Лекція 11

Шаблонні функції і класи

У цій лекції...

11.1 Основні терміни

Узагальнена функція визначає універсальну сукупність операцій, застосовних до різних типів даних. Тип даних, з якими працює функція, передається як параметр. Узагальнена функція оголошується за допомогою ключового слова **`template`**. Визначення шаблонної функції виглядає в такий спосіб.

```
template <typename T> тип_значення_що_повертається ім'я_функції(список_параметрів)
{
 // Тіло функції
}
```

Тут параметр *T* задає тип даних, з яким працює функція. Цей параметр можна використовувати й усередині функції, однак при створенні конкретної версії узагальненої функції компілятор автоматично підставить замість нього фактичний тип. За правилами узагальнений тип задається за допомогою ключового слова **`typename`**, хоча замість нього можна застосовувати традиційне ключове слово **`class`**.

У наступному прикладі демонструється узагальнена функція, що знаходить найбільший серед двох об'єктів.

```
// Приклад шаблонної функції.
template <class T>
inline T const& max(T const& a, T const& b)
{
   return a< b ? b : a;
}
```

Із шаблонами зв'язано декілька понять. По-перше, узагальнена функція (тобто функція, оголошена за допомогою ключового слова **`template`**) називається також *шаблоном функції*, або *шаблонною функцією* (template function). Ці терміни є синонімами. Конкретна версія узагальненої функції, створювана компілятором, називається *спеціалізацією* (specialization) чи *згенерованою функцією* (generated function). Процес генерації конкретної функції називається *конкретизацією* (instantiation). Іншими словами, згенерована функція є конкретним екземпляром узагальненої функції. Тип `T`, що указується в кутових

дужках, називається *параметром шаблону* (або *шаблонним параметром*), а тип, що указується в списку параметрів (наприклад, `int`) — *параметром виклику*.

> *При утворенні шаблонної функції компілятор може автоматично генерувати стільки її різних варіантів, скільки існує способів виклику цієї функції в програмі.*

11.2. Вивід аргументів

Під час виклику функції (наприклад, `max`) параметри шаблону визначаються аргументами, що передаються в функцію. Якщо в якості параметрів типу `T const&` передається два значення `int`, компілятор робить висновок, що замість `T` слід підставити `int`.

> *Автоматичне перетворення типів в шаблонних функціях не дозволяється. Відповідність типів параметрів і аргументів повинна бути точною.*

```
// Приклад вірного і помилкового вживання параметрів

template <typename T>
void max(T& a, T& b)
...
max(4,5);   // Вірно: T == int для обох аргументів
max(4,5.5); // Помилка: перший T == int, другий T == double
```

Існує три способи виправлення цієї помилки.

1. Привести обидва аргументи до одного типу:
   ```
   max(static_cast<double>(4), 5.5);
   ```
2. Указати тип `T` явно
   ```
   max<double>(4, 5.5);
   ```
3. Задати різні типи параметрів шаблонів.

11.3. Функція з кількома узагальненими типами

Використовуючи список, елементи якого розділені комами, можна визначити кілька узагальнених типів даних в операторі **`template`**. Наприклад, у наступній програмі створюється шаблонна функція, що має два узагальнених типи.

```
template <typename T1, typename T2>
inline T1 max (T1 const& a, T2 const& b)
{
   return a < b? b : a;
}
```

> *Кількість параметрів шаблону необмежена, але в шаблонах функцій (на відміну від шаблонів класів) не можна використовувати аргументи шаблону за умовчанням.*

Можливість задавати декілька параметрів шаблону дозволяє розв'язати проблему виводу аргументів, але породжує нові. Проблема полягає в тому, що ми повинні оголосити тип значення, що повертається. Якщо для цього використати один із двох типів параметрів `T1` або `T2`, аргумент для іншого параметру повинен конвертуватися в цей же тип, незалежно від волі програміста. В C++ немає способу задати правило вибору "найбільш потужного типу". Отже, залежно від порядка слідування аргументів від час виклику можна отримати як найбільше число серед пари 4 і 5.5 і `double`, і `int` (тобто, 5.5 або 5). Крім того, при конвертуванні типу другого параметру в тип значення, що повертається, утворюється новий локальний тимчасовий об'єкт, що унеможливлює повертання результату за посиланням. Отже, в нашому прикладі, типом значення, що повертається, повинен бути `T1`, а не `T1 const&`.

Оскільки типи параметрів виклику конструюються із параметрів шаблону, вони зазвичай пов'язані один з одним. Ця концепція називається *виводом аргументів шаблону функції* і забезпечує можливість викликати шаблонну функцію так само, як і звичайну.

В тих випадках, коли цей зв'язок відсутній, аргумент шаблона під час виклику необхідно задаваи явно. Наприклад, можна ввести третій тип параметра шаблона, який задає тип значення, що повертає функція.

```
template <typename T1, typename T2, typename RT>
inline RT max(T1 const& a, T2 const& b);
```

Але механізм виводу аргументів шаблона не розповсюджується на типи значень, що повертаються, а серед типів параметрів виклику функції RT відсутній. Отже, для його визначення необхідно явно задати список аргументів шаблону.

```
template <typename T1, typename T2, typename RT>
inline RT max(T1 const& a, T2 const& b);
...
max<int,double,double>(4,5.5); // Вірно, але занадто обтяжливо
```

Досі розглядалися два варіанти: коли всі аргументи шаблона функції задавалися явно, або явно не задавався жоден з них. Але існує ще одна можливість: явно задається лише перший аргумент, а решта — виводиться.

Слід явно задавати всі типи аргументів, які не можна визначити неявно.

Отже, якщо в нашому прикладі змінити порядок слідування параметрів шаблону, то під час виклику знадобиться указати лише тип значення, що повертається.

```
template <typename RT, typename T1, typename T2>
inline RT max(T1 const& a, T2 const& b);
...
max<double>(4,5.5); // Вірно, повертається double
```

В даному випадку RT задається явно, а типи T1 і T2 виводяться із аргументів виклику як int і double.

Жодна з наведених версій не дає суттєвих переваг, отже, краще зупинитися на найпростішому варіанті — версії max() з одним параметром.

11.4. Явна спеціалізація узагальненої функції

Незважаючи на те что узагальнена функція перевантажує сама себе, ее можна перевантажити явно. Цей процес називається *явною спеціалізацією* (explicit specialization). Перевантажена функція заміщає (чи "маскує") узагальнену функцію, зв'язану з даною конкретною версією. Розглянемо модифіковану версію програми, призначеної для перестановки двох змінних.

```
// Перевантаження шаблонної функції.
#include <iostream>
using namespace std;

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

int max(int &a, int &b)
{
   return a < b ? b : a;
}

int main()
{
   int i=10, j=20;
   double x=10.5, y=25.5;
   char a='a', b='z';

   cout << "i ? j: " << max(i,j) << '\n';
   cout << "x ? y: " << max(x,y) << '\n';
   cout << "a ? z: " << max(a,b) << '\n';
```

```
    return 0;
}
```

Ця програма виводить на екран нступні рядки.

```
i ? j: 20
x ? y: 25.5
a ? z: z
```

Існує альтернативна синтаксична конструкція, призначена для позначення явної спеціалізації функції. Цей метод використовує ключове слово **`template`**. Наприклад, перевантажену функцію **`max()`** з попереднього прикладу можна переписати в такий спосіб.

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

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

template<> int max<int>(int &a, int &b)
{
   return a < b ? b : a;
}

int main()
{
   int i=10, j=20;
   double x=10.5, y=25.5;
   char a='a', b='z';

   cout << "i ? j: " << max(i,j) << '\n';
   cout << "x ? y: " << max(x,y) << '\n';
   cout << "a ? z: " << max(a,b) << '\n';
   return 0;
}
```

Як бачимо, новий спосіб визначення спеціалізації містить конструкцію **`template<>`**. Тип даних, для якого призначена спеціалізація, вказується усередині кутових дужок після імені функції. Для спеціалізації будь-якого іншого типу узагальненої функції використовується така ж синтаксична конструкція. В даний час обидва способи визначення спеціалізації еквівалентні.

11.5. Явна конкретизація узагальненої функції

Конкретизація шаблонів — це процесс, під час якого на основі узагальненого визначення шаблонів генеруються типи і функції. *Спеціалізація* — це конкретний екземпляр шаблона. Коли компілятор зустрічає використання спеціалізації шаблону, від утворює його, підставляя замість параметрів шаблону необхідні аргументи. Ці дії виконуються автоматично і не вимагають жодних указівок в коді або визначенні шаблону. Такий процес називають *неявною, або автоматичною конкретизацією*.

Точка конкретизації утворюється в тому випадку, коли деяка конструкція вихідного коду посилається на спеціалізацію шаблона таким чином, що для цієї спеціалізації потрібно виконати конкретизацію шаблона. Точка конкретизації — це місце кода, в яке можна вставити шаблон с підставленими аргументами.

Існує три способи явної конкретизації.

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

template<typename T>
T max(T &a, T &b)
{
```

```
   return a < b ? b : a;
}

// Перший спосіб
template char max(char &a, char &b);

// Другий спосіб
template double max<>(double &a, double &b);

// Третій спосіб
template float max<float>(float &a, float &b);



int main()
{
   int i=10, j=20;
   double x=10.5, y=25.5;
   char a='a', b='z';

   cout << "i ? j: " << max(i,j) << '\n';
   cout << "x ? y: " << max(x,y) << '\n';
   cout << "a ? z: " << max(a,b) << '\n';
   return 0;
}
```

В програмі повинно бути не більше однієї явної конкретизації для визначеної спеціалізації шаблону.

Розглянемо ситуацію, в якій реалізується бібліотека. Нехай перша версія шаблону функції виглядає так.

```
// Файл max.hpp
template <typename T>
T max(T const& x, T const& y)
{
   return a < b ? b : a;
}
```

Користувач бібліотеки може включити наведений вище заголовочний файл і явно конкретизувати шаблон, що в ньому міститься.

```
// Код користувача
#include "max.hpp"
template int max(int, int);
```

11.6. Перевантаження шаблонної функції

Для того щоб перевантажити специфікацію узагальненої функції, достатньо створити ще одну версію шаблона, що відрізняється від інших своїм списком параметрів.

```
// Перевантаження шаблонної функції.
#include <iostream>
using namespace std;

// Перша версія шаблонної функції f().
template <class T> void f(T a)
{
 cout << "Inside f(T a)\n";
}

// Друга версія шаблонної функції f().
template <class T, class Y> void f(T a, Y b)
```

```
{
 cout << "Inside f(T a, Y b)\n";
}

int main()
{
 f(10);     // Виклик функції f(T).
 f(10, 20); // Виклик функції f(T, Y).

 return 0;
}
```

11.7. Використання стандартних параметрів шаблонних функцій

При визначенні шаблонної функції можна змішувати стандартні й узагальнені параметри. У цьому випадку стандартні параметри нічим не відрізняються від параметрів будь-яких інших функцій. Розглянемо приклад.

```
// Застосування стандартних параметрів у шаблонній функції.
#include <iostream>
using namespace std;
const int TABWIDTH = 8;

// Виводить на екран дані в позиції tab.
template<class T> void tabOut(T data, int tab)
{
   for(; tab; tab--)
   for(int i=0; i<TABWIDTH; i++) cout << ' ';
   cout << data << "\n";
}

int main()
{
   tabOut("Перевірка", 0);
   tabOut(100, 1);
   tabOut('T', 2);
   tabOut(10/3, 3);
   return 0;
}
```

Програма виводить на екран наступні повідомлення.

```
Перевірка
        100
              T
                      3
```

11.8. Обмеження на узагальнені функції

Узагальнені функції нагадують перевантажені, але на них накладаються ще більш жорсткі обмеження. При перевантаженні усередині тіла кожної функції можна виконувати різні операції. У той же час узагальнена функція повинна виконувати ту саму універсальну операцію для усіх версій, розрізнятися можуть лише типи даних. Розглянемо перевантажену функцію на наступному прикладі. Ці функції *не можна* заміняти узагальненими, оскільки вони мають різне призначення.

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

void myfunc(int i)
{   cout << "Значення = " << i << "\n"; }

void myfunc(double d)
{  double intpart, fracpart;
   fracpart = modf(d, &intpart);
```

```
   cout << "Дробова частина = " << fracpart << endl;
   cout << "Ціла частина    = " << intpart;
}

int main()
{
   myfunc(1);
   myfunc(12.2);
   return 0;
}
```

11.9. Узагальнені класи

Крім узагальнених функцій можна визначити узагальнені класи. При цьому створюється клас, у якому визначені всі алгоритми, але фактичний тип даних задається як параметр при створенні об'єкта.

Узагальнені класи виявляються корисними, якщо логіка класу не залежить від типу даних. Наприклад, до черг, що складаються з цілих чисел або символів, можна застосовувати той самий алгоритм.

Оголошення узагальненого класу має наступний вид.

```
template <class Ттип> class ім'я_класу
{
   .
   .
   .
}
```

Тут параметр *Ттип* задає тип даних, що уточнюється при створенні екземпляра класу. При необхідності можна визначити несколько узагальнених типів, використовуючи список імен, розділених комами.

Конкретний екземпляр узагальненого класу створюється за допомогою наступної синтаксичної конструкції.

```
ім'я_класу <тип> ім'я_об'єкта;
```

Тут параметр *тип* задає тип даних, якими оперує клас.

> *Функції — члени узагальненого класу автоматично стають узагальненими. Для їхнього оголошення не обов'язково використовувати ключове слово* **`template`**.

Наступна програма використовує узагальнений клас **`stack`**. Тепер его можна застосовувати для збереження об'єктів будь-якого типу. У даному прикладі створюються стеки символів і дійсних чисел.

```
// Демонстрація узагальненого стека.
#include <iostream>
using namespace std;

const int SIZE = 10;

// Створюємо узагальнений клас stack.
template <class StackType> class stack
{
  StackType stck[SIZE]; // Містить елементи стека.
  int tos; // Індекс вершини стека.

 public:
  stack() { tos = 0; } // Ініціалізує стек.
  void push(StackType ob); // Заштовхує об'єкт у стек.
  StackType pop(); // Виштовхує об'єкт зі стека.
 };

  // Заштовхуємо об'єкт у стек.
  template <class StackType>
```

```
  void stack<StackType>::push(StackType ob)
  {
    if(tos==SIZE) {
    cout << "Стек повний.\n";
    return;
    }
    stck[tos] = ob;
    tos++;
  }

// Виштовхуємо об'єкт зі стека.
template <class StackType> StackType stack<StackType>::pop()
{
  if(tos==0) {
  cout << "Стек порожній.\n";
  return 0; // Якщо стік порожній, повертається константа null.
  }
  tos--;
  return stck[tos];
}

int main()
{
  // Демонстрація стеку символів.
  stack<char> s1, s2; // Створюємо два стеки символів.
  int i;
  s1.push('a');
  s2.push('x');
  s1.push('b');
  s2.push('y');
  s1.push('c');
  s2.push('z');

  for(i=0; i<3; i++) cout << "Виштовхуємо s1: " << s1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Виштовхуємо s2: " << s2.pop() << "\n";

  // Демонстрація стека дійсних чисел
  stack<double> ds1, ds2; // Створюємо дві стеки
                          // дійсних чисел.
  ds1.push(1.1);
  ds2.push(2.2);
  ds1.push(3.3);
  ds2.push(4.4);
  ds1.push(5.5);
  ds2.push(6.6);

  for(i=0; i<3; i++) cout << "Виштовхуємо ds1: " << ds1.pop() << "\n";
  for(i=0; i<3; i++) cout << "Виштовхуємо ds2: " << ds2.pop() << "\n";

  return 0;
}
```

Як бачимо, оголошення узагальненого класу мало відрізняється від оголошення узагальненої функції. Фактичний тип даних, розташовуваний у стеці, в оголошенні класу заміняється узагальненим параметром і уточнюється лише при створенні конкретного об'єкта. При оголошенні конкретного об'єкта класу **stack** компілятор автоматично генерує усі функції і змінні, необхідні для обробки фактичних даних. У попередньому прикладі з'являються по двох стека різних типів — цілих чисел і дійсних чисел. Зверніть особливу увагу на наступні оголошення.

```
stack<char> s1, s2; // Створюємо дві стеки символів.
stack<double> ds1, ds2; // Створюємо дві стеки дійсних чисел.
```

Як бачимо, необхідний тип даних задається в кутових дужках. Змінюючи цей тип при створенні об'єкта класу **stack**, можна змінювати тип даних, що зберігаються в стеці. Наприклад, використовуючи наступне визначення, можна створити інший стек для збереження покажчиків на символи.

```
stack<char *> chrptrQ;
```

Можна створювати стеки, що зберігають об'єкти, тип яких визначений користувачем. Припустимо, що для збереження інформації використовується наступна структура.

```
struct addr {
  char name[40];
  char street[40];
  char city[30];
  char state[3];
  char zip[12];
};
```

У цьому випадку клас **stack** породжує стек, у якому зберігаються об'єкти класу **addr**. Для цього використовується наступне оголошення.

```
stack<addr> obj;
```

Клас **stack** демонструє, что узагальнені функції і класи є могутнім засобом, що полегшує програмування. З його допомогою програміст може визначати загальну форму об'єкта, у якому зберігаються дані довільного типу, і не піклуватися про окремі реалізації класів і функцій, призначених для різних типів. Компілятор автоматично створює конкретні версії класу.

11.10. Приклад використання двох узагальнених типів даних

Шаблонний клас може мати декілька шаблонних типів. Для цього їх достатньо перелічити в списку шаблонних параметрів в оголошенні **template**. Наприклад, наступна програма створює клас, що використовує два узагальнених типи.

```
/*  Приклад класу, що використовує два узагальнених типи. */

#include <iostream>
using namespace std;

template <class Type1, class Type2> class myclass
{
  Type1 i;
  Type2 j;
public:
  myclass(Type1 a, Type2 b) { i = a; j = b; }
  void show() { cout << i << ' ' << j << '\n'; }
};

  int main()
  {
    myclass<int, double> ob1(10, 0.23);
    myclass<char, char *> ob2('X', "Шаблони — могутній механізм.");

    ob1.show(); // Виводимо ціле і дійсне число.
    ob2.show(); // Виводимо символ і покажчик на символ.

    return 0;
}
```

Ця програма виводить наступні результати.

```
10 0.23
X Шаблони — могутній механізм.
```

У програмі з'являються об'єкти двох типів. Об'єкт **ob1** використовує цілі і дійсні числа. Об'єкт **ob2** використовує символ і покажчик на символ. В обох випадках при створенні об'єктів компілятор автоматично генерує відповідні дані й функції.

11.11. Застосування шаблонних класів: узагальнений масив

Щоб проілюструвати практичні вигоди, що надають узагальнені класи, розглянемо спосіб, що досить часто застосовується. Як відомо, оператор "**[]**" можна перевантажити. Це дозволяє створювати власні реалізації масиву, у тому числі "безпечні" масиви, що передбачають перевірку діапазону індексів у ході виконання програми. У мові C++ немає вбудованої перевірки діапазону індексів, тому в ході виконання програми індекс може вийти за припустимі межі, не генеруючи повідомлення про помилку. Однак, якщо створити клас, що містить масив, і перевантажити оператор "**[]**", виходу індексу за припустимі межі можна запобігти.

Комбінуючи перевантажений оператор із шаблонним класом, можна створити узагальнений безпечний масив довільного типу. Цей тип масиву показаний у наступній програмі.

```
// Приклад узагальненого безпечного масиву.
#include <iostream>
#include <cstdlib>
using namespace std;

const int SIZE = 10;

template <class AType> class atype {
  AType a[SIZE];
public:
  atype() {
    register int i;
    for(i=0; i<SIZE; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Перевірка діапазону для об'єкта atype.
 template <class AType> AType &atype<AType>::operator[](int i)
 {
  if(i<0 || i> SIZE-1) {
  cout << "\nЗначення індексу  ";
  cout << i << " виходить за межі припустимого діапазону.\n";
  exit(1);
  }
  return a[i];
 }

 int main()
 {
  atype<int> intob;       // Цілочисельний масив.
  atype<double> doubleob; // Масив дійсних чисел.

  int i;

  cout << "Цілочисельний масив: ";
  for(i=0; i<SIZE; i++) intob[i] = i;
  for(i=0; i<SIZE; i++) cout << intob[i] << " ";
  cout << '\n';

  cout << "Масив дійсних чисел: ";
  for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;
  for(i=0; i<SIZE; i++) cout << doubleob[i] << " ";
  cout << '\n';

  intob[12] = 100; // Генерує повідомлення про помилку.
```

```
  return 0;
}
```

Ця програма реалізує узагальнений тип масиву і демонструє його застосування на прикладі масивів цілих і дійсних чисел. Спробуйте створити масиви інших типів.

> *Узагальнені класи дозволяють створювати один варіант коду, налагоджувати його, а потім застосовувати до будь-якого типу даних, не передбачаючи для кожного типу свій варіант.*

11.12. Застосування стандартних типів в узагальнених класах

У специфікації шаблона узагальненого класу можна використовувати стандартні типи. Інакше кажучи, як шаблонні параметри можна застосовувати стандартні аргументи, наприклад, цілочисельні значення або покажчики. Синтаксис такого оголошення не відрізняється від оголошення звичайних параметрів функцій: необхідно лише вказати тип і ім'я аргументу. Розглянемо один з найбільш вдалих способів реалізації безпечних узагальнених масивів, що дозволяє задавати розмір масиву.

```
// Демонстрація стандартних шаблонних параметрів.
#include <iostream>
#include <cstdlib>
using namespace std;

// Тут цілочисельний аргумент size є стандартним.
template <class AType, int size> class atype {
  AType a[size]; // Довжина масиву передається через параметр size.
public:
  atype()
  {
    register int i;
    for(i=0; i<size; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Перевірка діапазону для об'єкта atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
  if(i<0 || i> size-1) {
  cout << "\nЗначення індексу ";
  cout << i << " виходить за межі припустимого діапазону.\n";
  exit(1);
  }
  return a[i];
}



int main()
{
  atype<int, 10> intob; // Цілочисельний масив з 10 елементів.
  atype<double, 15> doubleob; // Масив дійсних чисел,
                              // складає з 16 елементів.
  int i;

  cout << "Цілочисельний масив: ";
  for(i=0; i<10; i++) intob[i] = i;
  for(i=0; i<10; i++) cout << intob[i] << " ";
  cout << '\n';

  cout << "Масив дійсних чисел: ";
```

```
  for(i=0; i<15; i++) doubleob[i] = (double) i/3;
  for(i=0; i<15; i++) cout << doubleob[i] << " ";
  cout << '\n';

  intob[12] = 100; // Генерує повідомлення про помилку.

  return 0;
}
```

Уважно розглянемо специфікацію шаблона **atype**. Зверніть увагу на те, що параметр **size** оголошений як цілочисельний. Потім цей параметр використовується в класі **atype** для оголошення масиву **a**.

> *Хоча параметр **size** застосовується у вихідному коді как звичайна змінна, его значення відоме уже на етапі компіляції. Це дозволяє задавати розмір масиву.*

Крім того, параметр **size** використовується при перевірці діапазону індексу в операторній функції **operator[]()**. Зверніть увагу на спосіб, яким створюються масиви цілих і дійсних чисел. Другий параметр задає розмір кожного масиву.

> *Як стандартні параметри можна застосовувати лише цілі числа, покажчики і посилання. Інші типи, наприклад **float**, не допускаються. Аргументи, що передаються стандартним параметрам, повинні містити або цілочисельну константу, або покажчик, або посилання на глобальну функцію чи функцію-об'єкт.*

Таким чином, стандартні параметри можна розглядати як константи, оскільки их значення не можна змінювати. Наприклад, усередині функції **operator[]()** наступний оператор не допускається.

```
size = 10; // Помилка
```

Оскільки стандартні параметри вважаються константами, з їхньою допомогою можна задавати розмір масиву, що досить важливо в практичних застосуваннях.

Як показує приклад безпечного узагальненого масиву, стандартні параметри значно розширюють можливості узагальнених класів. Хоча значення стандартних аргументів повинні бути відомі вже на етапі компіляції, це обмеження не занадто обтяжне в порівнянні з потужністю, наданої ними.

11.13. Застосування аргументів за замовчуванням у шаблонних класах

Шаблонний клас може мати аргумент узагальненого типу, значення якого задано за замовчуванням. Наприклад, такий.

```
template <class X=int> class myclass { //...
```

Якщо при конкретизації об'єкта типу **myclass** не буде зазначений жодний тип, використовується тип **int**.

Стандартні аргументи також можуть мати значення за замовчуванням. Вони використовуються при конкретизації об'єкта, якщо не задані явні значення аргументів. Синтаксична конструкція, застосовувана для цих параметрів, не відрізняється від оголошення функцій, що мають аргументи за замовчуванням.

Розглянемо ще один варіант безпечного масиву, що передбачає аргументи за замовчуванням як для типу даних, так і для розміру масиву.

```
// Демонстрація шаблонних аргументів за замовчуванням.
#include <iostream>
#include <cstdlib>
using namespace std;

// Параметр типу AType за замовчуванням дорівнює int,
// а змінна size за замовчуванням дорівнює 10.
template <class AType=int, int size=10> class atype {
  AType a[size]; // Розмір масиву передається аргументом size.
public:
  atype()
  {
```

```
    register int i;
    for(i=0; i<size; i++) a[i] = i;
  }
  AType &operator[](int i);
};

// Перевірка діапазону для об'єкта atype.
template <class AType, int size>
  AType &atype<AType, size>::operator[](int i)
{
  if(i<0 || i> size-1) {
  cout << "\nЗначення индекса ";
  cout << i << " виходить за межі припустимого діапазону.\n";
  exit(1);
  }
  return a[i];
}

int main()
{
  atype<int, 100> intarray;  // Цілочисельний масив з 100 елементів.
  atype<double> doublearray; // Масив дійсних чисел,
                             // розмір заданий за замовчуванням.
  atype<> defarray; // За замовчуванням з'являється цілочисельний
                    // масив, що складається з 10 елементів.
  int i;

  cout << "Цілочисельний масив: ";
  for(i=0; i<100; i++) intarray[i] = i;
  for(i=0; i<100; i++) cout << intarray[i] << " ";
  cout << '\n';

  cout << "Масив дійсних чисел: ";
  for(i=0; i<10; i++) doublearray[i] = (double) i/3;
  for(i=0; i<10; i++) cout << doublearray[i] << " ";
  cout << '\n';

  cout << "Масив за замовчуванням: ";
  for(i=0; i<10; i++) defarray[i] = i;
  for(i=0; i<10; i++) cout << defarray[i] << " ";
  cout << '\n';

  return 0;
}
```

Зверніть увагу на рядок

```
template <class AType=int, int size=10> class atype {
```

Тут тип **AType** за замовчуванням є типом **int**, а змінна **size** дорівнює 10. Як демонструє програма, об'єкти класу **atype** можна створити трьома способами.

- Явно задаючи тип і розмір масиву.
- Явно задаючи тип масиву, використовуючи розмір за замовчуванням.
- Використовуючи тип і розмір масиву, установлені за замовчуванням.

Застосування аргументів за замовчуванням (особливо типів) підвищує універсальність шаблонових класів. Якщо деякий тип використовується частіше інших, його можна задати за замовчуванням, надавши користувачу можливість самому конкретизувати інші типи.

11.4. Явні спеціалізації класів

Як і при використанні шаблонових функцій, можна створити явну спеціалізацію узагальненого класу. Для цього, як і колись, застосовується конструкція **template<>**.

```
// Демонстрація спеціалізації класу.
#include <iostream>
 using namespace std;

 template <class T> class myclass {
  T x;
 public:
  myclass(T a) {
  cout << "Усередині узагальненого класу myclass\n";
  x = a;
  }
  T getx() { return x; }
 };

 // Явна спеціалізація для типу int.
 template <> class myclass<int> {
  int x;
  public:
    myclass(int a) {
    cout << "Усередині спеціалізації myclass<int>\n";
    x = a * a;
    }

  int getx() { return x; }
 };

 int main()
 {
  myclass<double> d(10.1);
  cout << "double: " << d.getx() << "\n\n";

  myclass<int> i(5);
  cout << "int: " << i.getx() << "\n";

  return 0;
 }
```

Програма виводить на екран наступні результати.

```
Усередині узагальненого класу myclass
 double: 10.1
```

Зверніть увагу на наступну рядок програми.

```
template <> class myclass<int> {
```

Вона повідомляє компілятору, що створюється явна цілочисельна спеціалізація узагальненого класу **myclass**. Така ж синтаксична конструкція застосовується для будь-якої іншої спеціалізації класу.

Явна спеціалізація класу розширює можливості узагальнених класів, оскільки вона дозволяє обробляти одну чи дві особливі ситуації, надаючи компілятору автоматично генерувати інші спеціалізації. Якщо програмі буде потрібно занадто багато спеціализацій, возможно, варто узагалі відмовитися від узагальнених класів.

11.15. Часткова спеціалізація

Спеціалізація шаблонів може бути частковою.

> *Можна визначити реалізацію шаблонів для певних фіксованих типів, залишаючи решту параметрів шаблону невизначеними.*

```
template<typename T1, typename T2>
class MyClass {
```

```
...
};
```

Часткову спеціалізацію можна здійснити кількома способами.

```
// Часткова спеціалізація: обидва параметри шаблона мають однаковий тип
template <typename T>
class MyClass<T,T> {
...
};

// Часткова спеціалізація: тип другого параметру — int.
template <typename T>
class MyClass<T,int> {
...
};

// Часткова спеціалізація: обидва параметри — вказівники.
template <typename T1, typename T2>
class MyClass<T1*,T2*> {
...
};
```

Покажемо, як застосувати наведені шаблони в різних оголошеннях.

```
MyClass<int, float> mif;   // Використовується MyClass<T1, T2>
MyClass<float, float> mff; // Використовується MyClass<T, T>
MyClass<float, int> mfi;   // Використовується MyClass<T1, int>
MyClass<int*, int*> mp;    // Використовується MyClass<T1*, T2*>
```

Якщо для оголошення однаково добре підходять кілька часткових спеціалізацій, виникає неоднозначність. яка не розв'язується компілятором.

```
MyClass<int, int> m;       // MyClass<T, T> або MyClass<T, int>
MyClass<float, float> mff; // MyClass<T, T> або MyClass<T1*, T2*>
```

Щоб уникнути неоднозначності в другому варіанті, можна використати додаткову часткову спеціалізацію для вказівників одного і того ж типу.

```
template <typename T>
class MyClass<T*,T*> {
...
};
```

Існують кілька обмежень на оголошення списків параметрів і аргументів часткової спеціалізації.

1. Аргументи часткової спеціалізації повинні відповідати параметрам первинних шаблонів.
2. Список параметрів часткової спеціалізації не може мати аргументів за замовчуванням. Замість них використовуються аргументи за замовчуванням первинного шаблону.
3. Аргументи часткової спеціалізації, які не є типами, повинні бути або незалежними значеннями, або простими змінними. Вони не можуть бути складними виразами (наприклад, `2*N`).
4. Список аргументів шаблону часткової спеціалізації не повинен бути ідентичним списку параметрів первинного шаблону.

```
template <typename T, int I = 3>
class S;           // Первинний шаблон

template <typename T>
class S<int, T>;   // ПОМИЛКА: невідповідність параметрів
```

```
template <typename T = int>
class S<T, 10>;    // ПОМИЛКА: аргументи за замовчуванням не дозволені

template<int I>
class S<int, I*2>  // ПОМИЛКА: вирази не допускаються

template<typename U, typename K>
class S<U, K>      // ПОМИЛКА: шаблон, що є ідентичним первинному шаблону
```

Часткова спеціалізація шаблону може мати більшу кількість параметрів, ніж первинний шаблон.

11.16. Ключові слова typename і export

Порівняно нещодавно в мову C++ були включені ключові слова, зв'язані із шаблонами: **`typename`** і **`export`**, що мають особливе значення для програмування. Стисло розглянемо кожне з них.

Ключове слово **`typename`** використовується в двох ситуаціях. По-перше, як указувалося раніше, воно може заміняти ключове слово **`class`** в оголошенні шаблона. Наприклад, шаблонну функцію **`swapargs()`** можна визначити так.

```
template <typename X> void swapargs(X &a, X &b)
{
  X temp;

  temp = a;
  a = b;
  b = temp;
}
```

Тут ключове слово **`typename`** задає узагальнений тип **`X`**. У цьому контексті ключові слова **`class`** і **`typename`** не розрізняються.

По-друге, ключове слово **`typename`** інформує компілятор про те, что деяке ім'я використовується в оголошенні шаблонного класу як ім'я типу, а не об'єкта. Розглянемо приклад.

```
typename X::Name someObject;
```

Тут ім'я **`X::Name`** використовується як ім'я типу.

Ключове слово **`export`** може передувати оголошенню **`template`**. Воно дозволяє використовувати шаблон з іншого файлу, повторюючи лише його оголошення, а не усе визначення.

11.17. Резюме

- Шаблони функцій визначають сімейство функцій для різних аргументів шаблонів.
- При передачі аргументів шаблона відбувається конкретизація шаблонів функцій для даних типів аргументів.
- Параметри шаблонів можна задавати явно.
- Шаблони функцій можна перевантажувати.
- При перевантаженні шаблонів функцій слід обмежувати зміни явним указанням параметрів шаблону.
- Слід пересвідчитись, що всі перевантажені версії шаблонів функцій розташовані до викликів відповідних функцій.
- Шаблон класу — це клас, що реалізований з одним або кількома шаблонними параметрами, які залишаються невизначеними.
- Щоб застосувати шаблон класу, треба використати конкретні типи як аргументи шаблону. Після цього шаблон конкретизується і компілюється для указаних типів.

- Для шаблонних класів конкретизуються лише ті функції-члени, які реально використовуються в програмі.
- Шаблони класів можна спеціалізувати для конкретних типів.
- Шаблони класів допускають часткову спеціалізацію.
- Параметри шаблонного класу можна задавати за замовчуванням. Ці значення можуть використовувати попередні параметри шаблона.

11.18. Контрольні питання

1. Як називається конкретна версія шаблонної функції створювана компілятором?
2. Як називається процес генерації конкретної функції?
3. Скільки варіантів шаблонної функції генерує компілятор?
4. Як явно перевантажити узагальнену функцію (2 варіанти)?
5. Як перевантажити шаблонну функцію?
6. Приклад оголошення узагальненого класу і його об'єкта?
7. Що таке точка конкретизації?
8. Назвіть три способи явної конкретизації.
9. Наведіть приклад оголошення узагальненого класу і його об'єкта.
10. Наведіть приклад запису оголошення функції узагальненого класу і її реалізації.
11. Чи можна створити реалізацію шаблонового класу з користувальницьким типом даних.
12. Наведіть приклад узагальненого безпечного масиву.
13. Наведіть приклад узагальненого безпечного масиву, але з завданням розміру цього масиву.
14. Які типи можна використовувати як стандартні параметри для шаблонного класу або функції.
15. Сформулюйте правила передачі стандартних параметрів шаблонного класу.
16. Наведіть приклад використання аргументів за замовчуванням у шаблонних класах.
17. Наведіть приклад явної спеціалізації класу.
18. Наведіть два випадки застосування typename.
19. Як використовується слово export стосовно до шаблонів.