Essential C++

> DOA于2019.3

第一章 C++编程基础

C++中，初始化各种变量可以用括号（面向对象的风格）

例如：

```
int test(0);    //将test初始化为0
```

array(这里指普通数组，而不是C++11中的array类型)和vector的区别

文件读写

有三种文件流类：

- ofstream：文件输出流类，用于向文件写入，默认会覆盖原文件内容，如需以增添模式读取，则需增添ios_base::app参数
- ifstream： 文件读入流类，用于读取文件
- fstream：文件流类，既可输出又可读取，需要指定参数ios_base::in|ios_base::app

第二章 面向过程的编程风格

值传递、引用传递与地址传递

一般来说用引用传递用得比较多、因为引用传递不用在函数中在新建一个变量并copy值过去，直接指向原来变量的地址，效率高，速度快，但是要注意它会改变原来的值，若要繁殖这种情况，则在参数列表前加一个const就行。

> 地址传递与引用传递其实差不多，但是引用传递的用法比地址传递简单

函数参数默认值

- 从左往右数第一个有默认值的参数右侧的参数都必须要有默认值
- 为了更高的可见性，默认值最好定在函数声明处而非定义处

局部静态变量

合理利用可以减少运算次数

vector的push_back、pop_back函数

inline声明

在函数前加一个inline声明使该函数成为内联函数。

> 内联函数是C++的增强特性之一，用来降低程序的运行时间。当内联函数收到编译器的指示时，即可发生内联：编译器将使用函数的定义体来替代函数调用语句，这种替代行为发生在编译阶段而非程序运行阶段。
>
> 值得注意的是，内联函数仅仅是对编译器的内联建议，编译器是否觉得采取你的建议取决于函数是否符合内联的有利条件。如何函数体非常大，那么编译器将忽略函数的内联声明，而将内联函数作为普通函数处理。

重载函数

参考Java的重载，值得注意的是C++还可以重载运算符，后面会介绍。

模板函数

关键字：template

```
    template<typename T>
    void display_message(const string &msg,const vector<T> &vec)
    {
        /*CODE*/
    }
```

函数指针

函数指针的定义形式如下，我们给个实例：

```
int test(int a)
{
    return a;
}

int main()
{
    //fp是函数指针的名字，前面加一个*星号，然后用括号括起来，以防止被认为是返回int*类型的函数
    //后面一个括号是你想指向的函数的参数列表，返回值类型也要跟指向的函数返回类型一致
```

```
    //给函数指针赋初始值为0就是不指向任何函数
    //直接将函数的名称用等号赋值给函数指针就行
    int (*fp)(int)=0;
    fp=test
}
```

- 可以将函数指针与数组共用，来达到函数指针数组的作用。
- 直接将函数名赋值给函数指针就行。
- 函数指针也能减少重复代码量

枚举enum

设定头文件

.h文件

- 在头文件中的函数声明不用加extern关键字，而需要多次使用的对象要加（比如指向函数的指针），否则会被认为是对象的定义，这就违背了“可以有多个声明，但只能有一个定义”的原则，但是这个原则也有个例外，就是const object，头文件中的const object不用加extern，因为const object的定义只要一出文件就不可见。

第三章 泛型编程风格

Standard Template Library(STL)

包括：

- 容器
- 泛型算法

指针的算术运算

- 可以直接通过加减法实现
- 可以通过下标运算符实现

array、vector、list的区别

- array、vector在内存中都是一块连续的地址，而list的内存结构是分散的，通过指针来链接前后元素，类似双向链表。

Iterator——泛型指针！！！

一句话：**很强大！！！**

所有容器的共通操作

- equality（==）和inequality（！=）运算符
- assignment（=）运算符，赋值
- empty() 容器为空返回true，否则返回false
- size() 返回容器的元素个数
- clear() 删除所有元素

- begin() 返回一个iterator，指向容器的第一个元素
- end() 返回一个iterator，指向容器的最后一个元素的 ***下一个位置***
- insert() 将单一或某个范围内的元素插入容器内
  - insert()有四个重载的方法
- erase()将容器内的单一元素或某个范围内的元素删除

> insert()和erase()的行为视容器本身为顺序性（sequential）容器或关联（associative）容器而有所不同
>
> 这两个函数的参数都是泛型指针iterator

顺序性容器

有：

- vector——末尾插入元素效率高
- list（双向链表）
- deque——最前端插入元素、末尾删除元素效率高（反队列？）

特点：

- 随机访问

> list不支持iterator的偏移运算，例如：slist.erase(it1,it1+num)
>
> 应该是因为list在内存中并不是连续的原因

使用泛型算法

头文件：#include

一些搜索算法:

1. find()
2. binary_search()
3. count()
4. search()

Function Object

三个部分：

- 六个算术运算
- 六个关系运算
- 三个逻辑运算

个人理解：

相当于把运算符函数化以实现泛型编程

Function Object Adapter

用于对Function Object进行修改操作

使用Map

键值对

map对象中有两个member：

- first
- second

first代表key域，second代表second域

要添加一个键值对的话，直接 `map[key]=value;`

查询map中是否含有一个键值对，可用map自带的find()方法： `map.find(key)`

使用Set

一群key的集合，有count方法，用来查找set中是否存在某个key。

set中的key不能重复，要想能重复的话，就要用multiset

Iterator Inserter

> 头文件：#include

对于vector来说，有三种insertion adapter

- back_inserter()
- front_inserter()
- inserter()

iostream Iterator

> 头文件：#include

- istream_iterator
- ostream_iterator

在定义istream_iterator是不为它指定istream对象，就代表end-of-file，例如：

```
istream_iterator<string> eof;
```

iostream iterator 可以绑定到不同的输入输出流去，例如：标准输入输出流（cin、cout）、文件输入输出流（ifstream、ofstream）

第四章 基于对象的编程风格

实现一个class

首先要声明class，然后定义，或者在声明的同时定义。每个class分为public和private两个部分。在class中声明的方法可以同时定义（在class内部定义的话，就会自动被识别为inline函数），也可以在class的外部定义，但要使用 `::` （类作用域解析运算符）

```
class Stack
{
    public:
        bool push(const string &elem)
        {
            _stack.push_back(elem);

            return true;
        }
        bool pop(string &elem)
        {
            elem = _stack.back();

            return true;
```

```
        }
        // bool peek(string &elem);

        bool empty();
        // bool full();

        int size()
        {
            return _stack.size();
        }
    private:
        vector<string> _stack;
};
bool Stack::empty()
{
    return _stack.empty();
}
```

构造函数和析构函数

C++的构造函数与Java的形式基本相同：

```
class Triangular
{
    public:
        Triangular();
        Triangular(int len);
        Triangular(int len,int beg_pos);
    private:
        string _name;
        int _next,_length,_beg_post;
};
```

值得注意的是，C++可以通过赋值的形式来调用单一参数的constructor，例如：

```
    Triangular t=8;
```

对于default constructor，除了最简单的形式：`Triangular();`,还可以给参数设置默认值：`Triangular(int len=1,int bp=1);`这也是一种默认的constructor，但这两种形式的default constructor不能同时存在！

Member Initialization List（成员初始化列表）

这是初始化class的第二种语法。实例：

```
Triangular::Triangular(int len,int bp)
    :_name("Triangular")
    {
        _length = len>0?len:1;
        _beg_pos = bp>0?bp:1;
    }
```

这个例子中Triangular参数列表后的`:_mame("Triangular")`就是初始化成员列表，它的意思是将字符串Triangular传给_name，下方的括号还是constructor执行的内容。如果有多个要在成员初始化列表中初始化的变量，用英文逗号隔开，例如：`: _name("Triangular"), _length(2)`。

Destructor

destructor与constructor对立，destructor由用户定义，一旦用户定义了，当一个object生命结束时，就会自动调用desctructor处理善后。Destructor主要用来释放constructor中或对象周期中分配的资源。

语法规则：在构造函数的名字前面加一个`~`号，没有返回值也没有参数，不能被重载。

```
class Matrix
{
    public:
        Matrix(){};

        ~Matrix()
        {
            //释放资源等操作
        }
}
```

Memberwise Initialization（成员逐一初始化）

默认情况下，将一个object赋值给另一个object（同class）时，例：

```
Triangular tri1(8);
Triangular tri2=tri1;
```

tri1的所有class data member都会被一一赋值给tri2。但有时候这种默认的做法就不好，例如，当一个class中有new一个其他的对象，如果此时tri1中的所有data member赋值给tri2，tri2中的那个对象的引用还是指向tri1中的对象的，若tri2比tri1先结束生命，且desctructor中有释放了那个object的内存，那么之后tri1再调用那个object就会发生错误。因此，此时要在建立一个copy constructor，将tri1中的对象的值赋值给tri2中另一个对象。

Mutable and Const

c语言中的const貌似只用来声明变量是不可改变的？但C++中的const还可以用来标识一个member function有没有对class object的内容进行更改，若有修改，则编译不会通过。其语法是在function的参数列表括号后、大括号之前加一个const关键字：

```
bool Triangular::next(int &value) const
{
    //...
}
```

对一个引用参数加上const，就要保证在这个方法内所有对这个参数的操作都不会改变其值，这些方法的定义处也要加上const来告诉编译器。若其中有些方法虽然改变了其中的某个值，但是不影响object的常量性，比如改变了一个object中的iterator的指针_next来达到循环遍历的操作，这个改变就不影响object的常量性。我们可以在`int _next;`前加一个mutable关键字，`mutable int _next;`

this指针

this指针是自动加在类中函数的参数列表中的，所以可以在函数中直接使用this指针，C++中的this指针与Java中的this指针作用无异。

静态类成员

静态data member对于同一类的所有对象都是共享的，只有一个值。调用data member也是要通过class名来调用，类似Java调用静态成员，不过C++用的是 `::` 符号。

静态的member function也是类似Java，声明为static的function不能调用non-static member，也通过 `::` 来调用。当在class内部声明一个函数为static时，在class外部定义的时候就不用加上static了（data member也是一样的）。

打造一个Iterator Class

运算符重载（有点意思）

嵌套类型（Nested Type）

```
typedef existing_type new_name;
```

将一个已存在的类型名赋予另一个新名字。学过C语言的人应该发现，这应该并不是Nested Type.....的确，这只是个typedef关键字。真正的Nested Type的含义是在一个类型中被定义的另一个类型。例如：

```
class Triangular{
    public:
        typedef Triangular_iterator iterator;

    private:
}
```

这其中的Triangular_iterator就是一个Nested Type。

Friend关键字

在一个class内可以将其他non-member function 或class指定为friend，写法是在class内在non-member function前加一个friend关键字。

被标记为friend的non-member function或class拥有与class member function同样的访问权限，都能访问private member。

友谊的建立通常是为了效率考虑。例如在某个non-member运算符函数中进行Point和Matrix的乘法运算。如果只是希望进行某个data member的读取和写入。那么，为他提供具有public访问的inline函数，是一个比较好的替代方案（类似在public域中设置get、set方法）。

实现copy assignment operator

即重载运算符 `=` ，将函数声明为 `operator=(xxx)`

实现一个function object

所谓function object就是提供有function call运算符的class。而function call的名字叫 `operator()` ，没错，名字中就带有一对括号，因此在声明它的时候还要在后面再加一对括号来接收参数。而后直接使用 `类名(参数);` 就可以直接调用这个operator()函数。例:

```
class LessThan//用来判断传入的_value值是否小于class内的_val值；
{
    public:
        LessThan(int val): _val(val){}
        //其他的xxx函数

        bool operator()(int _value) const;//当然，还要具体实现
    private:
        int _val;
}

/*==============分割线=============*/
//在调用的时候就可以这么写
int main()
{
    LessThan lt10(10);

    //结果应该是真
    cout<<lt10(5)<<endl;
    return 0;
}
```

重载iostream运算符

重载 `<<` 和 `>>` 两个运算符的形式就是写两个函数：

```
ostream& operator<<(/*参数*/)
{

}

/*==========================*/
istream& operator>>(/*参数*/)
{

}
```

> member function 的操作数必须是隶属于同一个class的对象

指针，指向Class Member Function

member function的指针的声明形式与之前的函数指针几乎一样，只不过要在括号内的`*`号前加上class名和`::`符号运算。例：

```
void (Triangular::*fp)(int);
```

可以用typedef来简化函数指针的声明：

```
typedef void (Triangular::*fp)(int);
fp p = 0;
//现在fp就相当于一个类型名，p才是一个具体的函数指针名了
//但是将Triangular中的函数的地址赋值给p则要加一个取值符号，而不是像普通的函数只用写函数名即可
p = &Triangular::test;
```

同时，若该Class Member Function在一个class内部，调用它时要在指针名前加一个*号。

第五章 面向对象编程风格

面向对象编程概念的两项最主要特质是：

- 继承
- 多态

protected域

protected部分的member可以被该类及其子类访问，而private只能被该类访问，因此在设计继承时，需要从父类继承的member应该放在protected中。

抽象基类

我们一般将抽象基类写成一个"接口"——其中的函数全都是纯虚函数（除了static的方法和变量）。

> 一个类中包含一个或多个纯虚函数，他就不能被实例化，只能作为父类，由其子类将纯虚函数实现之后才能实例化。相当于Java中的接口(Interface)。

纯虚函数的设置：将一个函数赋值为0。

```
virtual void gen_elems(int pos) = 0;
```

同时，一般不会为基类设计单独的constructor，但是要设计desctructor，不过desctructor一般为空，且不为纯虚函数。

若要实现Java中用父类的引用来指向子类的实体，则需声明为指针：

```
num_sequence *ps = new Fibonacci(12);
//num_sequence为Fibonacci的基类
```

基类该多么抽象

reference与pointer的区别

引用很容易与指针混淆，它们之间有三个主要的不同：

- 不存在空引用，即不能为null。引用必须连接到一块合法的内存。
- 一旦引用被初始化为一个对象，就不能被指向到另一个对象。指针可以在任何时候指向到另一个对象。
- 引用必须在创建时被初始化。指针可以在任何时间被初始化。

如何在初始化子类的同时初始化基类？

有时候基类会有一些必须要通过constructor初始化的data member，例如某个引用，这时就要在子类初始化的时候给基类传入参数。具体写法是这样：

```
inline Fibonacci::Fibonacci(int len,int beg_pos):num_sequence(len,beg_pos,_elems)
{}
```

运行时的鉴定机制

static_cast<>()

`static_cast<>()` 函数能将括号中的类型**无条件**转化为尖括号中的类型，它在转化是不会检验这样的转化是否合理。

dynamic_cast<>()

`dynamic_cast<>()` 函数将括号中的类型**有条件**转化为尖括号中的类型，它会进行检验操作，确保这样的类型转化不会引发错误。

第六章 以template进行编程

语法：

```
template<typename1 Type1,typename2 Type2...>
class BTnode{

    //...
}
```

在class template中定义一个函数就跟non-template class中定义一样，但是在该类的外部定义时，语法的差别就比较大了：

```
template<typename Type,typename2 Type2...>
inline BTnode<Type>::BTnode()
{
    //...
}
```

Templatele类型参数的处理

建议将所有的template类型参数视为class类型来处理，并且在参数列表声明为 `const reference` 类型（这样效率高，第一章也提到过）。

Template参数

template不仅能作为一个模板，也能用来“声明”class或者方法的参数。

```
template<int len>
class test
{
    public:
        int getLen(){
            return len;
        }

};

/*=========================*/
int main()
{
    test<2> t;
    cout<<t.getLen()<<endl;//输出结果为2
    return 0;
}
```

Member Template Function

可以将一个non-template class内的member function定义为template形式。

```
class PrintIt
{
    public:
        PrintIt(ostream &os):_os(os){};

        template<typename elemType>
        void print(const elemType &elem,char delimiter = '\n')
        {
            _os<<elem<<delimiter;
        }
    private:
        ostream& _os;
};

/*======================*/

int main()
{
    PrintIt p(cout);

    p.print("hello");
    p.print(1024);
    //从这可以看出调用template member function的时候不用加尖括号，直接在对应参数上添上你想输出的变量
    return 0;
}
```

第七章 异常处理

抛出异常

C++的异常抛出也是 `throw` 表达式。我们抛出的任何异常都是一个对象（object）或者class（标准异常）。

```
bool some_function()
{
    bool status = true;

    if(/*args*/)
    {
        throw iterator_overflow(_index,Triangular::max_elems);
        //这个Iterator_overflow是一个自定的异常类，他可以不用继承exception类（后面会说）
    }

    return status;
}
```

捕获异常

C++捕获异常也是靠catch子句来捕获，例：

```
bool some_function()
{
    bool status = true;

    //...

    catch( int errno)
    {
        //...
    }
    catch( const char *str)
    {
        //...
    }

    return status;
}
```

括号中的表达式相当于限定了这个catch所能处理的异常类型。每个抛出的异常会被相应的catch模块捕获并运行其中的代码。同时注意，catch必须和try一起使用（下一节）。

还有一种语法可以捕获任何类型的异常：

```
catch(...)//没错，就是三个省略号
{
    //...
}
```

提炼异常

catch子句应该与try块相应而生（类似Java）。

```
bool some_function()
{
    bool status = true;

    try
    {
        //...
    }
    catch(int errno)
    {
        //...
    }
    return status;
}
```

C++中的异常处理机制也是与Java类似：在具体的某一个触发throw语句的地方检测是否位于try块内，如果是，就运行相应的catch，异常被处理之后，***程序继续正常执行下去***。若不存在于try内，异常处理机制会在**“调用含有该异常的模块”**（也就是**“上一级”**）内搜寻try-catch块，做类似的事情，若不含，则继续向上搜寻上一级，直到main函数中。若在main()中还是找不到合适的处理程序，则会调用标准库提供的terminate()——默认终端整个程序的执行。

局部资源管理

为了确保异常的出现不会影响系统释放资源（传统的释放资源是在对象外部进行），我们建议使用一种**资源管理手法**——在初始化阶段内即进行资源请求、在destructor内释放资源。这样就保证了，即使出现异常，系统也能自动调用局部对象的destructor，进行资源释放。

```
class Example
{
    public:
        Example(int test):_test(test)
        {/*请求资源*/}
        ~Example()
        {
            /*释放资源*/
        }
    private:
        int _test;
};
```

auto_ptr

是标准库提供的class template，它会自动删除通过new表达式分配的对象。

使用它之前，要 `#include<memory>`

```
auto_ptr<string> aps(new string("test"));
```

标准异常

C++本身也提供了一些标准异常，他们都是基类 `exception` 的派生类，定义在 `<exception>` 中，具体的请自行百度。

其中有一个 `bad_alloc` 异常，它会在无法从空闲空间中分配到足够的空间来new一个对象时被抛出。

我们可以直接这样捕获这个异常 `catch(bad_alloc)` ，当然其他的标准异常也都可以这样捕获，但是这样就不能在catch中操作异常了。为了刨剔这个问题，我们可以通过catch捕获exception，这样任何exception的子类都能被捕获了：

```
catch(const exception &ex)
{
    cerr<<ex.what()<<endl;  //what()是exception的一个虚函数
}
```

我们自定义的任何异常也能继承exception，从而被上述的catch所捕获。

ostringstream与istringstream