Лабораторная работа №8.

Шаблоны Классов. Стандартные шаблоны классов (STL).

Шаблоны классов. Общие положения.

Шаблоны классов позволяют создавать параметризированные классы. Параметризированный класс создает семейство родственных классов, которые можно применять к любому типу данных, передаваемых в качестве параметра шаблона. Наиболее широкое применение шаблоны получили при создании контейнерных классов.

Преимуществом шаблонов классов состоит в том, что он может применяться к любым типам данных без переопределения кода.

Синтаксис описания шаблона следующий:

```
template<class Type>  NameClass
{
        // тело шаблона
};
```

Список шаблона <class Type> состоит из служебного слова class или typename после чего идет произвольный идентификатор формального типа. Если предполагается несколько типов, то они перечисляются через запятую. Действие формального типа ограничено рамками данного шаблона.

Рассмотрим простой пример шаблона стека, предназначенного для хранения объектов различного типа.

```
const int Max = 100;
template<typename Type> class Stack
{
protected:
        int top;
        Type st[Max];
public:
        Stack()
        {
                top = -1;
        }
        void Push(Type);
        Type Pop();
        friend ostream &operator << <>(ostream &, const Stack<Type> &);
};

template<typename Type> ostream &operator <<(ostream &out, const Stack<Type> &s)
{
        out << " Element: " << s.st[s.top] << endl;
        return out;
}

template<typename Type> void Stack<Type>:: Push(Type var)
{
        st[++top] = var;
}
template<typename Type>Type Stack<Type>::Pop()
{
        return st[top--];
```

}

Здесь приведен пример шаблона класса Stack. Для простоты методы Pop и Push не снабжены проверками на пустоту и переполнение стека. Сам стек представляется как массив объектов. Обратим внимание на то, как определяются методы за пределами классов. Во-первых, все методы шаблонов класса по умолчанию считаются шаблонами функций. Во-вторых, после имени шаблона класса в обязательном порядке должен присутствовать спецификатор типа. Если это определение метода, то в качестве спецификатора выступает имя формального типа, например, Type. Если же объявление реального объекта, то спецификатором должен быть конкретный тип, в том числе и тип, определенный пользователем.

Обратите внимание на объявление дружественной функции (оператора) в теле и за пределами шаблона. В отличие от дружественной функции обычного класса, в шаблоне в объявлении должен присутствовать спецификатор шаблона, идущий сразу за именем функции: friend ostream &operator << <Type>(ostream &, const Stack<Type> &);, который в общем случае может быть пустым. При определении дружественной функции, она объявляется как шаблон, а спецификатор типа после имени не допустим. Он здесь используется для передачи типа шаблона во втором аргументе: const Stack<Type> &;.

Объявление объекта типа шаблон должно обязательно быть со спецификатором конкретного типа, например, Stack<long> st_1; или Stack<double> st_2;. Дальнейшие обращения к объектам шаблонов ничем не отличаются от обращений к объектам обычных классов.

Общие правила описания шаблонов классов:

1. локальные классы не могут содержать шаблоны в качестве своих элементов;
2. шаблоны могут содержать статические компоненты, дружественные функции и классы;
3. шаблоны могут входить в иерархию классов и шаблонов.

Следующий пример показывает использование в качестве параметра шаблона тип, определенный пользователем. Добавьте к предыдущему примеру определение простого класса и передайте его тип в качестве параметра шаблона.

```
class Test
{
        int test;
public:
        Test(){};
        Test(int t):test(t){};
        void Out()
        {
                cout << " Test: " << test << endl;
        }
        friend ostream &operator <<(ostream &, const Test &);
};
ostream &operator <<(ostream &out, const Test &t)
{
        out << t.test << endl;
        return out;
}
```

Теперь можно использовать тип Test в качестве параметра:

```
Stack<Test> st;
st.Push(100);
```

```
st.Push(200);
cout << st.Pop();
cout << st.Pop();
```

Задание №1.

Создать шаблон класса, моделирующий линейный однонаправленный список. Предусмотреть операции добавлении, удаления и поиск заданного элемента. Привести примеры хранения в данном списке объектов стандартного и пользовательского типов.

Шаблоны и наследование.

Как было сказано выше, шаблоны классов могут участвовать в иерархии классов, при этом могут выступать как базовые для шаблонов, а так же и для обычных классов и могут быть производными от шаблонов и обычных классов.

Рассмотрим пример шаблонов классов, используя принципы наследования и полиморфизма.

```
// базовый класс
template<class Type> class Base
{
        Type base;
public:
        Base(){};
        Base(Type);
        virtual void Out();
};

template <class Type> Base<Type>:: Base(Type b):base(b){};
template<class Type> void Base<Type>::Out()
{
        cout << " Template base: " << base << endl;
}
// производный класс
template<class Type_1, class Type_2> class Derived :public Base<Type_1>
{
        Type_2 derived;
public:
        Derived(){};
        Derived(Type_1, Type_2);
        void Out();
};
template<class Type_1, class Type_2> Derived<Type_1, Type_2>::Derived(Type_1 b,
Type_2 d):Base<Type_1>(b)
{
        derived = d;
}
template<class Type_1, class Type_2> void Derived<Type_1, Type_2>::Out()
{
        Base<Type_1>::Out();
        cout << " Template derived: " << derived<< endl;
}
```

Следующие выражения показывают примеры наследования и полиморфизма:

```
// объект шаблона базового класса
Base<int> bas(100);
// объект шаблона производного класса
Derived<int, double> der(200, 3.1);
// метод Out базового класса
bas.Out();
// метод Out производного класса
der.Out();
// наследование метода базового класса объектом производного класса
der.Base<int>::Out();
// указатель на шаблон базового класса
Base <int> *ptr_Base = &der;
// работа полиморфной функции производного класса
ptr_Base->Out();
```

В шаблонах можно осуществлять как повышающее преобразование, так и понижающее. Дополним предыдущую программу функцией, получающей в качестве параметра указатель на шаблон базового типа, который внутри преобразуется к указателю на шаблон производного класса:

```
void fun(Base<int> *pBase)
{
        Derived <int, double> *pDerived =
        dynamic_cast<Derived<int, double> *> (pBase);
        if(pDerived) pDerived->Out();
        else cout << " Передан объект не производного класса" << endl;
}
```

Оттранслируйте следующие выражения и проанализируйте полученные результаты.

```
Derived<int, double> *ptr_Derived = new Derived<int, double>(100, 200);
Base<int> *ptr_Base = new Base<int>();
fun(ptr_Base);
fun(ptr_Derived);
```

Не забывайте о том, что подобное преобразование допустимо при наличии полиморфных функций.

Задание №2.

Создайте шаблон базового класса « символ », позволяющий хранить одиночные символы типа char, int и производный от него шаблон класса « строка », позволяющего хранить смесь произвольных символов определенной длины (например, не более 15 символов в строке). На примере данной иерархии продемонстрировать принципы наследования и полиморфизма в шаблонах классов, а так же возможность преобразования типов в пределах родственных классов.

Задание №3.

Дополните предыдущее задание дружественными перегруженными операциями вывода в стандартный поток. Продемонстрируйте вывод одиночного символа и строки в поток. Создайте статическое поле и метод(ы) работы с ним в шаблоне базового класса, покажите возможность наследования статических компонент.

Задание №4.

В созданной иерархии замените ключевое слово class, стоящее перед именем базового и производного шаблона классов на слово struct или union. Проанализируйте возможные сообщения компилятора и объясните их.

Стандартные шаблоны классов (STL).

Стандартные шаблоны классов являются мощным инструментом создания программного обеспечения в различных прикладных областях. Библиотек шаблонов классов входит в стандарт языка. Примечательно то, что многие алгоритмы, разработанные в данной библиотеке, прекрасно работают с обычными массивами.

STL состоит из следующих частей: контейнер, алгоритм и итератор.

Контейнер – это способ организации хранения данных, например, стек, связный список, очередь. Массив языка C++ в какой-то степени относится к контейнерам. Контейнеры представлены в библиотеке как шаблонные классы, что позволяет быстро менять тип хранимых в них данных.

Алгоритм – это процедуры, применяемые к контейнерам для обработки хранящихся данных. Они представлены как шаблоны функций. Однако эти функции не принадлежат какому-либо классу контейнеру, они не являются дружественными функциями. Наоборот, это совершенно независимые функции. Это обеспечивает универсальность алгоритмов и позволяет использовать алгоритмы не только к контейнерам, но и к другим типам данных.

Итераторы – обобщенная концепция указателей, они ссылаются на элементы контейнеров. Итераторы – ключевая часть STL, поскольку они связывают алгоритмы с контейнерами. Иногда итераторы называют интерфейсом библиотеки шаблонов.

Контейнеры.

Контейнеры представляют собой хранилища данных. При этом не имеет значения, какого типа данные в них хранятся. Это могут быть базовые типы int, float, double, etc, а также типы, объявленные пользователем. Контейнеры STL делятся на две основные категории: последовательные контейнеры и ассоциативные. К последовательным контейнерам относятся векторы, списки и очереди с двусторонним доступом. К ассоциативным контейнерам – множества, мультимножества, отображения и мультиотображения. Кроме того, наследниками последовательных контейнеров выступают стек, очередь и приоритетная очередь.

Методы.

Для выполнения простых операций над контейнерами используют методы. Методы проще алгоритмов, применять их можно к большей части классов контейнеров. Некоторые методы перечислены ниже.

size() – возвращает число элементов контейнера;
empty() – возвращает true, если контейнер пуст;
max_size() – возвращает максимально допустимый размер контейнера;
begin() – возвращает итератор на начало контейнера;
end() – возвращает итератор на последнюю позицию в контейнере;
rbegin() – возвращает реверсивный итератор на конец контейнера;
rend() – возвращает реверсивный итератор на начало.

Алгоритмы.

Алгоритм – это функция, которая производит некоторое действие над элементами контейнера. В стандарте языка C++ алгоритм независимая шаблонная функция, применимая даже к обычным массивам. Рассмотрим отдельные методы на примерах.

Алгоритм find().

Этот алгоритм ищет первый элемент в контейнере, значение которого равно указанному. Общий формат метода следующий: find(beg, end, val), где beg – итератор (указатель) первого значения, которое нужно проверить; end – итератор последней позиции; val – искомое значение.

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

int arr[] = {11, 22, 33, 44, 55, 66, 77, 88, 99, 0};

int main()
{
	setlocale(0,"RUS");
	int *ptr;
	ptr = find(arr, arr+10, 55);
	cout << " Первый объект со значением 55 найден в позиции: " << (ptr-arr) << endl;
	system("pause");
	return 0;
}
```

Для использования алгоритмов необходимо подключить файл algorithm. В рассмотренном примере в качестве контейнера выступает обычный массив, в котором ищется число 55. Начало поиска – элемент с нулевым индексом, окончание – указатель на единицу больший, чем указатель на последний элемент. Подобный подход носит название «значение после последнего». Результатом работы программы будет число 4. Если в контейнере нет искомого значения, программа выдаст указатель на следующий элемент после последнего. Для рассмотренного примера это будет число 10.

Следующий фрагмент демонстрирует пример использования метода find для контейнера типа vector.

```
vector<int> v;
	for(int i = 1; i <= 100; i++)
	{
		v.push_back(i*i);
	}

	if(find(v.begin(), v.end(), 49) != v.end())
	{
		cout <<  " Число 49 найдено в списке квадратов натуральных чисел, не
		превосходящих 100 " << endl;
	}
	else
	{
		cout <<  "  число 49 не найдено "
	}
```

Диапазон поиска определяется с помощью методов begin и end.

Задание №5.

Продемонстрировать работу алгоритма find() для последовательных контейнеров списки (list), очереди с двусторонним доступом (deque) и стек (stack). В программе необходимо подключить соответствующие заголовочные файлы.

Алгоритм count().

Этот алгоритм подсчитывает, сколько элементов в контейнере имеют данное значение. Его формат подобен формату алгоритма find. Рассмотрим фрагмент программного кода, использующего алгоритм count().

```
int arr[] = {11, 22, 33, 22, 55, 22, 77, 22, 99, 0};

int main()
{
	setlocale(0,"RUS");
	int n = count(arr, arr+10, 22);
	cout << " Число 22 встречается " << n << " раз(а) " << endl;

	system("pause");
	return 0;
}
```

Результатом алгоритма count() будет число вхождений искомого элемента или число 0, если искомый элемент не входит в данный контейнер.

Задание №6.

Продемонстрировать работу алгоритма count() для последовательных контейнеров списки (list), очереди с двусторонним доступом (deque) и стек (stack).

Алгоритм search().

Алгоритм search() оперирует одновременно с двумя контейнерами. Он отыскивает некоторую последовательность, входящую в один контейнер в другом контейнере. Формат алгоритма следующий: search(beg, end, sbeg, send), где beg – начало первого контейнера, end – конец первого контейнера, sbeg – начало второго контейнера, send – конец второго контейнера. Результатом работы алгоритма будет позиция, начиная с которой обнаружено совпадение. Если совпадения не обнаружено, необходимо обеспечить вывод соответствующего сообщения. Следующий фрагмент показывает пример работы алгоритма.

```
int arr1[] = {11, 24, 33, 11, 22, 33, 22, 66, 77, 8};
int arr2[] = {11, 22, 33};
int main()
{
	setlocale(0,"RUS");
	int *ptr;
	ptr = search(arr1, arr1+10, arr2, arr2+3);
	if(ptr == arr1+10)  cout << " Совпадений нет ";
	else cout << " Совпадения в позиции " << (ptr-arr1) << endl;
	system("pause");
	return 0;
}
```

Несложно заметить, что результатом данного примера будет число 3, так как совпадение двух массивов начинается с элемента со значением 11 первого массива, имеющего индекс 3.

Важно заметить, что параметрами алгоритма search() не обязательно должны быть контейнеры одного и того же типа. Второй параметр в этом случае играет роль маски. В этом состоит универсальность библиотеки STL.

Задание №7.*

Объявить классы Person и Student, имеющие поля Name, Age, идентифицирующие имя и возраст человека и студента. В каждом из классов предусмотреть поля, характерные для его типов, например для класса Student таким полем может быть Ball, а для класса Person – поле Prof, идентифицирующее профессию человека. Объявить два контейнера типа vector, в одном из которых находятся объекты типа Person, а во втором – объекты типа Student. Используя алгоритм search(), определить последовательность значений, заданных контейнером Student, попадающих в контейнер Person. Сравнение проводить по значениям полей Name. Показать пример работы алгоритма count() на заданных контейнерах, в качестве критерия поиска можно использовать поля Ball типа Student, подсчитывающего число студентов, сдавших последнюю сессию на 4 и на 5. Для класса Person критерием выбора может быть величина оклада.

Алгоритм sort().

Этот алгоритм сортирует элементы контейнера в порядке их увеличения. Следующий пример показывает возможность использования алгоритма сортировки для целочисленных контейнеров, например, для последовательности символов.

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

typedef unsigned int UINT;
vector<char> arr_char;

int main()
{
	arr_char.push_back('T');
	arr_char.push_back('E');
	arr_char.push_back('G');
	arr_char.push_back('L');
	arr_char.push_back('A');
	arr_char.push_back('Z');
	arr_char.push_back('K');
	arr_char.push_back('W');
	sort(arr_char.begin(), arr_char.end());
	for(UINT j=0; j < arr_char.size(); j++)
		cout << arr_char[j] << ' ';
	cout << endl;
	setlocale(0,"RUS");
	system("pause");
	return 0;
}
```

Алгоритм sort() имеет два параметра, идентифицирующие начало и конец сортируемого контейнера. В данном примере начало, и конец последовательности получаются как результат работы соответствующих методов.

Алгоритм merge().

Этот алгоритм работает с тремя контейнерами, объединяя два из них в третий. Рассмотрим пример использования этого алгоритма.

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

int arr1[] = {2, 3, 4, 6, 8};
int arr2[] = {1, 3, 5};
int dest[8];

int main()
{
	setlocale(0,"RUS");
	merge(arr1, arr1+5, arr2, arr2+3, dest);
	for(int i=0; i<8; i++)
		cout << dest[i] << ' ';
	   cout << endl;
	system("pause");
	return 0;
}
```

Задание №8.

Используя алгоритм merge() объединить два вектора и списка в результирующие контейнеры.

Функциональные объекты.

Многие алгоритмы для своей работы требуют функциональные объекты. Функциональным объектом считается объект шаблонного класса, в котором есть единственный метод – перегруженная операция вызова функции – (). Рассмотрим пример использования алгоритма sort с функциональным объектом, позволяющим отсортировать последовательность в порядке убывания.

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

double ddata[] = {38.44, 64.22, -9.345, 0.453, 8.01, 23};

int main()
{
	setlocale(0,"RUS");

	sort(ddata, ddata+6, greater<double>() );
	for(int j=0; j<6; j++)
		cout << ddata[j] << ' ';
```

```
	cout << endl;
	system("pause");
	return 0;
}
```

Функциональным объектом здесь является метод greater<double>(), который сортирует последовательность по убыванию. Обычно алгоритм sort() сортирует по возрастанию.

Функциональные объекты могут успешно работать только со стандартными типами данных C++ или с классами, в которых определены (перегружены) операторы <, <=, >, >= , ==, etc. Проблему можно обойти, определив собственную функцию для функционального объекта. Следующий пример показывает это.

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

char *Names[] = { "Петя", "Маша", "Дима", "Вова", "Вася", "Филимон"};

bool alpha(char *n1, char *n2)
{
	return(strcmp(n1,n2)<0) ? true : false;
}

int main()
{
	setlocale(0,"RUS");
	sort(Names, Names+6, &alpha);
	for(int i=0; i<6; i++)
		cout << Names[i] << ' ';
	cout << endl;
	system("pause");
	return 0;
}
```

Здесь вместо функционального объекта выступает функция bool alpha(), которая сравнивает две строки, используя стандартную функцию strcmp(). Наличие оператора взятия адреса перед именем функции alpha() не обязательно. Это скорее дань традициям языка C.

Задание №9.

Используя собственные функции как функциональные объекты, показать пример их применения к спискам, очередям с двусторонним доступом.

Задание №10.

Самостоятельно изучить последовательные контейнеры: векторы, списки и очереди с двусторонним доступом, а также алгоритмы и методы для работы с ними. Показать примеры их использования.

Итераторы.

Для обращения к элементам обычного массива используют операцию индексирования [] или обращение через указатель. Использовать указатели к более сложным контейнерам затруднительно. Во-первых, элементы контейнера могут храниться

в памяти не последовательно, а сегментировано. Методы доступа к ним существенно усложняются. Нельзя просто инкрементировать указатель для получения следующего значения. Во-вторых, при добавлении или удаления элементов из середины контейнера, указатель может получить некорректное значение.

Одним из решений проблемы является создание класса «умных» указателей – итераторов. Объект такого класса является оболочкой для методов, оперирующих с обычными указателями. Для них перегружены операции ++ и *, адекватно работающих даже в том случае, когда элементы сегментированы в памяти.

Итераторы играют важную роль в STL. Они определяют какие алгоритмы использовать к каким контейнерам. По категориям различают следующие виды итераторов: входные, выходные, прямые, двунаправленные и случайного доступа.

Работа с итераторами.

Доступ к данным. Вставка данных.

В контейнерах с итераторами произвольного доступа (векторах, очередях с двусторонним доступом), итерация осуществляется с помощью оператора []. Однако к спискам эту операцию применить нельзя. Рассмотрим пример работы со списками через итераторы.

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

int main()
{
	// доступ к данным
	double  arr[] = {2.33, 44.32, -6.54, .8};
	list<double> doubleList;
	for(int i=0; i<4; i++)
		doubleList.push_back(arr[i]);
	list<double> ::iterator iter = doubleList.begin();
	for(iter=doubleList.begin(); iter!=doubleList.end(); iter++)
		cout << *it++ << ' ';
		cout << endl;

	// вставка данных
	list<int> intList(5);
	list<int> ::iterator it;
	int data = 0;
	for(it=intList.begin(); it!=intList.end(); it++)
		 *it = data += 2;
	it = intList.begin();
	while(it !=intList.end())
		cout << *it++ << ' ';
		cout << endl;
	system("pause");
	return 0;
}
```

Объявление итераторов в данной программе:

- оператор list<double> ::iterator iter = doubleList.begin(); - определяет двунаправленный итератор для работы со списком, содержащего вещественные числа и инициализирует его указателем на начало списка;

- оператор list<int> ::iterator it; - определяет двунаправленный итератор для списка с целочисленными элементами.

Итераторы для вектора или очереди автоматически делаются с произвольным доступом. Имена итераторов – обычные идентификаторы, подчиняющиеся законам языка. Еще одно важное замечание: обращение к итераторам, например, *it = data += 2; или cout << *it++ << ' '; осуществляется как к обычным указателям.

Задание №11.

Определить вектор, очередь содержащие элементы стандартного типа (целого, вещественного и т. д.), определить для них итераторы и опробовать алгоритмы и методы работы с ними. Определить класс, состоящий из небольшого числа компонент, сохранить объекты этого класса в очереди. Показать пример использования итераторов при работе с очередью объектов собственного типа.

Специализированные итераторы.

Под специализированными итераторами понимают адаптеры итераторов и потоковые итераторы.

Адаптеры итераторов бывают трех видов: обратные (реверсивные) итераторы, итераторы вставки и итераторы неинициализированного хранения.

Обратные итераторы.

Для осуществления реверсивного прохода по контейнеру используются обратные итераторы. Следующий пример показывает определение и применение реверсивного итератора для вывода списка в обратном порядке.

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

int main()
{
	int arr[] = {2, 4, 6, 8, 10, 12};
	list<int> intList;

	for(int i=0; i<6; i++)
		intList.push_back(arr[i]);

	list<int>::reverse_iterator revit;
	revit = intList.rbegin();
	while(revit != intList.rend())
		cout << *revit++ << ' ';
	cout << endl;
	system("pause");

	return 0;
};
```

Обратный проход списка в данном примере здесь обеспечивает объявление реверсивного итератора list<int>::reverse_iterator revit;. При его использовании следует использовать методы rbegin() и rend(). С обычными итераторами их использовать нельзя.

Задание №12.

Определить список, содержащий элементы собственного типа (например, из предыдущего примера), объявить обратный итератор и обеспечить вывод списка в обратном порядке.