Лекція 11: Шаблони функцій та шаблони класів

Потреба в шаблонах функцій та шаблонах класів

Одним з недоліків мов програмування з типізацією є те, що коли визначається певна функція потрібно жорстко фіксувати для яких типів ця функція визначена та який результат вона повертає. Насправді, звичайно, це інколи гарно — бо гарантує виконання функції лише власне для тих типів для яких вона призначена. Але інколи, як наприклад для функції

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

було б непогано зробити так, щоб не було потреби визначати її окремо і для інших стандартних типів — long long, unsigned, string і так далі, а також можливо й для деяких нестандартних (для тих з них, де визначена операція порівняння), бо ми бачимо що код функції max для них буде такий самий.

Для тих з типізованих мов, що є об'єктно-орієнтовними та всі типи є наслідниками якогось базового класу можна спробувати реалізувати цю функцію для всіх типів з допомогою перетворення вниз, але в Сі++ для стандартних типів така можливість відсутня.

Ще одна причина появи синтаксису шаблонів є створення класів для реалізації стандартних контейнерів даних, наприклад, стеку.

При створенні таких структур даних на Сі завжди потрібно вказувати який тип даних в цій структурі використовується, наприклад:

```
struct Stack_{
  int data;
 struct Stack_ * next;
} Stack;
```

А отже, якщо потрібно визначити таку саму структуру з тими ж самими функціями (методами) потрібно знов створювати нову структуру та переписувати той самий код з іншим типом, що, звичайно, є погано.

Звичайно, навіть синтаксис Сі дозволяє обійтись без повторення коду за допомогою певних хитрощів, але такі варіанти є достатньо нетривіальними, тому починаючи з 98-го стандарту додали нову властивість синтаксису, яка зветься шаблони.

Шаблони функцій

Шаблон функції – це опис функції, яка залежить від даних довільного типу. Під час виклику такої функції компілятор автоматично проаналізує тип фактичних аргументів, згенерує для них програмний код. Це називається неявним створенням **екземпляру шаблону**.

Запис:

template <class Тип1, class Тип2,…, class ТипN>

Тип Функції НазваФункції(Тип1 аргумент1, Тип2 аргумент2,…,ТипN аргументN);

або

template <typename Тип1, typename Тип2,…, typename ТипN>

Тип_Функції НазваФункції(Тип1 аргумент1, Тип2 аргумент2,…,ТипN аргументN);

це оголошення шаблону.

В цьому визначенні:

- template - ключове слово, що вказує на визначення шаблону,
- Тип1, …, ТипN – назви деяких узагальнюючих типів.
- class або typename - ключові слова, що вказують на те що цей тип є узагальненим.
- Тип_Функції НазваФункції(Тип1 аргумент1, Тип2 аргумент2,…,ТипN аргументN) – опис визначення функції в рамках синтаксісу Сі++. У списку формальних параметрів можуть бути присутні як параметри узагальнюючих типів так і стандартні або користувацькі типи.

Якщо шаблон описаний перед головною програмою, то він, як і звичайна функція, оголошення не потребує.

Підхід у програмуванні, що грунтується на використанні шаблонів функцій називається **узагальнюючим програмуванням**.

Примітка. Ключові слова class та typename абосолютно рівносильні (синоніми) та немає ніякої різниці яким з них користуватись.

Приклад.

```
#include <iostream>
#include <cstring>
using namespace std;
// шаблон функції для пошуку максимального значення в масиві
template <typename T> T maxArray(const T* array, size_t size){
   T max = array[0]; // максимальне значення в масиві
   for (int ix = 0; ix < size; ix++)
      if (max < array[ix])
         max = array[ix];
   return max;
```

```
}
int main(){
   // тестуємо шаблон функції maxArray для масиву типу char
   char array [] = "aodsiafgerkeio";
   int len = strlen(array);
   cout << "Максимальний елемент масиву типу char: " << maxArray(array, len) << endl;
   // тестуємо шаблон функції maxArray для масиву типу int
   int iArray [5] = {3,5,7,2,9};
   cout << " Максимальний елемент масиву типу int: " << maxArray(iArray, 5) << endl;
}
```

Тут у нас в шаблоні функції використовувалися вбудовані типи даних, тому в рядку ми написали template <typename T> (або <class T>). Замість T можна підставити будь-яке інше ім'я, яке є коректним ідентифікатором.

У рядку template <typename T> T maxArray(const T* array, int size) виконується визначення шаблону з одним параметром - T, причому цей параметр буде мати один з вбудованих типів даних, так як вказано ключове слово typename.

Нижче, оголошена функція, яка відповідає всім критеріям оголошення звичайної функції, є заголовок, є тіло функції, в заголовку є ім'я і параметри функції, все як завжди. Але що цю функції перетворює в шаблон функції, так це параметр з типом даних T, це єдиний зв'язок з шаблоном, оголошеним раніше. Якби ми написали

```
int maxArray(const int* array, size_t size){
   int max = array[0]; // максимальне значення в масиві
   for (int ix = 0; ix < size; ix++)
      if (max < array[ix])
         max = array[ix];
   return max;
}
```

то отримали б звичайну цілу функцію від цілих аргументів,

Аналогічно, при T рівним std::string отимаємо

```
std::string maxArray(const std::string* array, size_t size){
   std::string max = array[0]; // максимальне значення в масиві
   for (int ix = 0; ix < size; ix++)
      if (max < array[ix])
         max = array[ix];
```

```
    return max;
}
```

Тобто, T — це навіть не тип даних, а зарезервоване місце під будь-який вбудований тип даних. Тобто коли виконується виклик цієї функції, компілятор аналізує параметр шаблону функції і створює екземпляр для відповідного типу даних: int, char і так далі.

Примітка. Слід розуміти, що навіть якщо обсяг коду менше, то це не означає, що пам'яті програма буде споживати менше. Компілятор сам створює локальні копії функції-шаблону і відповідно пам'яті споживається стільки, як якщо б ви самі написали всі екземпляри функції, як у випадку з перевантаженням.

Відзначимо тієї факт, що поки немає виклика функції-шаблоном, при компіляції вона в бінарному коді не створюється (**не інстанціюється**). А якщо оголосити групу викликів функції зі змінними різних типів, то для кожного компілятор створить свою реалізацію на базі шаблону.

Виклик шаблонної функції, в загальному випадку, відповідає виклику звичайної функції. В цьому випадку компілятор визначить, який тип використовувати замість типу T, на підставі визначення типів фактичних параметрів. Але якщо підставлені параметри виявляться різних типів, то компілятор не зможе вивести (**інстанціювати шаблон**) реалізацію шаблону. Так, в нижче наступному коді компілятор “спіткнеться” на третьому виклику, так як не може визначити, чому дорівнює Type

```
#include <iostream>
template<class Type>
Type _min(Type a, Type b) {
    return (a < b) ?a:b;
}

int main(int argc, char** argv) {
    std::cout << _min(1, 2) << std::endl;
    std::cout << _min(3.1, 1.2) << std::endl;
    std::cout << _min(5, 2.1) << std::endl; /* error: no matching function for call to ‘_min(int, double)’*/
    }
```

Вирішується ця проблема вказанням конкретного типу при виклику функції.

```
int main(int argc, char** argv) {
    std::cout << _min<double>(5, 2.1) << std::endl;
```

```
}
```

Примітка. Не всі шаблони завжди можуть бути коректно інстанційовані. Дійсно, компілятор просто підставляє потрібний тип в шаблон. Але чи завжди отримувана функція буде працездатна? Очевидно, що ні. Будь алгоритм може бути визначений незалежно від типу даних, але він обов'язково користується властивостями цих даних. У випадку з шаблонної функцією _min це вимога визначення оператора упорядкування (оператор <).

Будь-який шаблон функції передбачає наявність певних властивостей параметризованого типу, в залежності від реалізації (наприклад, оператору копіювання, оператору порівняння, наявності певного методу і т.д.). В очікуваному стандарті мови Сі++ за це будуть відповідати концепції.

Ще один, більш складний приклад. Визначимо шаблон функції від трьох типів:

```
#include <iostream>
template <class T1, class T2, class T3>
T3 strange_mul(T1 x, T2 y) {
   T3 res = static_cast<T3>(x) * static_cast<T3>(y);
   return res;
}
```

Тут ми визначили шаблон функції, який робить множення двох змінних різних типів, спочатку зводячи їх до третього типу. Ця функція буде коректно інстанційована, наприклад, наступними визначеннями типів та викликами:

```
long long y1 =  strange_mul<int, unsigned,long long>(1,2);
double y2 =  strange_mul<float, int, double>(1.0f,2);
```

Але у наступних випадках, вона повинна «впасти» при виконанні:

```
double y3 =  strange_mul<float, string, double>(1.0f,"2"); //error: invalid static_cast
string y4 =  strange_mul<string, string, string>("1","2"); // error: no match for ‘operator*’
```

В першому випадку, вона «впаде», бо не зможе виконати перетворення типу static_cast<double>, а в другому бо не зможе виконати множення для типу string.

В усіх схожих випадках відповідальність за коректність шаблону кладеться на плечі програміста, бо програму буде стикатись з помилкою безпосередньо на етапі виконання.

Перевантаження шаблону функції

Шаблони функцій також можна перевантажувати іншими шаблонами функцій, змінивши кількість переданих параметрів в функцію. Ще однією

особливістю перевантаження є те, що шаблонні функції можуть бути перевантажені зазвичай не шаблонними функціями. Тобто вказується те саме ім'я функції, з тими ж параметрами, але для певного типу даних, і все буде коректно працювати.

Шаблони функцій також можуть перевантажуватися.

Зазвичай дана перевантаження виконується при довизначенні шаблонів до вказівників, як на прикладі.

```
template<class Type> Type* _min(Type* a, Type* b){
  return (*a < *b)?*a:*b;
    }
```

Іншим варіантом, коли треба перевантажити шаблон — це спеціалізація шаблонної функції.

У деяких випадках шаблон функції є неефективним або неправильним для певного типу. В цьому випадку можна спеціалізувати шаблон, - тобто написати реалізацію для даного типу. Наприклад, у випадку з рядками можна вимагати, щоб функція порівнювала тільки кількість символів. У разі спеціалізації шаблону функції тип, для якого уточнюється шаблон в параметрі не вказується. Нижче наводиться приклад зазначеної спеціалізації.

```
template<>
std::string _min(std::string a, std::string b){
  if(a.size() < b.size()){
    return a;
  }
  return b;
}
```

Спеціалізація шаблону для конкретних типів робиться знову ж з міркування економічності: якщо ця версія шаблону функції в коді не використовується, то вона не буде включена в бінарний код.

Примітка. Для параметрів шаблону, так само як для параметрів функції є можливість задавати параметри за замовченням, щоправда для функцій ця можливість додана лише зі стандарту C++11.

Приклад

```
#include <iostream>
template <typename T=int> T func(T x, int y){
  T res = x / y;
  return res;
}
int main(){
int x=11,y=2,z;
double a=11.0,b,c;
```

```
z  = func(x,y);  //  5 - ціле
b = func<double>(x,y); // 5.5 - дійсне
c= func(a,y); // 5.5 - дійсне
}
```

Стандартні шаблони функцій

Деякі популярні шаблони функцій вже присутні в стандартній бібліотеці Сі++, зокрема:

1) Шаблон максимуму (бібліотека <algorithm.h>):

```
template<class T>
const T& max(const T& a, const T& b){
   return (a < b) ? b : a;
}
```

Використання:

```
int z1 = max(1,2);
double x=1.0, y=2.0;
y = max(x,y);
string z = max("A","ABC");
```

2) Шаблон мінімуму (бібліотека <algorithm.h>):

```
int z1 = min(1,2);
```

3) Шаблон функції обміну значеннями двох змінних (C++98: <algorithm>, C++11: <utility>)

```
// swap algorithm example (C++98)
#include <iostream>    // std::cout
#include <algorithm>   // std::swap

int main () {
  int x=10, y=20;   // x:10 y:20
  std::swap(x,y);   // x:20 y:10
}
```

Шаблони класів

Аналогічно як при створенні функцій, так і при розробці класів для різних типів даних, потрібно писати програмний код для кожного типу окремо. При цьому часто методи і операції над даними різних типів можуть містити один і той же повторюваний код. Щоб уникнути повторюваності написання коду для різних типів даних, в мові C++ використовуються так звані **шаблони (templates) класів**.

Шаблон класу дозволяє оперувати даними різних типів в загальному випадку. Тобто, немає прив'язки до певного конкретного типу даних (int, float, ...). Вся робота виконується над деякими узагальненим типом даних, наприклад типом з ім'ям T.

Фактично, оголошення шаблону класу є тільки описом. Створення реального класу з заданим типом даних виконується компілятором в момент компіляції, коли об’являється об’єкт класу.

Загальна форма декларації шаблонного класу має наступний вигляд:

ClassName <*types*> *objName*;

де

- *ClassName* – ім’я шаблонного класу;
- *types* – типи даних в програмі;
- *objName* – ім’я об’єкту (екземпляру) класу.

Ключове слово class може бути замінено на слово typename. Тоді загальна форма декларації шаблонного класу може бути наступною:

```
template <typename T1, typename T2, …, typename Tn >
class ClassName {
   // тіло класу
   // ...
}
```

або

```
template <class T1, class T2, …, class Tn> class ClassName
{
   // тіло класу
   // ...
}
```

де

- T1, T2, …, Tn – узагальнені імена типів, які використовуються в класі;
- ClassName – ім’я класу.

Згідно стандарту немає різниці між ключовими словами class та typename.

Примітка. Колись була домовленість, що ключове слово typename говорить про те, що в шаблоні буде використовуватися вбудований тип даних, такий як: int, double, float, char і т. д. А ключове слово class повідомляє компілятору, що в шаблоні функції як параметр будуть використовуватися користувацькі типи даних, тобто класи. Але це правило не стало обов’язковим навіть на рівні стандартів стилю.

Оголошення шаблону класу дає наступні переваги:

•уникається повторюваність програмного коду для різних типів даних. Програмний код (методи, функції) пишеться для деякого узагальненого типу T. Назва узагальненого типу можна давати будь-яку, наприклад, TTT;

•зменшення текстової частини програмного коду, і, як наслідок, підвищення читабельності програм;

•забезпечення зручного механізму передачі аргументів в шаблоні класу з метою їх обробки методами класу.

В прикладі декларується шаблон класу, що містить методи, які виконують наступні операції над деяким числом:

•множення числа на 2;

•ділення одного числа на інше. Для цілих типів виконується ділення націло;

•возведення числа в квадрат (ступень 2).

Декларація шаблону класу має вигляд:

```
// шаблон класу,
template <class T> class MyNumber{
  public:
  // конструктор
  MyNumber(void) { }

  // метод, множення на 2
  void Mult2(T* t);

  // метод, возведення в квадрат
  T MySquare(T x);

  // метод, ділення двох чисел типу T, результат - тип T
  T DivNumbers(T x, T y);
};
```

Для реалізації методів шаблону класу за межами класу потрібно вказувати template <class T1, ...class Tn> перед об’явою функцію.

```
// реалізація методу множення на 2
template <class T> void MyNumber<T>::Mult2(T* t){
  *t = (*t)*2;
}
```

```
// реализація методу, возведення в квадрат
template <class T> T MyNumber<T>::MySquare(T number){
   return (T)(number*number);
}

// метод ділення
template <class T> T MyNumber<T>::DivNumbers(T t1, T t2){
   return (T)(t1/t2);
}
```

Для використання шаблону в інший функції чи методі (зокрема в main()) потрібно об'явити тип використовувано класу обов'язково вказавши, яким типом ми інстанціюємо даний клас.

Примітка. Для параметрів шаблону класу так само як і для шаблонів функцій можна задавати параметри за замовченням.

Використання шаблону класу MyNumber в іншому програмному коді:

```
MyNumber <int> mi; // об'єкт mi класу для типу int
MyNumber <float> mf; // об'єкт mf класу для типу float

int d = 8;
float x = 9.3f;

// множення на 2
mi.Mult2(&d); // d = 16
mf.Mult2(&x); // x = 18.6

// возведення в квадрат
int dd;
dd = mi.MySquare(9); // dd = 81 - ціле число

double z;
z = mf.MySquare(1.1); // z = 1.21000... - дійсне

// ділення чисел
long int t;
float f;

t = mi.DivNumbers(5, 2); // t = 2 - ділення цілих чисел
f = mf.DivNumbers(5, 2); // f = 2.5 - ділення дійсних чисел
```

Загальна форма оголошення шаблону класу, що приймає аргументи

Бувають випадки, коли в шаблоні класу потрібно використовувати деякі аргументи. Ці аргументи можуть використовуватися методами, які описуються в шаблоні класу.

Загальна форма шаблону класу, що містить аргументи, наступна:

template <class T1, class T2, …, class Tn, *type1 var1,* type2 ***var2, ...,*** typeN ***varN*> class ClassName{**

// тіло шаблону класу

// ...

}

де

- T1, T2, …, Tn – назви узагальнених типів даних;
- *type1, type2, …, typeN* – конкретні типи аргументів з іменами *var1, var2, …, varN*;
- *var1, var2, …, varN* – імена аргументів, які використовуються в класі.

Примітка. В декларації шаблонів можна використовувати також інші шаблони та інціалізувати за потреби й аргументи стандартних і нестандартних типів за замовченням:

```
template <class T1, // параметр-тип
          typename T2, // параметр-тип
          int I, // параметр звичайного типу
          T1 DefaultValue, // параметр звичайного типу
          template <class> class T3, // параметр-шаблон
          class C1 = char> // параметр за замовчуванням
```

Примітка 2. У версії стандарту C++11 була додана можливість використання шаблонів зі змінним числом параметрів.

Давайте створимо шаблон класу Стек, де стек - структура даних, в якій зберігаються однотипні елементи даних. В стек можна додавати та видаляти дані. Новий елемент, який додається в стек, встановлюється в вершину стека. Видаляються елемент стека, який знаходиться на його вершині. У шаблоні класу Stack необхідно створити основні методи:

- push - додати елемент в стек;
- pop - видалити елемент з стека (на верхівці);
- top - вивести елемент що на верхівці стеку;
- printStack - виведення стеку на екран.

Отже, реалізуємо ці три методи, в результаті отримаємо найпростіший клас, який реалізує роботу структури стек. Також потрібні конструктори і деструктори.

```
#include <iostream>

template <class T, unsigned N=100>  // довжина стеку за замовченням
class Mystack {
   T a[N]; // будемо зберігати стек у масив
   unsigned num; // поточний розмір доданих даних

  public:
   Mystack(){ num=0; } // конструктор

   T top(){ // показати елемент
       return a[num];
   }

   bool push(T b){  // додати елемент

       if (num+1>=N) return false;
       a[num++]=b;
  return true;
   }


   T pop(){  //видалити елемент
       if (num==0) return 0; // тут краще робити виключення (exception)
  return a[--num];
   }

  void printStack(){ // вивід змісту стеку
  for (int i=0;i<num;++i){
     std::cout<<a[i]<<",";
     std::cout<<std::endl;
  }

  }

};
```

```
// використання стеку
        int main(){

          Mystack<int> s1; // стек цілих чисел
          Mystack<std::string> s2; // стек рядків

          s1.push(1);s1.push(2);
          s1.print();

          s2.push("aaa");s2.push("bbbb");
          s2.print();
        }
```

Шаблони класів працюють точно так же, як і шаблони функцій: компілятор копіює шаблон класу, замінюючи типи параметрів шаблону класу на фактичні (передані) типи даних, а потім компілює цю копію. Якщо у вас є шаблон класу, але ви його не використовуєте, то компілятор не буде його навіть компілювати. Шаблони класів ідеально підходять для реалізації контейнерних класів, так як дуже часто таких класів доводиться працювати з різними типами даних, а шаблони дозволяють це організувати в мінімальній кількості коду. Хоча синтаксис дещо заплутаний, і повідомлення про помилки іноді можуть бути «об'ємними», шаблони класів є однією з важливіших конструкцій мови C++.

Спеціалізація шаблонів класу

Також для шаблонів-класів визначена можливість спеціалізації шаблонів. Прикладами спеціалізації шаблонів в Сі++ є:

- Реалізація функції sort() залежить від численних обмінів елементів значеннями. Якщо операція обміну значеннями є швидкою, як для атомарних типів чи вказівників, то її можна використовувати безпосередньо. Якщо ж вона є повільною, тоді потрібно створити для кожного елемента вказівник і здійснювати обмін значеннями серед вказівників.
- Стандартним прикладом спеціалізації шаблону є vector<bool> — спеціалізація шаблона послідовного контейнера бібліотеки STL, яка використовує однобітне зберігання значень типу bool.

Синтаксис спеціалізації шаблонів наступний:

```
#include <iostream>
using namespace std;
```

```
// шаблон класу - class template

template <class T> class mycontainer {
   T element;
  public:
   mycontainer (T arg) {
              element=arg;
}
   T increase () {
    return ++element;
 }
};

template <>// !!! Увага: спеціалізація шаблону (template specialization)
class mycontainer <char> {
   char element;
  public:
   mycontainer(char arg) {element=arg;}
   char uppercase () {
    if((element>='a')&&(element<='z'))
    element+='A'-'a';
    return element;
   }
};

int main () {
  mycontainer<int> myint (7);
  mycontainer<char> mychar ('j');
  cout << myint.increase() << endl;
  cout << mychar.uppercase() << endl;
}
```

Багатофайлове використання шаблонів

Шаблон не є ні класом, ні функцією - це трафарет, який використовується для створення класів або функцій. Таким чином, шаблони працюють не так, як звичайні функції або класи. У більшості випадків це не є проблемою, але на практиці трапляються різні ситуації. Працюючи зі звичайними класами ми поміщаємо визначення класу в заголовки, а визначення методів цього класу в

окремий файл .cpp з аналогічним ім'ям. Таким чином, фактичне визначення класу компілюється як окремий файл всередині проекту. Однак з шаблонами все відбувається дещо інакше

```
// Array.h — інтерфейс шаблону
#ifndef ARRAY_H
#define ARRAY_H

#include <assert.h> // для assert()

template <class T>
class Array{
private:
   int m_length;
   T *m_data;

public:
   Array()   {
      m_length = 0;
      m_data = nullptr; // c++11 — порожній вказівник
   }

   Array(int length)   {
      m_data = new T[length];
      m_length = length;
   }

   ~Array()   {
      delete[] m_data;
   }

   void Erase()   {
      delete[] m_data;
      // Присвоюємо nullptr для m_data, щоб не отримати незнищений вказівник!
      m_data = nullptr; // c++11
      m_length = 0;
   }
```

```
    T& operator[](int index)    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    // довжина масиву - ціла
    int getLength();
};

#endif

// Array.cpp — реалізація шаблону
#include "Array.h"

template <typename T>
int Array<T>::getLength() { return m_length; }

// ArrayDriver.cpp — файл для тестування шаблону
#include "Array.h"

int main(){
  Array<int> intArray(10);
  Array<double> doubleArray(10);

  for (int count = 0; count < intArray.getLength(); ++count){
          intArray[count] = count;
          doubleArray[count] = count + 0.5;
  }

  for (int count = intArray.getLength()-1; count >= 0; --count)
          std::cout << intArray[count] << "\t" << doubleArray[count] << '\n';

}
```

Вищенаведена програма зкомпілюється, але виникне таку помилку лінкеру: undefined reference to `Array <int> :: getLength ()

Дійсно, для використання шаблону компілятор повинен бачити як визначення шаблону (а не тільки оголошення), так і тип шаблону, що застосовується для створення екземпляра шаблону. Пам'ятаємо, що мова Сі++

компілює файли окремо. Коли заголовки Array.h підключаються в main.cpp, то визначення шаблону класу копіюється в цей файл. У ArrayDriver.cpp компілятор бачить, що нам потрібні два примірника шаблону класу: Array <int> і Array <double>, він створить їх, а потім скомпілює весь цей код як частину файлу ArrayDriver.cpp.

Однак, коли справа дійде до компіляції Array.cpp (окремим файлом), компілятор забуде, що ми використовували Array <int> і Array <double> в ArrayDriver.cpp і не створить екземпляр шаблону функції getLength (), який нам потрібен для виконання програми. Ми отримаємо помилку линкера, так як компілятор не зможе знайти визначення Array <int> :: getLength () або Array <double> :: getLength ().

Цю проблему можна вирішити кількома способами.

Найпростіший варіант - помістити код з Array.cpp в Array.h нижче класу. Таким чином, коли ми будемо підключати Array.h, весь код шаблону класу (повне оголошення і визначення як класу, так і його методів) буде знаходитися в одному місці. Плюс цього способу - простота. Мінус - якщо шаблон класу використовується в багатьох місцях, то ми отримаємо багато локальних копій шаблону класу, що збільшить час компіляції і лінковки файлів (лінкер повинен буде видалити дублювання визначень класу і методів, щоб виконуваний файл не був «занадто роздутим»). Рекомендується використовувати це рішення до тих пір, поки час компіляції або лінковки не є проблемою. Якщо ви вважаєте, що розміщення коду з Array.cpp в Array.h зробить Array.h занадто великим, то альтернативою буде перейменування Array.cpp в Array.inl (.inl від англ. «Inline» = «вбудований»), а потім підключення Array.inl з нижньої частини файлу Array.h. Це дасть той же результат, що і розміщення всього коду в заголовки, але таким чином код вийде трохи чистіше. Є ще рішення - підключення файлів .cpp, але цей варіант не рекомендується використовувати через нестандартного застосування директиви #include.

Найбільш розповсюджений альтернативний варіант - використовувати підхід трьох файлів: визначення шаблону класу зберігається в заголовку, а визначення методів шаблону класу зберігаються в окремому файлі .cpp. Потім додається третій файл, який містить всі необхідні нам екземпляри шаблону класу.

Наприклад, файл ArrayTemplates.cpp:

```
// Ми гарантуємо, що компілятор побачить повне визначення шаблона класу Array
#include "Array.h"
#include "Array.cpp" // ми трохи порушуємо правила включень C++, але лише в одному місці
```

```
// Тут ми за допомогою #include підключаємо всі .h и .cpp з визначеннями шаблонів що нам потрібні
 template class Array<int>; // створюємо екземпляр шаблону класу Array<int>
template class Array<double>; // створюємо екземпляр шаблону класу Array<double>

// створюємо явно інші екземпляри шаблонів класу, які потрібні
```

Після цього компілюємо разом проект, наприклад:
g++ Array.cpp ArrayDriver.cpp ArrayTemplates.cpp -std=c++11

Частина ArrayTemplates змусить компілятор явно створити зазначені екземпляри шаблону класу. У прикладі, наведеному вище, компілятор створить Array <int> і Array <double> всередині ArrayTemplates.cpp. Оскільки ArrayTemplates.cpp знаходиться всередині нашого проекту, то він скомпілюється і вдало зв'яжеться з іншими файлами (пройде лінкінг). Цей метод більш ефективний, але вимагає створення / підтримки третього файлу (ArrayTemplates .cpp) для кожної з програм (проектів) окремо.

Наслідування класів шаблонів

Іноді потрібно використовувати наслідування для шаблонів класів, які використовуються в програмі.

Наприклад, нехай ми маємо шаблон класу Point2D, який має член x y, та методи для встановлення та виведення цих членів:

```
template<class T> class Point2D{
  T x;
  T y;
T getX();
T getY();
void setX(T);
void setY(T);
};
```

Тобто тепер можна визначити об’єкти Point2D визначеного типу, наприклад, Point2D<int>.

Зауважимо, что Point2D - Це не клас шаблону, а шаблон класу. Тобто це шаблон, з якого можна створювати класи. Point2D <int> - це такий клас (це не

об'єкт, але, звичайно, ви можете створити об'єкт з цього класу так само, як ви можете створити об'єкти з будь-якого іншого класу). Іншим таким класом буде Point2D <char>. Зверніть увагу, що це абсолютно різні класи, які не мають нічого спільного, крім того, що вони були створені з одного і того ж шаблону класу.

Нехай потрібно створити клас Point3D, який успадковує клас Point2D. Оскільки Point3D сам по собі не є шаблоном, то не можна ввести Point3D <int>. Оскільки Point2D не є класом, то безпосередньо з нього не можна вивести клас Point3D. Тобто при означенні Point3D необхідно вказати, з якого класу буде відбуватися спадкування. Це вірно незалежно від того, створюються ці класи з шаблону чи ні. Два об'єкти одного класу просто не можуть мати різні ієрархії успадкування.

Але можна наслідувати клас від інстанційованого класу Point2D (або декількох з них). Наприклад, оскільки Point2D <int> це клас, то можна вивести з нього Point3D:

```
class Point3D:  public Point2D<int>{
 T z;
 // ...
};
```

Оскільки Point2D<int> та Point2D<char> - це різні класи, ви навіть можете отримати похідні від обох одночасно (проте при доступі до їх членам вам доведеться мати справу з неоднозначними):

```
class PointX:
  public Point2D<int>,
  public Point2D<char>
{
  // ...
};
```

Тепер у мене є клас Point3D, який успадковує клас Point2D. Оскільки Point3D сам по собі не є шаблоном, не можна ввести Point3D <int>.

Для того, щоб створити клас наслідник шаблоном, потрібно зробити наступне:

```
template<typename T> class Point3D:
  public Point2D<T>{
  // ...
};
```

Тепер у нас є шаблон Point3D, з якого можна отримати клас Point3D <int>, похідний від Point2D <int>, і інший клас Point3D <char>, похідний від Point2D

<char>. Можливо, було б непогано мати один тип Point3D, щоб ви могли передавати всі види Point3D однієї і тій самій функції (яка сама по собі не повинна знати тип області). Оскільки класи Point3D <T>, створені шляхом створення екземпляру шаблону Point3D, формально незалежні один від одного, той варіант, що був зроблений не працює таким чином. Однак тут можна використовувати множинне наслідування:

```
class Point3D // не є наслідником Point2D
{
 // Незалежний інтерфейс
};

template<typename T> class SpecificPoint3D:
 public Point3D,
 public Point2D<T>
{
 // Залежний інтерфейс
};

void foo(Point3D&); //Функція що працює з загальним типом  Point3D

int main(){
 SpecificPoint3D <int> int_pt;
 foo(int_pt);

 SpecificPoint3D <char> char_pt;
 foo(char_pt);
}
```

Якщо важливо, що універсальний Point3D є похідним від універсального Point2D, можна зробити той самий трюк і з Point2D:

```
class  Point2D{
 // загальний інтерфейс Point2D
};

class Point3D:
 public virtual Point2D // virtual тому що "diamond inheritance"
{
 // інтерфейс для  Point3D
};
```

```
template<typename T> class SpecificPoint2D :
 public virtual  Point2D
{
 // реалізація Point2D для типу T
};

template<typename T> class SpecificPoint3D:
 public  Point3D,  // можливо це також буде virtual, для подальшого наслідування
 public SpecificPoint2D <T> // тут не потрібно віртуальне наслідування
{
 // реалізація Point3D для типу T
};
```

Шаблони як члени класів

Окремі проблеми в реалізаціях виникають і з членами-шаблонами. Якщо шаблон, який є членом класу, який у свою чергу є параметром шаблону, використовується в цьому шаблоні і не допускає виведення параметрів, то необхідно використовувати кваліфікатор template :

```
 class A
 {
 / * ... * /
 public:
  template <class T> T & ConvertTo ();
  template <class T> void ConvertFrom (const T & data);
 };
 template <class T>   void f (T Container)
 {
  int i1 = Container.template ConvertTo <int> () + 1;
  Container.ConvertFrom (i1); // кваліфікатор не потрібен
 }
```

Шаблон змінної

Стандарт C++14 додав можливість оголошувати змінні параметризовані за типом даних.

Наприклад, число pi можна визначити як шаблон змінної без визначення конкретного типу:

```
template<typename T> constexpr T pi{3.1415926535897932385};
```

Слід зазначити, що подібно іншим шаблонам, шаблони змінних не можна оголошувати всередині функцій або в блоках коду.