Лекція 15

Функтори

У цій лекції...

15.1. Базові класи
15.2. Арифметичні операції
15.3. Предикати
15.4. Логічні операції
15.5. Адаптери
15.6. Розподільники

Функторами називаються об'єкти, що інкапсулюють виклик функції, тобто містять перевантажену версію операторної функції `operator()`. Ці об'єкти передаються шаблонним алгоритмам як вказівники на функцію. Оскільки код функтора не викликається, а підставляється безпосередньо в текст програми, ефективність алгоритму підвищується.

15.1. Базові класи

Проаналізуємо програму, у якій застосовується алгоритм `for_each()`. Він реалізований у вигляді шаблонної функції, що має три аргументи — початок і кінець діапазону, а також виклик функції. Робота цього алгоритму полягає в застосуванні викликуваної функції до кожного елемента з заданого діапазону.

От як виглядає оголошення алгоритму `for_each()`.

```
template<class InputIterator, class Function> inline
     Function for_each(InputIterator first, InputIterator last, Function F)
```

У найбільш простому варіанті алгоритм `for_each()` одержує вказівник на функцію `F()`.

Застосування алгоритму for_each()

```
 #include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

void ZeroEven(int n)
{
   if (n%2) n = 0;
   cout << n  << " " ;
}

int main()
{
   const int SIZE = 15 ;
   vector<int> array(SIZE);
   vector<int>::iterator start, end, it;

   for (int i = 0; i < SIZE; i++)  array [i] = i + 1 ;

   start = array.begin();
   end   = array.end();

   cout << "array { " ;
   for(it = start; it != end; it++)
       cout << *it << " ";
   cout << " }" << endl;

   for_each(start, end, ZeroEven) ;
   cout << "\n" ;
   return 0;
}
```

У програмі з'являється цілочисельний вектор `array` і три ітератори довільного доступу — `start`, `end` і `it`. Вектор заповнюється числами від 1 до 15 і виводиться на екрані. Потім на початок і кінець вектора `array` установлюються ітератори `start` і `end` відповідно. Наприкінці програми за допомогою алгоритму

for_each парні елементи вектора при виводі на екран заміняються нулями (елементи самого вектора при цьому не змінюються). Програма виводить на екран наступні повідомлення.

Результат

```
 array = { 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  }
0 2 0 4 0 6 0 8 0 10 0 12 0 14 0
```

У даному випадку алгоритму for_each() передавався вказівник на функцію ZeroOdd(). Покажемо, як створюється і застосовується функтор, що виконує ту ж операцію.

Застосування функтора

```
 #include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

template <typename T>
class ZeroOdd
{
public:
   void operator()(T i){ if (i%2) i = 0; cout << i << " " ;}
};

int main()
{
   const int SIZE = 15 ;
   vector<int> array(SIZE);
   vector<int>::iterator start, end, it;

   ZeroOdd<int> FunctorZeroOdd;

   for (int i = 0; i < SIZE; i++)  array [i] = i + 1 ;

   start = array.begin();
   end   = array.end();

   cout << "array = { " ;
   for(it = start; it != end; it++)
       cout << *it << " ";
   cout << " }" << endl;

   for_each(start, end, FunctorZeroOdd) ;
   cout << "\n\n" ;
   return 0;
}
```

Результат

```
array = { 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15  }
0 2 0 4 0 6 0 8 0 10 0 12 0 14 0
```

У цій програмі описаний шаблонний клас ZeroOdd, що містить перевантажену версію оператора виклику функції. Таким чином, об'єкти цього класу є функторами. Тепер досить визначити об'єкт класу ZeroOdd і передати його алгоритму for_each() як третій параметр, як показано вище.

Для того щоб спростити операції над функторами, що мають один чи два параметри, і стандартизувати імена типів аргументів і значень, що повертаються, у заголовку <functional> передбачені базові класи для унарних і бінарних функторів.

```
 template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
 }:

 template <class Arg1, class Arg2, class Result>
 struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
 };
```

Наприклад, ми могли б вивести клас ZeroOdd з базового класу unary-function.

```
template <typename T>
class ZeroOdd:public unary_function<T,T>
{
public:
void operator()(T i){ if (i%2) i = 0; cout << i << " " ;}
};
```

15.2. Арифметичні операції

Стандартна бібліотека містить опис функторів для виконання всіх арифметичних операцій.

```
template <class T>
struct plus : binary_function<T, T, T> {
   T operator()(const T& x, const T& y) const { return x + y; }
};

template <class T>
struct minus : binary_function<T, T, T> {
   T operator()(const T& x, const T& y) const { return x - y; }
};

template <class T>
struct multiplies : binary_function<T, T, T> {
   T operator()(const T& x, const T& y) const ( return x *  y; }
};

template <class T>
struct divides : binary_function<T, T, T> {
   T operator()(const T& x, const T& y) const { return x / y; }
};

template <class T>
struct modulus : binary_function<T, T, T> {
   T operator()(const T& x, const T& y) const { return x % y; }
};

template <class T>
struct negate : unary_function<T, T> {
   T operator()(const T& x) const { return -x; }
};
```

Покажемо, як за допомогою цих класів можна описати похідний клас, що містять перевантажені арифметичні оператори.

Виконання арифметичних операцій

```
 #include <functional>
#include <iostream>
#include <complex>

using namespace std ;
template <typename T>
class Arithmetic : public plus<T>,
                   public minus<T>,
                   public multiplies<T>,
                   public divides<T>
{
public:
   T value;
   Arithmetic(){value=0;}
   Arithmetic(T x){value=x;}
   result_type operator+(second_argument_type arg2)
                        {return value + arg2;}
   result_type operator-(second_argument_type arg2)
                        {return value - arg2;}
   result_type operator*(second_argument_type arg2)
                        {return value * arg2;}
```

```
    result_type operator/(second_argument_type arg2)
                          {return value / arg2;}
    result_type& operator=(result_type& arg) {return value=arg; }
};

int main()
{
    Arithmetic<complex<double> > obj1,obj2,obj3,obj4,obj5;
    complex<double> z(10,20);

    obj1 = z;
    cout << "obj1 = " << obj1.value << endl;

    obj2 = obj1 + 10;
    cout << "obj2 = obj1 + 10 = " << obj2.value << endl;

    obj3 = obj2 - 12;
    cout << "obj3 = obj2 - 12 = " << obj3.value << endl ;

    obj4 = obj3 * 40;
    cout << "obj4 = obj3 * 40 = " << obj4.value << endl ;

    obj5 = obj4/80;
    cout << "obj5 = obj4/80 = " << obj5.value << endl ;

    return 0;
}
```

Шаблонний клас `Arithmetic` є похідним від базових класів `plus`, `minus`, `multiplies`, `divides`, у яких визначені типи `result_type` і `second_argument_type`. Це дозволяє уніфікувати інтерфейс і виконувати всі операції одноманітно. Якби спадкування не було, можна було б визначити типи `result_type` і `second_argument_type` як тип `T`, і клас `Arithmetic` став би звичайним шаблоновим класом. Таким чином, спадкування арифметичних функторів у даному випадку впливає лише на вид інтерфейсу.

15.3. Предикати

У заголовку `<functional>` визначені базові класи функціональних об'єктів для всіх операторів порівняння.

```
template <class T>
struct equal_to : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x == y; }
};

template <class T>
struct not_equal_to : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x != y; }
};

template <class T>
struct greater : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x > y; }
};

template <class T>
struct less : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x < y; }
};

template <class T>
struct greater_equal : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x >= y; }
};

template <class T>
struct less_equal : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x <= y; }
};
```

Ці предикати дозволяють задавати операції порівняння, що використовуються в стандартних алгоритмах. От як, наприклад, можна упорядкувати целочисельний масив по зростанню.

Застосування предикатів

```
 #include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

int main()
{
   const int SIZE = 15 ;
   int array[SIZE]={15,14,13,12,11,10,9,8,7,6,5,4,3,2,1};
   sort (array, array + SIZE, less_equal<int> ());
   for(int i = 0; i < SIZE; i++) cout << array[i] << " ";
   cout << "\n\n" ;

   return 0;
}
```

Результат

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
```

Програміст може створювати власні предикати, підставляючи їх як параметри стандартних алгоритмів. Розглянемо наступний приклад.

Визначення власних предикатів

```
 #include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

template <typename T>
class IsMultiply7
{
public:
   bool operator()(T i){ if (i%7==0) return true; }
};

int main()
{
   const int SIZE = 15 ;
   vector<int> array(SIZE);
   vector<int>::iterator start, end, it;

   IsMultiply7<int> FunctorIsMultiply7;

   for (int i = 0; i < SIZE; i++)  array [i] = i + 1 ;

   start = array.begin();
   end   = array.end();

   cout << "array { " ;
   for(it = start; it != end; it++)
       cout << *it << " ";
   cout << " }\n" << endl;

   cout << *find_if(start, end, FunctorIsMultiply7) ;
   cout << "\n\n" ;

   return 0;
}
```

У цій програмі для пошуку першого елемента, кратного семи, використовується стандартний алгоритм `find_if()`. Цей алгоритм повертає ітератор, установлений на шуканий елемент. Умова, якій повинний

задовольняти шуканий елемент, визначається предикатом `FunctorIsMultiply7`, що представляє собою об'єкт класу `IsMultiply7`. Діапазон пошуку обмежений ітераторами `start` і `end`.

15.4. Логічні операції

Крім операцій порівняння, у стандартній бібліотеці визначені шаблонні класи, що дозволяють виконувати логічні операції. Їхні визначення приведені нижче.

```
 template <class T>
struct logical_and : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x && y; }
};

template <class T>
struct logical_or : binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x || y; }
};

 template <class T>
struct logical_not : unary_function<T, bool> {
    bool operator()(const T& x) const { return !x; }
};
```

Перевіримо виконання логічної операції `logical_and`. Для цього застосуємо до двох булевих масивів шаблонний алгоритм `transform`.

Виконання логічних операцій

```
 #include <iostream>
#include <functional>
#include <algorithm>
using namespace std;
int main()
{
   const int SIZE = 8;
   bool byte1 [SIZE] = { 1, 1, 0, 1 , 0, 1, 0, 0};
   bool byte2 [SIZE] = { 0, 1, 0, 0 , 1, 1, 1, 0};
   int byte3 [8];

   transform(byte1, byte1+8,byte2, byte3,logical_and<bool> ());

   cout << "Byte1 ";
   for (int i = 0; i < SIZE; i++)
   cout << byte1[i] << " ";
   cout << endl;

   cout << "Byte2 ";
   for (i = 0; i < SIZE; i++)
   cout << byte2[i] << " ";
   cout << endl;

   cout << "Byte3 ";
   for (i = 0; i < SIZE; i++)
   cout << byte3[i] << " ";
   cout << endl;

   return 0;
}
```

Функція `transform()` виконує бінарну операцію над двома заданими послідовностями. Першим параметром алгоритму `transform()` є ітератор, установлений на початок, другим — ітератор, установлений на кінець визначеного діапазону. Ці два параметри визначають границі першого аргументу бінарної операції. Третій параметр шаблонової функції `transform()` є другим параметром бінарної операції. Результат бінарної операції записується як четвертий параметр. Сама бінарна операція задається п'ятим параметром. У даному випадку функція `transform()` застосовує логічну операцію `logical_and` до двох булевим масивів: `byte1` і `byte2`. Результат записується в масив `byte3` і виводиться на екран.

Результат

```
Byte1 1 1 0 1 0 1 0 0
```

```
Byte2 0 1 0 0 1 1 1 0
Byte3 0 1 0 0 0 1 0 0
```

15.5 Адаптери

Стандартна бібліотека шаблонів містить функтори, що дозволяють створювати композиції функторів. До їхнього числа відносяться *зв'язувачі*, *інвертори*, *адаптери функцій-членів*, *адаптери вказівників на функцію* й *адаптери посилань на функцію*.

15.5.1 Зв'язувачі

У деяких ситуаціях зручно створювати одномісний предикат із уже існуючого двомісного, зв'язавши один з аргументів фіксованим значенням.

Наприклад, операція `less` вимагає при кожнім звертанні явно вказувати обидва аргументи, наприклад `less<int>(1,2)`. Для того щоб можна було порівнювати об'єкти, використовуючи стандартні алгоритми, необхідно або написати власний предикат (наприклад, `IsMultiply7`), або *зв'язати* другий аргумент.

У стандартній бібліотеці передбачено два механізми зв'язування: один зв'язує перший аргумент предиката, а іншої — другий аргумент. Кожний з них використовує шаблонні класи `binder1st` і `binder2nd` відповідно.

От як виглядають визначення відповідних функцій.

```
template<class Operation, class T> binder1st <Operation>
 bind1st(const Operation& op, const T& x);

template<class Operation, class T> binder2nd <Operation>
 bind2nd(const Operation& op, const T& y);
```

Функція `bind1st()` повертає унарний функтор, у якого лівий операнд `op` зв'язаний зі значенням `x`. Відповідно, функція `bind2nd()` повертає унарний функтор, у якого зв'язаний правий операнд.

Нижче приведений приклад, у якому використаний зв'язувач другого аргументу. У цій задачі потрібно вивести на екран перший елемент масиву, не переважаюче число 10.

Застосування зв'язувачів

```
 #include <iostream>
#include <vector>
#include <algorithm>
#include <functional>

using namespace std;

template <typename T>
class IsLess10
{
public:
   bool operator()(T i){ if (i<10) return true; }
};


int main()
{
   const int SIZE = 15 ;
   vector<int> array(SIZE);
   vector<int>::iterator start, end, it;

   IsLess10<int> FunctorIsLess10;

   for (int i = 0; i < SIZE; i++)  array [i] = i + 1 ;

   start = array.begin();
   end   = array.end();

   cout << "array { " ;
   for(it = start; it != end; it++)
       cout << *it << " ";
   cout << " }\n" << endl;
```

```
    cout << *find_if(start, end, bind2nd(less<int>(),10));
    cout << "\n\n" ;

    return 0;
}
```

Безпосередньо застосувати предикат `less<int>` у даному випадку не можна, оскільки функції `find_if()` необхідний унарний предикат. Щоб вирішити поставлену задачу, ми зв'язали другий аргумент.

Програма виводить на екран число 1.

15.5.2. Інвертори

З предикатами тісно зв'язаний ще один вид функторів — *інвертори*. Ці функтори повертають заперечення предиката. Для кожного різновиду предикатів призначений свій функтор. Так, для заперечення унарних предикатів необхідний інвертор `not1()`, а для заперечення бінарних предикатів — функтор `not2()`. Ці функції повертають об'єкти класів `unary_negate` і `binary_negate` відповідно.

```
template <class Predicate>
   class unary_negate:
     public unary_function<typename Predicate::argument_type, bool> {
public:
   explicit unary_negate(const Predicate& pred);
   bool operator() (const typename Predicate::argument_type& x) const;
};
template <class Predicate>
   unary_negate<Predicate<> not1(const Predicate& pred);
```

Функція `operator()` повертає заперечення предиката `pred(x)`.

```
template <class Predicate>
   class binary_negate:
     public binary_function<typename Predicate::first_argument_type,
                            typename Predicate::first_argument_type, bool> {
public:
   explicit binary_negate(const Predicate& pred);
   bool operator() (const typename Predicate::first_argument_type& x) const;
};
template <class Predicate>
   binary_negate<Predicate<> not2(const Predicate& pred);
```

За допомогою інвертора `not1` легко одержати заперечення унарного предиката `bind2nd(less<int>(),10)` і вивести на екран перший елемент масиву `array`, що чи більше дорівнює 10. Для цього досить викликати функцію `find_if()`, як показано нижче.

```
cout << *find_if(start, end, not1(bind2nd(less<int>(),10)));
```

Тепер програма виведе на екран число 10.

15.5.3. Адаптери вказівників на функцію

У заголовку `<functional>` визначено декілька класів, що дозволяють перетворити вказівник на функцію так, щоб його можна було використовувати в стандартних алгоритмах. Приведемо їхнього визначення.

Клас вказівників на унарні функції називається `pointer_to_unary_function`.

```
template <class Argument, class Result>
 pointer_to_unary_function : public unary_function<Argument, Result> {
public:
 explicit pointer_to_unary_function(Result (*f)(Arg));
 Result operator()(Arg x) const;
};
```

Функція `operator()` повертає `f(x)`.

Адаптер вказівника на функцію `ptr_fun()` визначений у такий спосіб.

```
template <class Arg, class Result>
 pointer_to_unary_function<Arg, Result>
   ptr_fun(Result(*f)(Arg));
```

Клас вказівників на бінарні функції називається `pointer_to_binary_function()`.

```
template <class Arg1, class Arg2, class Result>
 pointer_to_binary_function:
   public binary_function<Arg1, Arg2, Result> {
 public:
```

```
   explicit pointer_to_binary_function(Result (*f)(Arg1, Arg2));
   Result operator()(Arg1 x, Arg2 y) const;
```

Функція `operator()` повертає `f(x,y)`.

Адаптер вказівника на функцію `ptr_fun()` визначений у такий спосіб.

```
template <class Arg, class Result>
 pointer_to_binary_function<Arg, Result>
   ptr_fun(Result(*f)(Arg1, Arg2));
```

Функція `ptr_fun()` повертає об'єкт класу `pointer_to_unary_function` чи `pointer_to_binary_function`.

Повернемося до задачі, у якій було потрібно знайти перший елемент масиву, кратний 7. Не прибігаючи до створення функтора, скористаємося адаптером функції.

Застосування адаптерів вказівників на функцію

```
 #include <iostream>
#include <algorithm>
#include <functional>

using namespace std;

bool IsMultiply7 (int n) {  return (n % 7) == 0;}

int main()
{
   const int SIZE = 15;
   int array [15] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
   int* p = find_if(array, array + SIZE, ptr_fun (IsMultiply7));
   if (p != array + SIZE)  cout << *p << endl;
   return 0;
}
```

Програма виводить на екран число 7.

15.5.4. Адаптери вказівників на функції-члени класу

Адаптери вказівників на функцію-член класу називаються `mem_fun` і `mem_fun1`. Вони повертають об'єкти класів `mem_fun_t` і `mem_fun_t1`.

Нижче приведене визначення класу `mem_fun_t`.

```
template <class S, class T> class mem_fun_t :
  public unary_function<T*, S> {
public:
  explicit mem_fun_t(T::*p)());
  S operator()(T* p) const;
};
```

Функція `mem_fun()` повертає об'єкт класу `mem_fun_t`. Він являє собою функтор, що викликає функцію-член, на яку посилається аргумент. Визначення цієї функції виглядає так.

```
template<class S, class T> mem_fun_t<S,T>
  mem_fun(S (T::*f)());
```

Визначення класу `mem_fun1_t` виглядає в такий спосіб.

```
template <class S, class T, class A> class mem_fun1_t :
  public binary_function<T*, A, S> {
public:
  explicit mem_fun1_t(S (T::*p)(A));
  S operator()(T* p, A x) const;
};
```

Функція `mem_fun1()` повертає об'єкт класу `mem_fun1_t`. Цей функтор викликає функція-член, на яку посилається вказівник, що є першим параметром, і передає їй аргумент, що відповідає другому параметру.

Визначення цієї функції виглядає так.

```
template<class S, class T, class A> mem_fun1_t<S,T,A>
  mem_fun(S (T::*f)(A));
```

Визначення класу `mem_fun1_t` виглядає в такий спосіб.

```
template <class S, class T, class A> class mem_fun1_t :
  public binary_function<T*, A, S> {
public:
  explicit mem_fun1_t(S (T::*p)(A));
```

```
  S operator()(T* p, A x) const;
};
```

Функція `mem_fun1()` повертає об'єкт класу `mem_fun1_t`. Цей функтор викликає функція-член, на яку посилається вказівник, що є першим параметром, і передає їй аргумент, що відповідає другому параметру.

Визначення цієї функції виглядає так.

```
template<class S, class T, class A> mem_fun1_t<S,T,A>
  mem_fun(S (T::*f)(A));
```

Розглянемо приклад, у якому виконується подвоєння елементів вектора.

Застосування адаптерів вказівників на функцію-член класу

```
 #include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;

class TComplex
{
  double Re;
  double Im;
public:
   TComplex(double x=1, double y=2):Re(x), Im(y){}
   TComplex print()
   {
       Re = 2*Re;
       Im = 2*Im;
       cout << Re << "+i* "<< Im << endl;
       return *this;
   }
};


int main()
{
   const int SIZE = 2;
   TComplex array[2];
   vector<TComplex*> v(2);
   TComplex u(1,2), w(3,4);
   v[0]=&u;
   v[1]=&w;
   for_each(v.begin(), v.end(), mem_fun(&TComplex::print));

   return 0;
}
```

Зверніть увагу на те, що для використання адаптера вказівника на функцію-член необхідно, щоб вектор містив вказівники на об'єкти класу `TComplex`, а не самі об'єкти. Якщо вектор містить об'єкти, варто застосувати адаптер посилання на функцію-член.

15.5.5. Адаптери посилань на функції-члени класу

Адаптери посилань на функцію-член класу називаються `mem_fun_ref()` і `mem_fun1_ref()`. Вони повертають об'єкти класів `mem_fun_ref_t` і `mem_fun_ref_t1`.

Нижче приведене визначення класу `mem_fun_ref_t`.

```
template <class S, class T>  mem_fun_ref_t :
  public unary_function<T, S> {
public:
  explicit mem_fun_ref_t(S (T::*p)());
  S operator()(T& p) const;
};
```

Функція `mem_fun_ref()` повертає об'єкт класу `mem_fun_t`. Він являє собою функтор, що викликає функцію-член, на яку встановлена посилання. Визначення цієї функції виглядає так.

```
template<class S, class T> mem_fun_ref_t<S,T>
  mem_fun_ref(S (T::*f)());
```

Визначення класу `mem_fun1_ref_t()` виглядає в такий спосіб.

```
template <class S, class T, class A> class mem_fun1_ref_t :
  public binary_function<T, A, S> {
public:
  explicit mem_fun1_ref_t(S (T::*p)(A));
  S operator()(T& p, A x) const;
};
```

Функція `mem_ref_fun1()` повертає об'єкт класу `mem_fun1_t`. Цей функтор викликає функція-член, на яку встановлене посилання, задана першим параметром, і передає їй аргумент, що відповідає другому параметру.

Визначення цієї функції виглядає так.

```
template<class S, class T, class A> mem_ref_fun1_t<S,T,A>
  mem_fun(S (T::*f)(A));
```

Розв'язання задачі, розглянутої вище, тепер можна записати інакше.

Застосування адаптерів посилань на функції-члени класу

```
 #include <iostream>
#include <algorithm>
#include <functional>
#include <vector>

using namespace std;

class TComplex
{
   double Re;
   double Im;
public:
   TComplex(double x=1, double y=2):Re(x), Im(y){}
   TComplex print()
   {
       Re = 2*Re;
       Im = 2*Im;
       cout << Re << "+i*"<< Im << endl;
       return *this;
   }
};


int main()
{
   const int SIZE = 2;
   TComplex array[2];
   vector<TComplex> v(2);
   TComplex u(1,2), w(3,4);
   v[0]=u;
   v[1]=w;
   for_each(v.begin(), v.end(), mem_fun_ref(&TComplex::print));

   return 0;
}
```

Результат

```
2+i*4
6+i*8
```

15.6. Розподільники

Одним з основних вимог, пропонованих до програм, є мобільність, тобто можливість виконувати їх на різних комп'ютерах. Для цього необхідно інкапсулювати в об'єктах інформацію про моделі пам'яті, зокрема типи вказівників, тип відстані між вказівниками, тип розміру об'єктів, а також функції, що виділяють і звільняють пам'ять.

У бібліотеці STL ця задача вирішена за допомогою *розподільників* — об'єктів, що інкапсулюють інформацію про моделі пам'яті.

Як зазначено вище, усі розподільники пам'яті повинні містити визначення наступних типів і функцій.

Змінні-члени

`const_pointer`	Константний вказівник на об'єкт класу `value_type`
`const_reference`	Константне посилання на об'єкт класу `value_type`
`difference_type`	Різниця між двома адресами
`pointer`	Вказівник на об'єкт класу `value_type`
`reference`	Посилання на об'єкт класу `value_type`
`size_type`	Тип, що дозволяє зберігати розмір найбільшого можливого об'єкта, розташовуваного в пам'яті
`value_type`	Тип об'єкта, розташовуваного в пам'яті

15.7. Резюме

- *Функторами* називаються об'єкти, що інкапсулюють виклик функції, тобто містять перевантажену версію операторної функції `operator()`. Ці об'єкти передаються шаблонним алгоритмам як вказівники на функцію. Оскільки код функтора не викликається, а підставляється безпосередньо в текст програми, ефективність алгоритму підвищується.

- Стандартна бібліотека містить опис функторів для виконання всіх арифметичних операцій: `plus, minus, multiplies, divides, modulus, negate`.

- Шаблонний клас `Arithmetic` є похідним від базових класів `plus`, `minus`, `multiplies`, `divides`, у яких визначені типи `result_type` і `second_argument_type`. Це дозволяє уніфікувати інтерфейс і виконувати всі операції одноманітно. Якби спадкування не було, можна було б визначити типи `result_type` і `second_argument_type` як тип `T`, і клас `Arithmetic` став би звичайним шаблоновим класом. Таким чином, спадкування арифметичних функторів у даному випадку впливає лише на вид інтерфейсу.

- У заголовку `<functional>` визначені базові класи функціональних об'єктів для всіх операторів порівняння: `equal_to, greater, less, greater_equal, less_equal.`

- У стандартній бібліотеці визначені шаблонні класи, що дозволяють виконувати логічні операції: `logical_and, logical_or, logical_not.`

- Стандартна бібліотека шаблонів містить функтори, що дозволяють створювати композиції функторів. До їхнього числа відносяться *зв'язувачі*, *інвертори*, *адаптери функцій-членів*, *адаптери вказівників на функцію* й *адаптери посилань на функцію*.

- У стандартній бібліотеці передбачено два механізми зв'язування: один зв'язує перший аргумент предиката, а іншої — другий аргумент. Кожний з них використовує шаблонні класи `binder1st` і `binder2nd` відповідно. Функція `bind1st()` повертає унарний функтор, у якого лівий операнд `op` зв'язаний зі значенням `x`. Відповідно, функція `bind2nd()` повертає унарний функтор, у якого зв'язаний правий операнд.

- З предикатами тісно зв'язаний ще один вид функторів — *інвертори*. Ці функтори повертають заперечення предиката. Для кожного різновиду предикатів призначений свій функтор. Так, для заперечення унарних предикатів необхідний інвертор `not1()`, а для заперечення бінарних предикатів — функтор `not2()`. Ці функції повертають об'єкти класів `unary_negate` і `binary_negate` відповідно.

- У заголовку `<functional>` визначено декілька класів, що дозволяють перетворити вказівник на функцію так, щоб його можна було використовувати в стандартних алгоритмах: `pointer_to_unary_function, pointer_to_binary_function.`

- Адаптери вказівників на функцію-член класу називаються `mem_fun` і `mem_fun1`. Вони повертають об'єкти класів `mem_fun_t` і `mem_fun_t1`.

- Адаптери посилань на функцію-член класу називаються `mem_fun_ref()` і `mem_fun1_ref()`. Вони повертають об'єкти класів `mem_fun_ref_t` і `mem_fun_ref_t1`.

- Одним з основних вимог, пропонованих до програм, є мобільність, тобто можливість виконувати їх на різних комп'ютерах. Для цього необхідно інкапсулювати в об'єктах інформацію про моделі пам'яті, зокрема типи вказівників, тип відстані між вказівниками, тип розміру об'єктів, а також функції, що виділяють і звільняють пам'ять. У бібліотеці STL ця задача вирішена за допомогою *розподільників* — об'єктів, що інкапсулюють інформацію про моделі пам'яті.

15.7. Контрольні питання

1. Що називається функтором?

2. Які функтори здійснюють арифметичні операції?
3. Опишіть шаблонний клас `Arithmetic`.
4. Які базові класи функціональних об'єктів визначені для операторів порівняння?
5. Які шаблонні класи дозволяють виконувати логічні операції?
6. Які функтори дозволяють створювати композиції функторів?
7. Які механізми зв'язування передбачено у стандартній бібліотеці?
8. Що таке інвертори?
9. Які класи дозволяють перетворити вказівник на функцію так, щоб його можна було використовувати в стандартних алгоритмах: `pointer_to_unary_function, pointer_to_binary_function.`
10. Опишіть адаптери вказівників на функцію-член класу.
11. Опишіть адаптери посилань на функцію-член класу.
12. Що таке розподільники?