12. Шаблони и вложени класове

План:
Шаблони
Шаблони и наследяване
Вложени класове

Шаблони.

Класове-шаблони дават възможност да се конструират обекти с данни от произволен тип.
Вече са използвани класове-шаблони при конструиране на обекти.

```
vector<int> v_i;
vector<double> v_d;
vector<Employee> v_e;
```

Тук `int`, `double` и `Employee` са параметри на класа-шаблон `vector`, дефиниран в STL.

** Типове като параметри на шаблона

За да дефинираме клас-шаблон, означаваме произволен тип с `T` и добавяме `template<typename  T>` преди дефиницията на класа.
Пример. Дефинираме клас-шаблон наредена двойка елементи с данни от произволен тип.

```
template<typename T>
class Pair {
public:
   Pair(T a, T b);
   T get_first() const;
   T get_second() const;
private:
   T first;
   T second;
};
```

Всички член-функции се дефинират също като шаблони.

```
// pairs.cpp
#include <iostream>
 #include <string>
 using namespace std;

 template<typename T>
 class Pair {
 public:
    Pair(T a, T b);
    T get_first() const;
```

```
   T get_second() const;
   void print() const;
private:
   T first;
   T second;
};

template<typename T>
Pair<T>::Pair(T a, T b)
{  first = a;
   second = b;
}

template<typename T>
T Pair<T>::get_first() const
{  return first; }

template<typename T>
T Pair<T>::get_second() const
{  return second; }

template<typename T>
void Pair<T>::print() const
{ cout << "Pair: (" << first << ","
       << second << ")" << endl;  }

int main()
{
   Pair<int> integers(10,22);
   integers.print();

   Pair<double> doubles(1.5, 2.25);
   doubles.print();

   Pair<string> strings("One", "Two");
   strings.print();
   return 0;
}
```

Пример. Класът **`List`** (**`list2.cpp`**, **`list0.cpp`**) съхранява свързан списък от низове.
Използвайки шаблони, **`List`** ще може да съхранява стойности от произволен тип, както това става в стандартния клас **`list`** от STL.
За тази цел декларираме клас-шаблон, като задаваме формален параметър **`T`** на шаблона:

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

При създаване на обект от този клас, съдържащ низове, задаваме фактически параметър `string` на шаблона по познатата схема:

```
List<string> staff;
```

Дефиниция на класа-шаблон `List` в сравнение с по-рано дефинирания клас `List`:

```
template<typename T>
class List {
public:
   List();
   void push_back(T s);
   void insert(Iterator<T> pos, T s);
   void erase(Iterator<T> pos);
   Iterator<T> begin();
   Iterator<T> end();
private:
   Node<T>* first;
   Node<T>* last;
};
```

```
class List {
public:
   List();
   void push_back(string s);
   void insert(Iterator pos, string s);
   Iterator erase(Iterator pos);
   Iterator begin();
   Iterator end();
private:
   Node* first;
   Node* last;
};
```

Ще пренапишем класовете за свързан списък, като използваме шаблони.

```
// list.cpp
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

/* forward declarations */
template<typename T> class List;
template<typename T> class Iterator;

/**   A class to hold the nodes of the linked list. */
template<typename T>
class Node {
public:
/**  Constructs a node for a given data value.
     @param s the data to store in this node */
   Node(T s);
private:
   T data;
   Node<T>* previous;
   Node<T>* next;
friend class List<T>;
friend class Iterator<T>;
};

/** An iterator denotes a position in the list or
    past the end of the list. */
template<typename T>
class Iterator {
public:
   /**   Constructs an iterator that is not attached to any list. */
   Iterator();

  /**     Looks up the value at a position.
          @return the value of the Node to which the iterator points  */
   T operator*() const;

   /**   Advances the iterator to the next position. */
   void operator++(int dummy);

   /**   Moves the iterator to the previous position. */
   void operator--(int dummy);

   /**   Compares two iterators.
      @param b the iterator to compare with this iterator
      @return true if this iterator and b are equal */
   bool operator==(Iterator<T> b) const;

   /**   Compares two iterators.
      @param b the iterator to compare with this iterator
      @return true if this iterator and b are not equal */
   bool operator!=(Iterator<T> b) const;
private:
   Node<T>* position;
   Node<T>* last;
friend class List<T>;
};

/**A linked list of values of a given type.
```

```
   @param T the type of the list values */
template<typename T>
class List {
public:
   /**   Constructs an empty list. */
   List();

   /**   Constructs a list as a copy of another list.
      @param b the list to copy */
   List(const List<T>& b);

   /**   Deletes all nodes of this list. */
   ~List();

   /** Assigns another list to this list.
      @param b the list to assign
      @return a reference to this list */
   List<T>& operator=(const List<T>& b);

   /**  Appends an element to the list.
      @param s the value to append */
   void push_back(T s);

   /**  Inserts an element into the list.
      @param iter the position before which to insert
      @param s the value to append */
   void insert(Iterator<T> iter, T s);

   /**  Removes an element from the list.
      @param i the position to remove
      @return an iterator pointing to the element after the
      erased element */
   Iterator<T> erase(Iterator<T> i);

   /**  Gets the beginning position of the list.
      @return an iterator pointing to the beginning of the list */
   Iterator<T> begin() const;

   /** Gets the past-the-end position of the list.
      @return an iterator pointing past the end of the list */
   Iterator<T> end() const;
private:
   /**  Copies another list to this list.
      @param b the list to copy */
   void copy(const List<T>& b);

   /**  Deletes all nodes of this list. */
   void free();

   Node<T>* first;
   Node<T>* last;
};

template<typename T>
List<T>::List()
{  first = NULL;
   last = NULL;
}

template<typename T>
List<T>::~List()
{  free();
}

template<typename T>
List<T>::List(const List<T>& b)
{  first = NULL;
   last = NULL;
   copy(b);
}

template<typename T>
List<T>& List<T>::operator=(const List<T>& b)
{  if (this != &b)
   {
      free(); copy(b);
   }
   return *this;
}
```

```
template<typename T>
void List<T>::push_back(T s)
{  Node<T>* newnode = new Node<T>(s);
   if (last == NULL) /* list is empty */
   {
      first = newnode;
      last = newnode;
   } else
   {
      newnode->previous = last;
      last->next = newnode;
      last = newnode;
   }
}

template<typename T>
void List<T>::insert(Iterator<T> iter, T s)
{  if (iter.position == NULL)
   {
      push_back(s);
      return;
   }
   Node<T>* after = iter.position;
   Node<T>* before = after->previous;
   Node<T>* newnode = new Node<T>(s);
   newnode->previous = before;
   newnode->next = after;
   after->previous = newnode;
   if (before == NULL) /* insert at beginning */
      first = newnode;
   else
      before->next = newnode;
}

template<typename T>
Iterator<T> List<T>::erase(Iterator<T> i)
{ Iterator<T> iter = i;
   assert(iter.position != NULL);
   Node<T>* remove = iter.position;
   Node<T>* before = remove->previous;
   Node<T>* after = remove->next;
   if (remove == first)
      first = after;
   else
      before->next = after;
   if (remove == last)
      last = before;
   else
      after->previous = before;
   iter.position = after;
   delete remove;
   return iter;
}

template<typename T>
Iterator<T> List<T>::begin() const
{  Iterator<T> iter;
   iter.position = first;
   iter.last = last;
   return iter;
}

template<typename T>
Iterator<T> List<T>::end() const
{  Iterator<T> iter;
   iter.position = NULL;
   iter.last = last;
   return iter;
}

template<typename T>
Iterator<T>::Iterator()
{  position = NULL;
   last = NULL;
}

template<typename T>
T Iterator<T>::operator*() const
{  assert(position != NULL);
   return position->data;
```

```
}

template<typename T>
void Iterator<T>::operator++(int dummy)
{ assert(position != NULL);
   position = position->next;
}

template<typename T>
void Iterator<T>::operator--(int dummy)
{  if (position == NULL)   position = last;
   else  position = position->previous;
   assert(position != NULL);
}

template<typename T>
bool Iterator<T>::operator==(Iterator<T> b) const
{  return position == b.position;
}

template<typename T>
bool Iterator<T>::operator!=(Iterator<T> b) const
{  return position != b.position;
}

template<typename T>
Node<T>::Node(T s)
{  data = s;
   previous = NULL;
   next = NULL;
}

template<typename T>
void List<T>::copy(const List<T>& b)
{   for (Iterator<T> p = b.begin(); p != b.end(); p++)
      push_back(*p);
}

template<typename T>
void List<T>::free()
```

```
{   while (begin() != end())  erase(begin());
}

int main()
{
   List<string> staff;
   staff.push_back("Cracker, Carl");
   staff.push_back("Hacker, Harry");
   staff.push_back("Lam, Larry");
   staff.push_back("Sandman, Susan");

   /* add a value in fourth place */
   Iterator<string> pos;
   pos = staff.begin();
   pos++;
   pos++;
   pos++;

   staff.insert(pos, "Reindeer, Rudolf");

   /* remove the value in second place */
   pos = staff.begin();
   pos++;

   staff.erase(pos);

   /* print all values */
   for (pos = staff.begin(); pos != staff.end(); pos++)
      cout << *pos << "\n";
   return 0;
}
```

**** Променливи като параметри на шаблон**

Освен имена на типове, параметри на шаблона могат да бъдат и променливи.

Пример:

```
  template<typename T, int ROWS, int COLUMNS>
  class Matrix {
  public:
```

```
  ...
  private:
     T data[ROWS][COLUMNS];
  };
```

За да конструираме обекти от този клас, задаваме стойности на параметрите-променливи на шаблона (размерите на матрицата).

```
Matrix<double, 3, 4> a; // A 3 × 4 matrix of double values
Matrix<string, 2, 2> b;
```

Операция присвояване е възможна само за обекти с еднакви типове и размери.

```
  Matrix<int, 3, 4> a;
  Matrix<double, 3, 4> b;
  Matrix<int, 5, 7> c;
  Matrix<int, 3, 4> d;
  b = a; // Error, element types don’t match.
  c = a; // Error, sizes don’t match, so types differ.
  d = a; // OK. Element types and sizes match.
```

Шаблони и наследяване

Пример: Клас-шаблон като базов и производен клас

```
// inh_t.cpp
#include <iostream>
#include <string>
using namespace std;

template<typename T>
class A {
public:
    A(T aa):a(aa){}
    T geta() const { return a; }
private:
    T a;
};

class B : public A<int> {
public:
    B(int bb):A(bb){};
};

template<typename T>
class C : public A<int> {
public:
    C(T cc, int aa):A(aa){ c = cc; }
    T getc() const { return c; }
private:
    T c;
};

template<typename T>
class D : public A<T> {
public:
    D(T dd):A<T>(dd){}
};

int main()
{
    A<int> a1(10);
    B b1(20);
    cout << b1.geta() << endl;
    C<string> c1("abc", 30);
    cout << c1.geta() << " " << c1.getc() << endl;
    D<double> d1(0.5);
    cout << d1.geta() << endl;
   return 0;
}
```

Пример: Клас-шаблон като производен клас

```
// inh_t1.cpp
#include <iostream>
#include <string>
using namespace std;

class A {
public:
    A(int aa):a(aa){}
    int geta() const { return a; }
private:
    int a;
};

template<typename T>
class B : public A {
public:
    B(T bb, int aa):A(aa),b(bb){}
    T getb() const { return b; }
private:
    T b;
};

int main()
{
    A a1(10);
    cout << a1.geta() << endl;
    B<string> b1("abc", 20);
    cout << b1.geta() << " " << b1.getb() << endl;
   return 0;
}
```

Вложени класове

В STL класът `iterator` е дефиниран в класа `list`:

```
list<string>::iterator pos = staff.begin();
```

За да се вложи един клас в друг, вътрешният клас се дефинира във външния клас:

```
class List {
   ...
   class Iterator;
   ...
};
```

Пример. Класът `List` със същия интерфейс, както и класът `list` от STL.

```
// list1.cpp
#include <string>
#include <iostream>
#include <cassert>
using namespace std;

template<typename T> class List;

template<typename T>
class Node {
public:
   Node(T s);
private:
   T data;
   Node<T>* previous;
   Node<T>* next;
friend class List<T>;
friend class List<T>::Iterator;
};

template<typename T>
class List {
public:
   List();
   List(const List<T>& b);
   ~List();
   List<T>& operator=(const List<T>& b);
   class Iterator;

   void push_back(T s);
   void insert(Iterator iter, T s);
   Iterator erase(Iterator i);
   Iterator begin();
   Iterator end();
private:
   void copy(const List<T>& b);
   void free();
   Node<T>* first;
   Node<T>* last;
};
```

```
template<typename T>
class List<T>::Iterator {
public:
   Iterator();
   T operator*() const;
   void operator++(int dummy);
   void operator--(int dummy);
   bool operator==(Iterator b) const;
   bool operator!=(Iterator b) const;
private:
   Node<T>* position;
   Node<T>* last;
friend class List<T>;
};

template<typename T>
List<T>::List()
{  first = NULL;
   last = NULL;
}

template<typename T>
List<T>::~List()
{  free();
}

template<typename T>
List<T>::List(const List<T>& b)
{  first = NULL;
   last = NULL;
   copy(b);
}

template<typename T>
List<T>& List<T>::operator=(const List<T>& b)
{  if (this != &b)
   {  free();
      copy(b);
   }
   return *this;
}

template<typename T>
void List<T>::push_back(T s)
{  Node<T>* newnode = new Node<T>(s);
   if (last == NULL) /* list is empty */
   {  first = newnode;
      last = newnode;
   }
   else
   {  newnode->previous = last;
      last->next = newnode;
      last = newnode;
   }
}

template<typename T>
void List<T>::insert(Iterator iter, T s)
{  if (iter.position == NULL)
   {  push_back(s);
      return;
   }
   Node<T>* after = iter.position;
   Node<T>* before = after->previous;
   Node<T>* newnode = new Node<T>(s);
   newnode->previous = before;
   newnode->next = after;
   after->previous = newnode;
   if (before == NULL) /* insert at beginning */
      first = newnode;
   else
      before->next = newnode;
}

template<typename T>
typename List<T>::Iterator List<T>::erase(Iterator i)
{  Iterator iter = i;
   assert(iter.position != NULL);
   Node<T>* remove = iter.position;
   Node<T>* before = remove->previous;
   Node<T>* after = remove->next;

   if (remove == first) first = after;
   else          before->next = after;
   if (remove == last) last = before;
   else     after->previous = before;

   iter.position = after;
   delete remove;
   return iter;
}

template<typename T>
typename List<T>::Iterator List<T>::begin()
{  Iterator iter;
   iter.position = first;
   iter.last = last;
   return iter;
}

template<typename T>
typename List<T>::Iterator List<T>::end()
{  Iterator iter;
   iter.position = NULL;
   iter.last = last;
   return iter;
}

template<typename T>
List<T>::Iterator::Iterator()
{  position = NULL;
   last = NULL;
}

template<typename T>
T List<T>::Iterator::operator*() const
{  assert(position != NULL);
   return position->data;
}
```

```
template<typename T>
void List<T>::Iterator::operator++(int dummy)
{  assert(position != NULL);
   position = position->next;
}

template<typename T>
void List<T>::Iterator::operator--(int dummy)
{  if (position == NULL) position = last;
   else                  position = position->previous;
   assert(position != NULL);
}

template<typename T>
bool List<T>::Iterator::operator==(Iterator b) const
{  return position == b.position;
}

template<typename T>
bool List<T>::Iterator::operator!=(Iterator b) const
{  return position != b.position;
}

template<typename T>
Node<T>::Node(T s)
{  data = s;
   previous = NULL;
   next = NULL;
}

template<typename T>
void List<T>::copy(const List<T>& b)
{  for (Iterator p = b.begin(); p != b.end(); p++)
      push_back(*p);
}

template<typename T>
void List<T>::free()
{  while (begin() != end()) erase(begin());
}

int main()
{  List<string> staff;

   staff.push_back("Cracker, Carl");
   staff.push_back("Hacker, Harry");
   staff.push_back("Lam, Larry");
   staff.push_back("Sandman, Susan");

   /* add a value in fourth place */
   List<string>::Iterator pos;
   pos = staff.begin();
   pos++;
   pos++;
   pos++;

   staff.insert(pos, "Reindeer, Rudolf");

   /* remove the value in second place */
   pos = staff.begin();
   pos++;

   staff.erase(pos);

   /* print all values */
   for (pos = staff.begin(); pos != staff.end(); pos++)
      cout << *pos << "\n";

   return 0;
}
```