top
КОМПЬЮТЕРНАЯ АКАДЕМИЯ

ОБЪЕКТНО -ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА C++

Урок №14

Введение в STL

Содержание

Преобразование типов в стиле C++

Для начала напомним, что оператор приведения типов позволяет компилятору преобразовывать один тип данных в другой. И язык программирования C, и язык программирования C++ поддерживают такую форму записи приведения типа:

```
(тип) выражение

Например,

double d;
d=(double) 10/3;
```

Операторы приведения типа в языке C++

Кроме стандартной формы в стиле C, язык C++ поддерживает дополнительные операторы приведения типа:

const_cast

```
имя_объекта.имя_члена класса;
```

dynamic_cast

```
имя_объекта.имя_члена класса;
```

reinterpret_cast

```
имя_объекта.имя_члена класса;
```

static_cast

```
имя_объекта.имя_члена класса;
```

Рассмотрим данные конструкции более детально:

- **const_cast** используется для явного переопределения модификаторов const и/или volatile. Новый тип должен совпадать с исходным типом, за исключением изменения его атрибутов const или volatile. Чаще всего оператор const_cast используется для снятия атрибута const. Например:

  Примечание: Следует помнить, что только оператор const_cast может освободить от «обета постоянства», т.е. ни один из остальных операторов этой группы не может «снять» с объекта атрибут const.

```
#include <iostream>
using namespace std;
//указатель на объект является константным,
//следовательно, через него изменить значение
//объекта нельзя

void test_pow(const int* v){
    int*temp;
    //снимаем модификатор const
    //и теперь можем изменять объект
    temp=const_cast<int*>(v);
    //изменение объекта
    *temp= *v * *v;
}

void main(){
    int x=10;
```

```
//на экране — 10
    cout<<"Before — "<<x<<"\n\n";
    test_pow(&x);
    //на экране — 100
    cout<<"After — "<<x<<"\n\n";
}
```

Примечание: Кстати, стоит упомянуть о том, что такое **volatile**!!! Данный модификатор даёт компилятору понять, что значение переменной, может быть изменено неявным образом. Например, есть некая глобальная переменная, её адрес передаётся встроенному таймеру операционной системы. В дальнейшем эта конструкция может использоваться для отсчёта реального времени. Естественно, что значение этой переменной изменяется автоматически без участия операторов присваивания. Так вот такая переменная должна быть объявлена с использованием ключевого слова volatile. Это связано с тем, что многие компиляторы не обнаружив ни одного изменения переменной с помощью какого-либо присваивания, считают, что она не изменяется и оптимизируют выражения с ней, подставляя на её место конкретное значение. При этом компилятор переменную не перепроверяет. Действительно, зачем, если переменная для него имеет характер «неизменяемой»?! volatile — будет блокировать подобные действия компиляторов.

- **dynamic_cast** проверяет законность выполнения заданной операции приведения типа. Если такую операцию выполнить нельзя, то выражение устанавливается равным нулю. Этот оператор в основном используется

для полиморфных типов. Например, если даны два полиморфных класса, B и D, причем класс D выведен из класса B, то оператор dynamic_cast всегда может преобразовать указатель D* в указатель B*. Оператор dynamic_cast может преобразовать указатель B* в указатель D* только в том случае, если адресуемым объектом является объект D. И вообще, оператор dynamic_cast будет успешно выполнен только при условии, что разрешено полиморфное приведение типов (т.е. если новый тип можно законно применять к типу объекта, который подвергается этой операции). Рассмотрим пример, демонстрирующий всё вышесказанное:

```
#include <iostream>
using namespace std;
//базовый класс
class B{
    public:
    //виртуальная функция для
    //последующего переопределения в потомке
    virtual void Test(){
        cout<<"Test B\n\n";
    }
};

//класс-потомок
class D:public B{
    public:
    //переопределение виртуальной функции
    void Test(){
        cout<<"Test D\n\n";
    }
};
```

```
void main(){
    //указатель на класс-родитель
    //объект класса-родителя
    B *ptr_b, obj_b;
    //указатель на класс-потомок
    //и объект класса-потомка
    D *ptr_d, obj_d;

    //приводим адрес объекта (D*) к указателю типа D*
    ptr_d= dynamic_cast<D*> (&obj_d);
    //если все прошло успешно — вернулся !0
    //произошло приведение
    if(ptr_d){
        cout<<"Good work — ";
        //здесь вызов функции класса-потомка
        //на экране — Test D
        ptr_d->Test();
    }

    //если произошла ошибка — вернулся 0
    else cout<<"Error work!!!\n\n";

    //приводим адрес объекта (D*) к указателю типа B*
    ptr_b= dynamic_cast<B*> (&obj_d);
    //если все прошло успешно — вернулся !0
    //произошло приведение
    if(ptr_b){
        cout<<"Good work — ";
        //здесь вызов функции класса-потомка
        //на экране — Test D
        ptr_b->Test();
}

    //если произошла ошибка — вернулся 0
    else cout<<"Error work!!!\n\n";
```

```
//приводим адрес объекта (D*) к указателю типа B*
    ptr_b= dynamic_cast<B*> (&obj_d);
    //если все прошло успешно – вернулся !0
    //произошло приведение
    if(ptr_b){
        cout<<"Good work – ";
        //здесь вызов функции класса-потомка
        //на экране – Test D
        ptr_b->Test();
    }
    //если произошла ошибка – вернулся 0
    else cout<<"Error work!!!\n\n";

    //приводим адрес объекта (B*) к указателю типа B*
    ptr_b= dynamic_cast<B*>(&obj_b);
    //если все прошло успешно – вернулся !0
    //произошло приведение
    if(ptr_b){
        cout<<"Good work – ";
        //здесь вызов функции класса-потомка
        //на экране – Test B
        ptr_b->Test();
    }
    //если произошла ошибка – вернулся 0
    else cout<<"Error work!!!\n\n";

    //ВНИМАНИЕ!!! ЭТО НЕВОЗМОЖНО
    //попытка привести адрес объекта (B*)
    //к указателю типа D*
    ptr_d= dynamic_cast<D*> (&obj_b);
    //если все прошло успешно – вернулся !0
    //произошло приведение
    if(ptr_d)
        cout<<"Good work – ";
    //если произошла ошибка – вернулся 0
    else cout<<"Error work!!!\n\n";
}
```

```
Результат работы программы:
Good work – Test D
Good work – Test D
Good work – Test B
Error work!!!
```

- **static_cast** выполняет неполиморфное приведение типов. Его можно использовать для любого стандартного преобразования. При этом никакие проверки во время работы программы не выполняются. Фактически, static_cast — это аналог стандартной операции преобразования в стиле C. Например, так:

```
#include <iostream>
using namespace std;
void main(){
    int i;
    for(i=0;i<10;i++)
        //приведение переменной i к типу double
        //результаты деления на экране, естественно
        //вещественные
        cout<<static_cast<double>(i)/3<<"\t";
}
```

- **reinterpret_cast** переводит один тип в совершенно другой. Например, его можно использовать для перевода указателя в целый тип и наоборот. Оператор reinterpret_cast следует использовать для перевода типов указателей, которые несовместимы по своей природе. Рассмотрим пример:

```
#include <iostream>
using namespace std;
void main(){
```

```
    //целочисленная переменная
    int x;
    //строка (указатель типа char)
    char*str="This is string!!!";
    //демонстрируем строку на экран
    cout<<str<<"\n\n"; // на экране – This is string!!!
    //преобразуем указатель типа char в число
    x=reinterpret_cast<int>(str);
    //демонстрируем результат
    cout<<x<<"\n\n"; // на экране – 4286208
}
```

Итак, мы познакомились с преобразованиями в стиле C++. Вполне очевидны положительные стороны этих новых способов. Во-первых данные преобразования позволяют сделать то, чего стандарт языка C даже не предполагал: снять модификатор const, или радикально поменять тип данных. Во-вторых, стиль C++ увеличивает контроль над преобразованиями и позволяет выявить ошибки с ними связанные на ранних этапах.

Стандартная библиотека шаблонов (STL)

Основные понятия

(контейнер, итератор, алгоритм, функтор, предикат, аллокатор)

На сегодняшний день, почти все компиляторы языка C++ содержат специальную встроенную библиотеку STL. Данная библиотека содержит набор классов и функций, которые представляют собой реализацию часто используемых алгоритмов. А поскольку, библиотека предназначена для работы с различными типами данных, все классы и функции в ней являются шаблонными. Детальным рассмотрением этого замечательного средства мы и займемся в ближайшее время.

Для того чтобы разобраться с библиотекой STL, нам необходимо познакомиться с основными понятиями, на которых она базируется. Итак. Приступим.

Наша библиотека включает в себя четыре компонента, составляющие её внутреннюю структуру:

- **Контейнер** — блок для хранения данных, управления ими и размещения. Иными словами, это объект, предназначенный для хранения и использования других элементов.
- **Алгоритм** — специальная функция для работы с данными, содержащимися в контейнере.

- **Итератор** — специальный указатель, который позволяет алгоритмам перемещаться по данным конкретного контейнера.
- **Функторы** — механизм для инкапсуляции функций в конкретном объекте для использования его другими компонентами.

Кроме вышеописанных конструкций, в STL поддерживаются еще некоторые встроенные компоненты:

- **Аллокатор** — распределитель памяти. Такой распределитель памяти имеется у каждого конкретного контейнера. Данная конструкция просто-напросто управляет процессом выделения памяти для контейнера. Следует отметить, что по умолчанию распределителем памяти является объект класса allocator. Однако, можно определить свой собственный распределитель.
- **Предикат** — функция нестандартного типа, используемая в контейнере. Предикат бывает унарный и бинарный. Может возвращать логическое значение (истину либо ложь).

В данной части урока мы рассмотрели лишь основные определения, касаемые STL. Далее, мы будем анализировать каждый из этих механизмов отдельно.

Класс auto_ptr

В языке программирования C++ есть понятие, которое проявляется в следующем утверждении — «***выделение ресурса — это инициализация***». Данное утверждение идеально используется при выделении абсолютно любого типа ресурсов. Это делает работу программы более надежной, особенно, если в ходе работы программы возможно возникновение исключений.

Например, программа открывает некий файл или выделяет фрагмент памяти. При завершении работы этой программы естественно необходимо закрыть файл или освободить память. Однако исключительная ситуация может проявиться еще до того, как будет выполнено одно из вышеописанных действий, в результате чего действие так и не выполнится.

```
void f(){
    FILE*f;

    if (!(f = fopen("test.txt", "rt")))
    {
        //не удалось открыть файл — выходим
        exit(0);
    }
    //удалось, работаем с файлом дальше
    //но, здесь может возникнуть исключение
    //и до следующей строки мы не доберемся
    //соответственно файл не будет закрыт
    fclose(f);
}
```

Решением данной проблемы является следующий набор действий:

1. Создать класс.
2. Реализовать конструктор, в котором будет открываться файл.
3. Реализовать деструктор, в котором файл будет закрываться.
4. Для начала работы с файлом необходимо создать объект этого класса. (Файл будет автоматически открыт в конструкторе).

А, теперь главное.

В случае возникновения исключительной ситуации при работе программы объект будет естественно уничтожен, но при этом для него будет вызван его деструктор, который и закроет файл (освободив память). Если же, программа завершится корректно, то и в этом случае вам не придется думать о явном закрытии файла, так как созданный объект будет автоматически уничтожен при выходе из области видимости и файл будет закрыт опять таки деструктором класса.

```
class FileOpen
{
    FILE* f;

public:

    FileOpen(char* filename, char* mode)
    {
        if (!(f = fopen(filename, mode)))
        {
```

```
            exit(0);
        }
    }
    ~FileOpen(){
        fclose(f);
    }
};

void f(){
    FileOpen MyFile("test.txt", "r+");
    //здесь выполняем нужную работу с файлом
}
```

Для усовершенствования нашего принципа мы можем использовать класс auto_ptr (automatic pointer — автоматический указатель). Данный класс предоставляется стандартной библиотекой C++ и предназначен для работы с объектами, которые обычно необходимо удалять явно (например, объекты, созданные динамически с помощью оператора new).

Для создания объекта класса auto_ptr параметром конструктора должен быть указатель на объект, созданный динамически. Дальше с auto_ptr можно работать почти как с обычным указателем, который указывает на тот же динамический объект, на который указывал исходный указатель. Нам не нужно думать о явном удалении объекта, он будет автоматически удален деструктором класса auto_ptr.

Синтаксис класса auto_ptr в стандартной библиотеке выглядит так:

```
template<class X>
class Std::auto_ptr
{
```

```
X* ptr;
public:

    //конструктор и деструктор
    explicit auto_ptr(X* p = 0)throw()
    {
        ptr = p;
    }

    ~auto_ptr() throw()
    {
        delete ptr;
    }

    //оператор разыменования позволяет получить объект
    X& operator*()const throw()
    {
        return *ptr;
    }
        //оператор -> позволяет получить указатель

    X* operator->()const throw()
    {
        return ptr;
    }
};
```

Примечание: Кроме переопределения операторов -> и *, класс auto_ptr содержит две функции-члена: X* get () const; — возвращает указатель на объект класса, скрывающегося под шаблоном X.
X* release () const; — возвращает указатель на объект класса, скрывающегося под шаблоном X, но при этом забирает у auto_ptr права на этот объект. ВНИМАНИЕ!!! Сам объект не уничтожается!!!

Теперь, вполне естественно, рассмотрев общую конструкцию auto_ptr, полюбопытствовать, как он выглядит в работе:

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

class TEMP{
    public:
    TEMP(){
        cout<<"TEMP\n\n";
    }
    ~TEMP(){
        cout<<"~TEMP\n\n";
    }
    void TEST(){
        cout<<"TEST\n\n";
    }
};

void main(){
    //создаём два автоматических указателя
    //под один из них выделяем память типа TEMP
    auto_ptr<TEMP>ptr1(new TEMP), ptr2;

    //передача права владения
    ptr2=ptr1;

    //вызов функции через автоматический указатель
    ptr2->TEST();

    //присваивание автоматического указателя
    //обычному указателю на объект класса
    TEMP*ptr=ptr2.get();

    //вызов функции через обычный указатель
    ptr->TEST();
}
```

Анализ и использование класса string

Ну а теперь мы приступим к детальному рассмотрению классов библиотеки STL. Для начала обратим внимания на класс string или строка. Класс предоставляет методы для работы с символами и представляет собой массив символов с поддержкой строк любой длины.

Список наиболее часто используемых функций класса string.

- **operator[]** — доступ к конкретным символам в строке для чтения или записи.
- **c_str()** — конвертация строки в const char* для использования в функциях, не умеющих работать со string.
- **append** — добавление символов к концу строки.
- **operator=** — присваивание строке других строк, символьных массивов и даже чисел.
- **insert** — вставка символов или других строк в переменную типа string.
- **erase** — удаление одного или более символов из заданной строки в заданной позиции.
- **replace** — замещение одного или более символов в заданной позиции.
- **length**(или **size**) — возвращение количества символов в строке.

- **empty** — определение, есть ли в строке символы.
- **find** — нахождение первого вхождения символа или подстроки в данной строке.
- **rfind** — аналог find, но осуществляет поиск с конца строки назад.
- **find_first_of** — нахождение первого вхождения символов из набора в строке.
- **substr** — возвращение подстроки.
- **find_first_not_of** — нахождение первого символа в строке, не входящего в заданный набор.
- **compare** — сравнение строки (также поддерживаются операторы !=, <, >)

Класс string — является одним из самых используемых классов. Давайте разберем пример работы с ним:

```
#include <stdio.h>
#include <stdlib.h>
#include <string>
using namespace std;

void main()
{
    //Присвоить строку символов объекту типа string
    string s = "Hello world";

    // Получить первое слово в строке
    int nWordEnd = s.find(' ');
    string sub_string = s.substr(0,nWordEnd);

    // Вывести результаты
    printf("String: %s\n", s.c_str());
    printf("Sub String: %s\n", sub_string.c_str());
}
```

Комментарии к примеру

- Вы можете прямо присваивать строку символов объекту string. Это упрощает использование строк с данными в программе.
- Метод find найдет первое вхождение символа в строке. Возвращает метод позицию (начиная с 0) найденного символа, или –1, если подходящего символа в строке не нашлось.
- Метод substr возвращает копию части строки, начинающейся с позиции, заданной первым параметром метода, и длиной, заданной во втором параметре метода. Если опустить второй аргумент, то возвратится строка символов начиная с данной позиции до конца исходной строки.
- Для вывода строки на экран используется функция printf, выводя строку как символьный массив. Используем метод c_str() для преобразования объекта string в символьный массив.

Детальный анализ итератора

Итак, пришла пора познакомиться поближе.

Итератор — это своеобразный указатель общего назначения.

На самом деле простые указатели являются частным случаем итераторов, позволяющих работать с различными данными универсальным способом. Любая функция, принимая в качестве параметров итераторы, при их обработке не задумывается о типе переменных, на которые эти итераторы указывают. Итераторы позволяют получить доступ к содержимому контейнера так же, как указатели используются для доступа к элементам массива. Для использования итераторов необходимо подключить библиотеку <iterator>.

Рассмотрим типы итераторов и назначение каждого из них:

Типы итераторов

- ***Входные*** (*input*) — служат для чтения адресуемых данных. Поддерживают операции равенства, разыменования и инкремента. ==, !=, *i, ++i, i++, *i++.
- ***Выходные*** (*output*) — адресуют объекты, в которые данные должны быть записаны. Поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента. ++i, i++, *i = t, *i++ = t.

- ***Однонаправленные*** (*forward*) — обладают всеми свойствами входных и выходных, а также могут перемещаться от начала последовательности адресуемых данных в конец. Поддерживают все операции итераторов ввода/вывода и, кроме того, позволяют без ограничения применять присваивание. ==, !=, =, *i, ++i, i++, *i++.
- ***Двунаправленные*** (*bidirectional*) — обладают свойствами однонаправленных, но и способны перемещаться в любом направлении по цепочке данных: как вперед, так и назад. Имеют дополнительную операцию декремента (--i, i--, *i--).
- ***Итераторы произвольного доступа*** (*random access*) — обладают функциональностью всех четырех других итераторов. Поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ по индексу. i += n, i + n, i –= n, i — n, i1 — i2, i[n], i1 < i2, i1 <= i2, i1 > i2, i1 >= i2.

Примечание: Кстати, указатели языка C++ также являются итераторами произвольного доступа.

Хотим обратить Ваше внимание, что вышеописанная классификация не является строгой. Как правило, итераторы, которые обладают более расширенными свойствами, можно легко применять вместо итераторов более узкой направленности. Например, вместо итератора ввода может быть использован прямой итератор.

Кроме того, библиотека STL содержит так называемые обратные итераторы (*reverse iterator*). Такие итераторы

бывают либо двунаправленными, либо представляются итераторами произвольного доступа. В любом проявлении, они должны обладать свойством перемещения по контейнеру в обратном направлении. То есть, если подобный итератор указывает на хвост некой последовательности, то его увеличение приведет к позиционированию на предыдущий элемент.

И, наконец, есть еще два типа итераторов, которые не вошли в основную классификацию, а именно:

Потоковые итераторы — итераторы, позволяющие перемещаться по потоку.

Итераторы вставки — итераторы, упрощающие вставку элементов в определённый контейнер.

Из всего вышесказанного, вполне очевидно, что мир итераторов достаточно широк. В процессе изучения STL, мы с Вами будем знакомиться с итераторами всех классификаций, для совершенно разных стандартных классов. Однако, у всех итераторов достаточно много общего, и дело не только в их предназначении, но и в работе с ними. Рассмотрим две функции, которые используются для работы с итераторами:

advance() и distance()

```
template <class InIter, class Dist>
    void advance(InIter &itr, Dist d);
template <class InIter> ptrdiff_t
                distance(InIter &start, InIter end);
```

Примечание: ptrdiff_t — тип данных, который позволяет представить разность между двумя указателями.

Функция advance() увеличивает итератор itr на величину d. Функция distance() возвращает количество элементов, расположенных между итераторами *start* и *end*.

Эти две функции необходимы, поскольку только итераторы произвольного доступа позволяют добавлять к итераторам и вычитать из них некую величину. Функции advance() и distance() позволяют обойти эти ограничения. Однако следует отметить, что некоторые итераторы не обеспечивают эффективную реализацию этих функций.

Итак, мы рассмотрели еще один неотъемлемый элемент библиотеки STL. Подведём итог. Итераторы — это механизм, позволяющий осуществлять доступ к отдельным элементам безразмерных конструкций. И в роли этих конструкций выступают, изучаемые нами сейчас и в дальнейшем — классы-контейнеры.

Домашнее задание

1. Написать программу, которая использует класс string для анализа строки, содержащей математическое выражение, например, вида — (2+3)*4+1. Строка вводится с клавиатуры. Программа выдает результат вычисления выражения.