Стандартная библиотека шаблонов в языке программирования С++ (Standard Template Library -STL)

Лекция 1

Шаблоны функций и классов

Шаблоны

STL основывается на относительно новом понятии ***шаблона.***

Предположим, что для некоторого числа ***x > 0*** нужно часто вычислять значение выражения:

2 * x + (x * x + 1) / (2 * x),

где ***x*** может быть типа ***double*** или ***int.*** Например, если ***x*** имеет тип ***double*** и равен 5.0, тогда значение выражения составляет 12.6, но если ***x*** имеет тип ***int*** и равен 5, то значение выражения будет 12.

Шаблонные функции

Вместо того чтобы писать две функции:

```
double  f(double  x)
{   double  x2  =  2  *  x;
     return x2  +  (x  *  x  +  1)/x2;
}
```

и

```
int f(int x)
{   int x2 = 2 * x;
    return x2 +  (x * x + 1)/x2;
}
```

нам достаточно создать один ***шаблон:***

Шаблонные функции

```
// ftempl.cpp: Шаблонная функция.
#include <iostream.h>
 template <class T>
T f (T x)
{    T x2 = 2 * x;
     return x2 + (x * x + l)/x2;
}
int main()
{    cout << f(5.0) << endl << f(5) << endl;
    return 0;
}
```

Программа выведет: 12.6   12

В этом шаблоне T – тип, задаваемый аргументом при вызове f.

Шаблонные классы

Пусть нам нужен класс Pair, чтобы хранить пары значений. Иногда оба значения принадлежат к типу *double*, иногда к типу *int*. Тогда вместо двух новых классов:

```
class  PairDouble
{   public:
    PairDouble(double xl, double yl): x(xl), y(yl) {}
    void showQ();
    private:
    double x, y;
};
void PairDouble::showQ()
{
    cout << x/y << endl;
}
```

Шаблонные классы

После чего следует аналогичный фрагмент с классом PairInt для типа *int.* Вместо этого напишем один шаблонный класс:

```
// cltempl.cpp:  Шаблонный класс.
#include <iostream.h>
template <class T>
class  Pair
{    public:
     Pair(T xl, T yl): x(xl), y(yl){}
// Начальная инициализация x(xl), y(yl)
          void showQ();
     private:
          T x, y;
};
```

Шаблонные классы

```
template <class T>
void Pair<T>::showQ()
{
    cout << x/y << endl;
}
int main()
{
 Pair<double> a(37.0,  5.0);
 Pair<int> u(37,  5);
 a.showQ();
 u.showQ();
 return  0;
}
```

Замечания

Как пользователи STL мы можем не беспокоиться об определениях, так как шаблонные функции и классы STL доступны в виде *файлов заголовков,* которые можно использовать, не вдаваясь в подробности их программирования. Единственный аспект применения шаблонов, который мы увидим в наших программах,- это обозначение фактического типа с помощью конструкции наподобие *Pair<double>.*

Процесс создания конкретной версии шаблонной функции называется *инстанцированием* шаблона или *созданием экземпляра.*

Пространства имен

Существует другой новый элемент языка, который мы обязаны принять во внимание. Если программа состоит из многих файлов, нужно принять меры во избежание *конфликта имен.* Концепция пространства имен может быть хорошим способом решения этой задачи.

```
// namespac.cpp: Пространства имен.
#include <iostream.h>
namespace A
{   int i = 10;
}
namespace B
{   int i = 20;
}
```

Пространства имен

```
void  fA()
{    using namespace A;
 cout << "In  fA: " <<
 A::i << " " << B::i << " " << i << endl;
}
void fB()
{    using namespace B;
 cout << "In  fB: " <<
 A::i << " " << B::i << " " << i << endl;
}  ________________________________
int main()
{    fA();  fB();
cout << "In main: " << A::i << " " << B::i << endl;
//  cout << i << endl;  Здесь это недопустимо.
using A::i;
cout << i << endl;  // А это разрешено.
return  0; }
```

Пространства имен

Эта программа на выходе даст:

In fA: 10 20 10
In fB: 10 20 20
In main: 10 20
10

Благодаря идентификаторам A и B мы впоследствии можем ссылаться на эти пространства имен. Для пространства имен A можем написать либо что-нибудь вроде:

A:: … либо одно из выражений:

```
using namespace A;
using A:: i;
```

Ранее было:

```
void Pair<T>::showQ() ….
```

Знакомство с STL

Рассмотрим программу, которая читает с клавиатуры переменное количество ненулевых целых чисел и печатает их в том же порядке после того, как введен 0.

```
// readwr.cpp: Чтение и вывод чисел
#include <iostream>
#include <vector>
using namespace std;
int vector_STL()
{   vector<int> v;  int x;
cout << "Enter positive integers, followed by 0:\n";
while  (cin >> x, x != 0)  v.push_back(x);
vector<int>::iterator i;
for (i=v.begin(); i != v.end();  ++i)
cout << *i << " ";  cout << endl; return 0;
}
```

Векторы и итераторы

Шаблон *vector* используется как массив переменной длины. Сначала эта длина равна 0.

vector<int> v; – описание класса вектор v.

v.push_back(x); – добавление элемента в конец вектора.

vector<int>::iterator i; – определение итератора.

Итератор – объект, обеспечивающий доступ к содержимому контейнера.

Последовательные контейнеры – *это вектор, список, очередь и обыкновенный массив.*

Итератор используется аналогично указателю.

```
int a[N], *p;  // &a[0] и a эквивалентны
for (p=a; p != a+N; p++)  cout << *p << " ";
```

Итераторы

```
for (i = v.begin(); i != v.end(); ++i)
cout << *i << " ";
```

Для этого итератора i определены также операторы ++ и -- как в префиксном, так и в суффиксном варианте.

В приведенном цикле лучше не заменять != на <. Хотя в примере это сработает, но оператор < неприменим к некоторым другим типам, отличными от vector<int>, в то время как оператор != работает во всех случаях.

Прохождения вектора в обратном порядке:

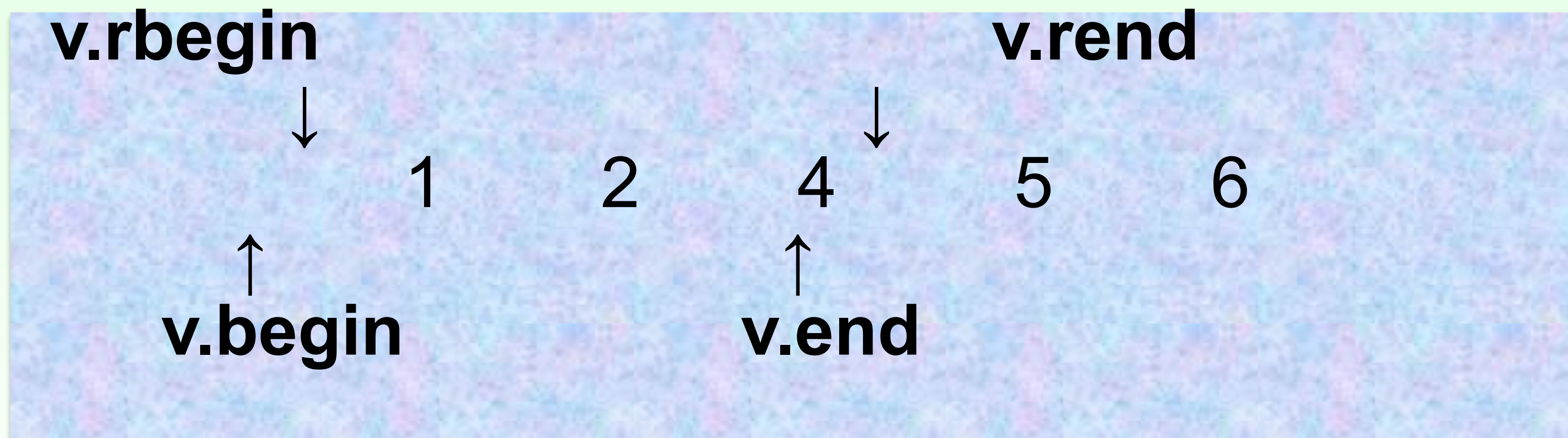
```
i = v.end();
if (i !=v.begin()) // Проверка, что 0 был единственным введённым числом,
    do cout << *--i << " ";
    while  (i ! = v.begin());
```

Обратные итераторы

Существует более простой путь прохождения вектора (и других структур данных) задом наперед. Он требует использования двух других функций-членов, *rbegin* и *rend*, вместе с обратным итератором, *reverse_iterator:*

```
vector<int>::reverse_iterator i;
for (i=v.rbegin(); i != v.rend();  ++i)
cout << *i << " ";
```

Заметьте, что в этом случае мы пишем ++i вместо --i.



Векторы, списки и двусторонние очереди

В *readwr.cpp* 3 раза встречается слово *vector.*

```
#include <vector>
...
vector<int> v;
...
vector<int>::iterator i;
```

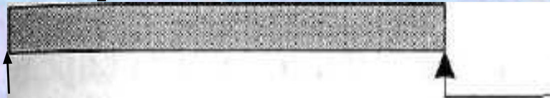
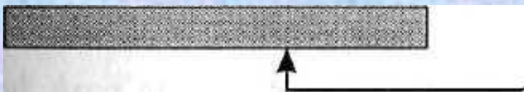
Применение концепции вектора обеспечивает выделение непрерывной памяти. В качестве альтернативы можно употребить связный список, как рекомендуется в книгах по структурам данных. *С помощью STL мы можем использовать (двойные) связные списки, не программируя их самостоятельно.* Все, что нам требуется для программы *readwr.cpp,-* заменить всюду слово *vector* на *list,* как показано в следующей программе:

Использование списка

```
// Чтение и вывод переменного количества
// ненулевых целых (ввод завершается нулем).
#include <iostream>
#include <list>
using namespace std;
{   list<int> v;
int x;
cout << "Enter positive integers, followed by 0:\n";
while  (cin >> x, x != 0)
v.push_back(x);
list<int>::iterator i;
for (i=v.begin(); i != v.end(); ++i)
cout << *i << " ";  cout << endl; return 0;
}
```

Свойства трех последовательных контейнеров

Для данного типа *T* типы *vector*<*T*>, *deque*<*T*> и *list*<*T*> называются *последовательными контейнерами.*

Vector<T>	Вставка и удаление здесь. Произвольный доступ.
Deque<T> 	Вставка и удаление здесь. Произвольный доступ.
List<T> 	Вставка и удаление в любом месте. Нет произвольного доступа.

Замечания

Программа *readwr.cpp* также будет правильно выполняться, если заменить слово *list* на *deque* (двусторонняя очередь), что дает нам третье решение. Пользователь не заметит никаких различии в поведении этих трех версий программы, но внутреннее представление данных будет различаться. Это скажется на наборе доступных операций, которые смогут выполняться эффективно.

Существует четвертая разновидность последовательного контейнера, обычный массив, который описывается как *T a[N];*

Перегрузка функций

Перегрузкой функций называется использование нескольких функций с одним и тем же именем, но с различным списком параметров. Эти функции отличаются друг от друга либо типом хотя бы одного параметра, либо количеством параметров, либо и тем и другим одновременно.

Если алгоритм не зависит от типа данных, лучше реализовать его не в виде группы перегруженных функций, а в виде шаблона функции. В этом случае компилятор сам сгенерирует текст функции для конкретного типа данных.

Краткие итоги

1. **Шаблоны функций или шаблонные классы – это инструкции, согласно которым создаются локальные версии функций и классов для определенного набора параметров и типов данных.**
2. **Шаблоны – это мощный инструмент в С++, который намного упрощает труд программиста.**
3. **Пространства имён. Строка «using namespace std;» в начале каждой программы означает импорт всего пространства имен std. Это пространство имен содержит все имена из стандартной библиотеки языка С++.**
4. **Последовательные контейнеры –** ***это вектор, список, очередь и обыкновенный массив.***

Краткие итоги

5. Итератор – объект, обеспечивающий доступ к содержимому контейнера,

лучше не заменять != на <.

```
vector<int>::iterator i;
for (i = v.begin(); i != v.end(); ++i)…..
```

6. Обратные итераторы.

```
vector<int>::reverse_iterator i;
for (i=v.rbegin(); i != v.rend();  ++i)
```