;
   ...
 };
```

Відзначимо, що ключове слово `typename` необхідно використовувати завжди, якщо потрібно кваліфікувати ідентифікатор шаблона як тип, навіть якщо інша інтерпретація позбавлена змісту. Отже, загальне правило в мові C++ затверджує, що будь-який ідентифікатор шаблона розглядається як значення, якщо він не кваліфікований ключовим словом `typename`.

Крім усього іншого, ключове слово `typename` можна використовувати замість ключового слова `class` в оголошенні шаблона.

```
template <typename T> class MyClass;
```

Шаблонні члени

Член^-функції-члени класу можуть бути шаблонними. Однак шаблонні функції-члени не можуть бути віртуальними. Наприклад:

```
class MyClass {
   ...
   template <typename T>
   void f(T);
 };
```

де вираз `MyClass::f` повідомляє набір функцій-членів з параметрами будь-якого типу. Ви можете передавати їй будь-який аргумент, за умови, що его тип передбачає всі оператори, виконувані у функції `f()`.

Це властивість часта використовується для підтримки автоматичного перетворення типів членів у шаблонних класах. Наприклад, у наступному визначенні аргумент `x` функції `assign()` повинний мати точно такий же тип, як і зухвалий об'єкт:

```
template <typename T>
 class MyClass {
   private:
     T value;
   public:
     void assign (const MyClass<T>& x) { // x повинний мати такий же
                                         // тип, що і *this
        value = x.value;
     }
     ...
 };
```

При такім оголошенні було би помилкою використовувати різні шаблонні типи для об'єктів у функції `assign()`, навіть якщо передбачене автоматичне перетворення одного типу в іншій.

```
void f()
 {
   MyClass<double> d;
   MyClass<int> i;

   d.assign(d); // OK
   d.assign(i); // ПОМИЛКА: i має тип MyClass<int>
                // а потрібно тип MyClass<double>
}
```

Задавши інший шаблонний тип для функції-члена, можна послабити вимога строгого збігу типів. Шаблонний аргумент функції-члена може мати будь-як шаблонний тип. Тоді, оскільки ці типи допускають присвоювання, можна написати наступний код.

```
template <typename T>
 class MyClass {
   private:
     T value;
   public:
     template <typename X>               // Шаблонний член
     void assign (const MyClass<X>& x) { // Дозволені інші шаблонні типи
         value = x.getValue();
     }
     T getValue () const {
        return value;
     }
     ...
 };

 void f()
```

```
{
    MyClass<double> d;
    MyClass<int> i;

    d.assign(d); // OK
    d.assign(i); // OK (int можна привласнювати змінної типу double)
}
```

Відзначимо, що аргумент `x` функції `assign()` тепер відрізняється від типу об'єкта `*this`. Таким чином, у нас немає безпосереднього доступу до закритих і захищених членів класу `MyClass<>`. Замість цього ми повинні використовувати щось вроде функції `getValue()`.

Особливим видом шаблонного члена є *шаблонний конструктор* (template constructor). Шаблонні конструктори звичайно використовуються для здійснення неявних перетворень типу при копіюванні об'єктів. Зверніть увагу на те, що шаблонний конструктор не придушує неявне оголошення конструктора, що копіює. Якщо типи збігаються точно, генерується і викликається неявний конструктор, що копіює. Наприклад:

```
template <typename T>
 class MyClass {
 public:
   // конструктор, що копіює, з неявним перетворенням
   типів // не придушує автоматичний конструктор, що
   копіює, template <typename U
> MyClass (const MyClass<U>& x)
; ...
};

void f()
{
   MyClass<double> xd;
   ...
   MyClass<double> xd2(xd); // викликає неявно згенерований
                            // конструктор, що
   копіює, MyClass<int> xi(xd); // викликає шаблонний
   конструктор ...
}
```

Тут тип об'єкта `xd2` збігається з типом об'єкта `xd` і тому ініціалізується за допомогою неявно сгенерированного конструктора, що копіює. Тип об'єкта `xi` відрізняється від типу об'єкта `xd` і тому ініціалізується шаблонним конструктором. Таким чином, якщо ви реалізуєте шаблонний конструктор, не забудьте передбачити конструктор за замовчуванням, якщо его поводження за замовчуванням вас не влаштовує. Інший приклад шаблонних членів класу приведений у розділі 5.1.1.

Вкладені шаблонні класи

Вкладені класи також можуть бути шаблонними.

```
template <typename T>
class MyClass {
```

```
    ...
    template <typename T2>
    class NestedClass;
    ...
};
```

18.2.1. Неявна ініціалізація фундаментальних типів

Якщо ви використовуєте синтаксис явного виклику конструктора без аргументів, об'єкти фундаментальних типів инициализируются нулем.

```
int i1;          // невизначене значення
 int i2 = int();  // ініціалізується нулем
 int i3{};        // ініціалізується нулем (по стандарті C++11)
```

Цей засіб дозволяє писати шаблонний код, що гарантує, що змінні будь-якого типу завжди будуть мати визначене значення, задане за замовчуванням. Наприклад, у наступній функції ініціалізація гарантує, що об'єкт `x` фундаментального типу ініціалізується нулем:

```
template <typename T>
void f()
{
   T x = T();
   ...
 }
```

Якщо шаблонна функція виконує примусову ініціалізацію нулем, то значення, що повертається нею, називається *значенням, що ініціалізоване нулем* (zero initialized). У противному випадку воно *ініціалізується значенням за замовчуванням*.

18.2.2. Визначення функції main()

Варто роз'яснити один важливий аспект мови, що часто розуміють неправильно: коректні і стерпні версії функції `main()`. У відповідності зі стандартом мови С++ стерпними є тільки два визначення функції `main()`.

```
int main()
{
   ...
 }
```

і

```
int main (int argc, char* argv[])
{
   ...
 }
```

Тут аргумент `argv` (масив аргументів командного рядка) можна також визначити як `char**`. Відзначимо, що у функції `main` в обов'язковому порядку потрібно тип значення, що `повертається,` int.

Функцію `main()` не обов'язково завершувати оператором `return`. На відміну від мови С, у мові С++ наприкінці функції `main()` неявно визначений оператор

```
return 0;
```

Це значить, що кожна програма, що здійснює вихід з функції `main()` без виконання оператора `return`, вважається завершеної успешно. Будь-яке значення, що відрізняється від нуля, означає помилку. У зв'язку з этим мої приклади в книзі ніколи не містять оператор `return` наприкінці функції `main()`.

Для завершення програми мовою C++ без виходу з функції `main()` звичайно варто використовувати функції `exit()`, `quick_exit()` (починаючи зі стандарту C++11) чи `terminate()`.

Література

Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. — М.: Вильямс, 2005. — глава 3.