2018年春

程序设计实习(I): C++程序设计

第十四讲 C++高级主题

刘家瑛

liujiaying@pku.edu.cn

主要内容

- **三种常用的C++机制**
  - C++的cast运算符
  - 智能指针 auto_ptr
  - 异常处理
- **实战编程问题**
  - 条件编译
  - 多文件工程的问题

类型转换

- 在C语言中，**类型转换**有几种方式：
  - **(expression)** 在表达式外边加括号，由编译器来决定怎么改变
  - **new_type (expression)** 强制类型括号住表达式
  - **(new_type) expression** 括号住强制类型

- 若没有了解每个转换的细节，就有可能出现问题，例如：
  - 指针指向不应该指向的区域：

    出现**野指针**或者**指向位置错误**
  - 计算**数值被截去**

C++的cast运算符

- 为了C语言类型转换中语义模糊和固有的危险陷阱,

← C语言不去判断所要操作的类型转换是否合理

- C++的四种 "cast" 运算符
  - **static_cast**
  - **dynamic_cast**
  - **reinterpret_cast**
  - **const_cast**

- **C++强制类型转换运算符**

强制类型转换运算符<要转换到的类型> (待转换表达式)

static_cast, interpret_cast, const_cast和dynamic_cast

1. static_cast

- static_cast用来进用行比较"自然"和低风险的转换

  e.g. 整型和实数型, 字符型之间互相转换

- static_cast 不能用于**不同类型的指针**之间互相转换

  不能用于**整型和指针**之间的互相转换

  不能用于**不同类型的引用**之间的转换

```
//program 20.1.1.cpp  static_cast示例
#include <iostream>
using namespace std;
class A{
   public:
        operator int() { return 1; }
        operator char * (){ return NULL; }
};
int main(){
        A a;
        int n;
        char * p = "New Dragon Inn";
        n = static_cast<int>(3.14); // n 的值变为 3
        n = static_cast<int>(a);     //调用a.operator int, n的值变为 1
```

```
    p = static_cast<char*>(a);
    //调用a.operator char *，p的值变为 NULL
    n = static_cast<int> (p);
    //编译错误，static_cast不能将指针转换成整型
    p = static_cast<char*>(n);
    //编译错误，static_cast不能将整型转换成指针
    return 0;
}
```


2. reinterpret_cast

- 不同类型的**指针**之间的转换
- 不同类型的**引用**之间转换
- 指针和能容纳得下指针的整数类型之间的转换

  转换的时候，执行的是**逐个比特拷贝**的操作

```
//program 20.1.2.cpp reinterpret_cast示例
#include <iostream>
using namespace std;
class A{
      public:
            int i;
            int j;
             A(int n):i(n),  j(n) { }
};
int main(){
              A a(100);
              int & r = reinterpret_cast<int&>(a); //强行让 r 引用 a
              r = 200;  //把 a.i 变成了 200
              cout << a.i << “, ” << a.j << endl;  // 输出 200, 100
              int n = 300;
```

```
A * pa = reinterpret_cast<A*> (& n); //强行让 pa 指向 n
pa->i = 400;  // n 变成 400
pa->j = 500;  //此条语句不安全,很可能导致程序崩溃
cout << n << endl;  // 输出 400
long long la = 0x12345678abcdLL;
pa = reinterpret_cast<A*>(la);
// la太长，只取低32位0x5678abcd拷贝给pa
unsigned int u = reinterpret_cast<unsigned int>(pa);
//pa逐个比特拷贝到u
cout << hex << u << endl;  //输出 5678abcd
typedef void (* PF1) (int);  //函数指针，后面那个括号
typedef int (* PF2) (int,  char *);  //是函数的参数
PF1 pf1;        PF2 pf2;
pf2 = reinterpret_cast<PF2>(pf1);
//两个不同类型的函数指针之间可以互相转换
}
```

输出结果:
200, 100
400
5678abcd

3. const_cast

- 用来进行**去除const属性**的转换
- 将const引用转换成同类型的非const引用
- 将const指针转换为同类型的非const指针时用它
- 例如：

```
const string s = "Inception";
string & p = const_cast<string&>(s);
string * ps = const_cast<string*>(&s); // &s的类型是
                                       // const string *
```

4. dynamic_cast

- 用于将**多态基类的指针或引用**，
  强制转换为**派生类的指针或引用**
- 且能够检查转换的安全性
- 对于不安全的指针转换，转换结果返回NULL指针
- dynamic_cast**不能用于将非多态**基类的指针或引用，
  强制转换为派生类的指针或引用

回顾C++存在基类时的类型转换规则

- f(派生类) → y(基类)：**f:对象/对象地址; y:对象/引用/指针**
- 基类指针 → **(强制指针类型转换)**派生类指针

```
//program 20.1.3.cpp dynamic_cast示例
#include <iostream>
#include <string>
using namespace std;

class Base{   //有虚函数, 因此是多态基类
      public:
            virtual ~Base() { }
};
class Derived : public Base  {  };
int main()
{
            Base b;
            Derived d;
            Derived * pd;
            pd = reinterpret_cast<Derived*> (&b);
```

```
    if( pd == NULL) //此处pd不会为NULL
                    // reinterpret_cast不检查安全性，总是进行转换
        cout << "unsafe reinterpret_cast" << endl; //不会执行
    pd = dynamic_cast<Derived*> (&b);
    if( pd == NULL) //结果会是NULL, 因为 &b不是
                        //指向派生类对象, 此转换不安全
        cout << "unsafe dynamic_cast1" << endl;  //会执行
    Base * pb = & d;
    pd = dynamic_cast<Derived*> (pb); //安全的转换
    if( pd == NULL)  //此处pd 不会为NULL
        cout << "unsafe dynamic_cast2" << endl; //不会执行
    return 0;
}
```

输出结果：

unsafe dynamic_cast1

Derived & r = dynamic_cast<Derived&>(b);

那该如何判断该转换是否安全呢?

答案: 不安全则抛出异常

智能指针 auto_ptr

- auto_ptr是一个类模板
- 维护动态分配内存的指针
  - 提供运算符*, ->
  - 当auto_ptr对象被释放时,

    将自动对所维护的指针执行 delete操作
  - 防止内存泄漏

```
#include <iostream>
#include <memory>
using namespace std;
class myClass {
public:
    ~myClass() { cout << "destructor myclass" << endl; }
};
int main()
{
    auto_ptr<myClass>  ptr( new myClass );
    return 0;
}
```

输出结果:

```
destructor myclass
```


智能指针 auto_ptr

- **指针型变量**
  - 指向的动态分配的内存空间不会自动被释放

- 用**auto_ptr对象代替指针**, 则
  - 释放auto_ptr对象,
  - 将引起对指针所指向的动态分配的内存空间的delete操作

智能指针auto_ptr

- 通过auto_ptr的构造函数, 可以让auto_ptr对象托管一个new运算符返回的指针, 写法如下:

  auto_ptr<T> ptr(new T);

  // T可以是 int, char, 类名等各种类型
- 之后ptr就可以像 T* 类型的指针一样来使用
- 即 *ptr 就是用new动态分配的那个对象, 且不必操心释放内存的事
- **auto_ptr对象不能托管指向动态分配的数组的指针**

```
//program 20.3.1.cpp auto_ptr示例1
#include <iostream>
#include <memory>
using namespace std;
class A{
public:
        int i;
        A(int n):i(n) { };
        ~A() { cout << i << " " << "destructed" << endl; }
};
int main() {
        auto_ptr<A> ptr(new A(2)); // new出来的动态对象的指针
                                   //交给ptr托管
        cout << ptr->i << endl;    // 输出 2
        ptr->i = 100;              //动态对象的 i 成员变量变为 100
```


```
    A a(*ptr);                  // * ptr就是前面 new 的动态对象
    cout << a.i << endl;  //输出 100
    a.i = 20;
    return 0;
}
```

输出结果：

```
2
100
20 destructed // a析构
100 destructed //new创建的对象析构
```


auto_ptr成员函数

类auto_ptr<T>有以下常用的成员函数：

- **T * release();**

解除对指针的托管，并返回该指针

解除对指针的托管，并不会delete该指针

- **void reset(T *p = NULL);**

delete原来托管的指针，托管新指针p

若p为NULL，则执行后变成没有托管任何指针

- **T * get() const;**

返回托管的指针

```
//program 20.3.2.cpp auto_ptr示例2
#include <iostream>
#include <memory>
using namespace std;
class A{
public:
        int i;
        A(int n):i(n) { };
        ~A() { cout << i << " " << "destructed" << endl; }
};
int main(){
        auto_ptr<A> ptr1(new A(2)); //A(2)由ptr1托管,
        auto_ptr<A> ptr2(ptr1);     //A(2)交由ptr2托管,
                                    //ptr1什么都不托管
        auto_ptr<A> ptr3;
```


```
    ptr3 = ptr2;            //A(2)交由ptr3托管, ptr2什么都不托管
    cout << ptr3->i << endl; //输出 2
    A * p = ptr3.release();  // p 指向 A(2), ptr3解除对A(2)托管
    ptr1.reset(p);           //ptr1重新托管A(2)
    cout << ptr1->i << endl; //输出 2
    ptr1.reset(new A(3));  // delete 掉A(2), 托管A(3), 输出 2
                           // destructed
    cout << "end" << endl;
    return 0; //程序结束, ptr1消亡时, 会delete 掉A(3)
}
```

输出结果：

```
2
2
2 destructed
end
3 destructed
```


异常处理

- C++异常处理基础：try, throw, catch
- 异常声明 (exception specification)
- 意外异常 (unexpected exception)
- 异常处理的作用
- 动态内存管理的异常处理
  - new

程序运行发生异常

- 程序运行中总难免发生错误
  - 数组元素的**下标超界**, 访问NULL指针
  - **除数为0**
  - 动态内存分配new需要的**存储空间太大**
  - …

程序运行发生异常

- 引起这些异常情况的**原因**:
  - 代码质量不高, 存在BUG
  - 输入数据不符合要求
  - 程序的算法设计时考虑不周到
  - …

程序运行发生异常

- 我们总希望**在发生异常情况时**
  - 不只是简单地终止程序运行
  - 能够反馈异常情况的信息:

    哪一段代码发生的,什么异常
  - 能够对程序运行中已发生的事情做些处理:

    取消对输入文件的改动、释放已经申请的系统资源
  - …

异常检测

- 通常的做法：

在预计会发生异常的地方，加入相应的代码，

但这种做法并不总是适用的

```
…                          //对文件A进行了相关的操作
fun( arg, … );  //可能发生异常
…
```


异常检测

- **caller**该如何知道**fun(arg, …)**是否发生异常
  - 没有发生异常，可以继续执行
  - 发生异常，应该在结束程序运行前还原对文件A的操作
- **fun(arg, …)**是别人已经开发好的代码
  - **fun(arg, …)**的编写者不知道其他人会如何使用这个函数
  - **fun(arg, …)**会出现在表达式中，通过返回值的方式区分是否发生异常
    - 不符合编写程序的习惯
    - 可能发生多种异常，通过返回值判断也很麻烦

异常处理基本语句

throw语句

throw 表达式;

- 抛出一个异常，异常是一个表达式
- 其值类型可以是一个基本类型，也可以是个类

try …catch语句

```
try{
    语句组
}
catch(异常类型) {
    异常处理代码
}
```


try...catch

```
#include <iostream>
using namespace std;
int main()
{
        double m , n;
        cin >> m >> n;
        try {
                cout << "before dividing." << endl;
                if( n == 0)
                        throw -1; //抛出int类型异常
                else
                        cout << m / n << endl;
                cout << "after dividing." << endl;
        }
```


```
        catch(double d) {
                cout << "catch(double) " << d <<  endl;
        }
        catch(int e) {
                cout << "catch(int) " << e << endl;
        }
        cout << "finished" << endl;
        return 0;
}
```

程序运行结果如下：

9 6↙

before dividing.

1.5

after dividing.

finished

```
//program 20.4.2.cpp  捕获任何异常的catch块
#include <iostream>
using namespace std;
int main()
{
        double m , n;
        cin >> m >> n;
        try {
                cout << "before dividing." << endl;
                if( n == 0)
                        throw -1;    //抛出整型异常
                else if( m == 0 )
                        throw -1.0; //抛出double型异常
                else
                        cout << m / n << endl;
                cout << "after dividing." << endl;
        }
```


```
        catch(double d) {
                cout << "catch(double) " << d <<  endl;
        }
        catch(...) { //匹配任何类型的异常
                cout << "catch(...) " << endl;
        }
        cout << "finished" << endl;
        return 0;
}
```

程序运行结果：

9 0↙

before dividing.

catch(...)

finished

0 6↙

before dividing.

catch(double) -1

finished

异常的再抛出

如果一个函数在执行的过程中，

- 抛出的异常在本函数内就被catch块捕获并处理了，那么该异常就不会抛给这个函数的调用者(也称“上一层的函数”)
- 如果异常在本函数中没被处理，就会被抛给上一层的函数

```
//protram 20.4.3.cpp   异常再抛出
#include <iostream>
#include <string>
using namespace std;
class CException{
    public :
        string msg;
        CException(string s):msg(s) { }
};
```

```
double Devide(double x,  double y){
        if(y == 0)
                throw CException("devided by zero");
        cout << "in Devide" << endl;
        return x / y;
}
int CountTax(int salary){
        try {
                if( salary < 0 )
                        throw -1;
                cout << "counting tax" << endl;
        }
        catch ( int ) {
                cout << "salary < 0" << endl;
        }
```

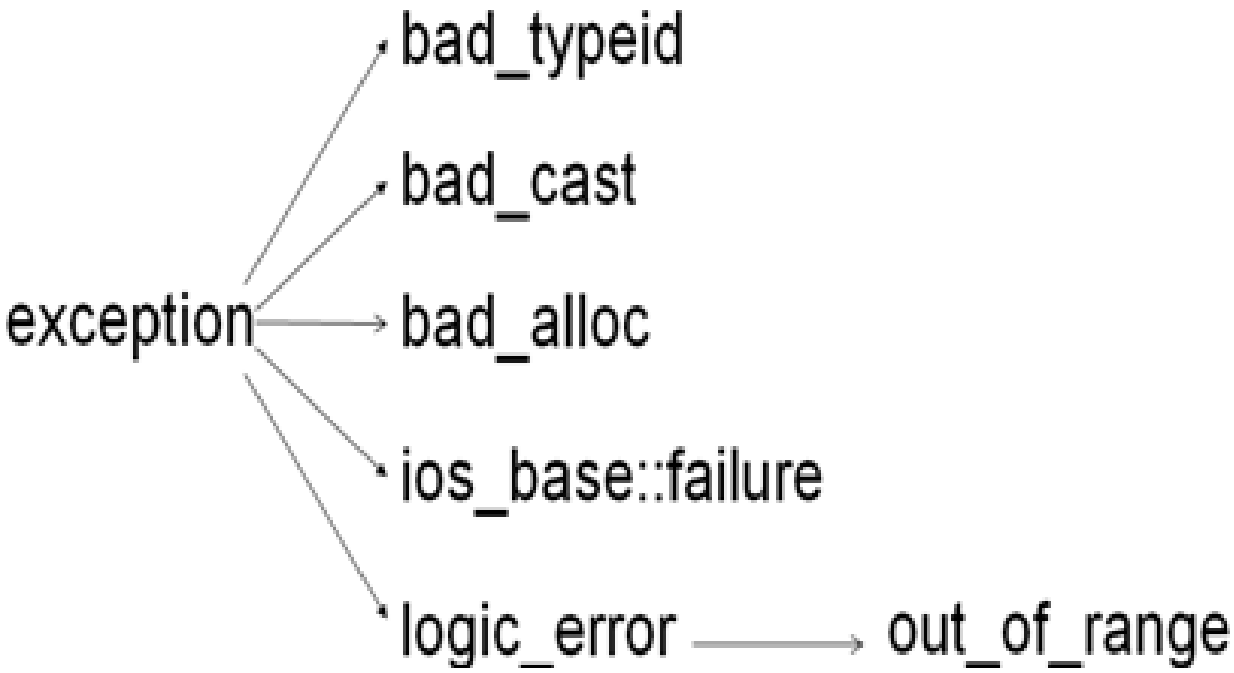

```
        cout << "tax counted" << endl;
        return salary * 0.15;
}
int main(){
        double f = 1.2;
        try {
                CountTax(-1);
                f = Devide(3, 0);
                cout << "end of try block" << endl;
        }
        catch(CException e) {
                cout << e.msg << endl;
        }
        cout << "f=" << f << endl;
        cout << "finished" << endl;
        return 0;
}
```

输出结果：

```
salary < 0
tax counted
devided by zero
f=1.2
finished
```

C++标准异常类

C++标准库中有一些类代表异常，这些类都是从**exception类**派生而来，常用的几个异常类如下：



bad_cast

在用 dynamic_cast进行从多态基类对象(或引用), 到派生类的引用的强制类型转换时, 如果转换是不安全的, 则会抛出此异常

```
//program 20.4.5.cpp  bad_cast异常
#include <iostream>
#include <stdexcept>
using namespace std;
class Base
{
        virtual void func(){}
};
class Derived : public Base
{
public:
        void Print() { }
};
```


```
void PrintObj( Base & b){
        try  {
            Derived & rd = dynamic_cast<Derived&>(b);
            //此转换若不安全, 会抛出bad_cast异常
            rd.Print();
        }
        catch (bad_cast& e) {
            cerr << e.what() << endl;
        }
}
int main (){
        Base b;
        PrintObj(b);
        return 0;
}
```

输出结果：

Bad dynamic_cast!

bad_alloc

在用new运算符进行动态内存分配时,
如果没有足够的内存,则会引发此异常

```
//program 20.4.6.cpp bad_alloc异常
#include <iostream>
#include <stdexcept>
using namespace std;
int main (){
  try  {
  char * p = new char[0x7fffffff]; //无法分配这么多空间,会抛出异常
  }
  catch (bad_alloc & e)  {
        cerr << e.what() << endl;
  }
  return 0;
}
```

输出结果:
bad allocation

out_of_range

用vector或string的at成员函数根据下标访问元素时，如果下标越界，就会抛出此异常。例如：

```
//program 20.4.7.cpp out_of_range异常
#include <iostream>
#include <stdexcept>
#include <vector>
#include <string>
using namespace std;
int main (){
vector<int> v(10);
    try {
        v.at(100)=100;  //抛出out_of_range异常
    }
    catch (out_of_range& e) {
        cerr << e.what() << endl;
    }
```


```
    string s = "hello";
    try {
        char c = s.at(100); //抛出out_of_range异常
    }
    catch (out_of_range& e) {
        cerr << e.what() << endl;
    }
    return 0;
}
```

输出结果:

```
invalid vector<T> subscript
invalid string position
```


预编译

- C++语言中, **预编译**(编译预处理) 指在真正开始对程序编译前, 对源程序进行一番处理, 形成一个临时源程序文件
- 预编译不会修改程序
- 主要包含:
  - 符号常量的定义和宏定义 (内联函数)
  - 文件包含
  - 条件编译

条件编译

```
#define DEBUG_VERSION
#ifdef DEBUG_VERSION
        第一部分
#else
        第二部分
#endif
```

- 如果有 #define DEBUG_VERSION则第一部分被编译，否则第二部分被编译
- **#ifndef** 则相反

条件编译

- 因此编写.h文件时，往往这样写：

```
#ifndef  SOMEHEAD_H
#define  SOMEHEAD_H
头文件具体内容
#endif
```

- 这样能避免在多个文件的工程中，有的头文件可能被重复包含的问题

多文件工程的问题 (1)

1) 如何使用在其他文件中定义的全局变量?

a.cpp 里定义了全局变量 int a;

b.cpp 里要用, 则如下声明:

extern int a;

2) 如何使用在其他文件中定义的全局函数?

a.cpp里写了函数 void fun() { }

b.cpp里要用, 则在b.cpp里写出函数声明即可:

extern void fun();

3) 如何使用在其他文件中写的类?

a.h 写类的声明, a.cpp里写类的成员函数的实现, 那么在b.cpp里只要**#include <a.h>**就可以使用该类了

多文件工程的问题 (2)

4) 如何写一个类库或函数库给别人用, 却不让别人看到源代码?

- 把你的函数或类实现在 myclass.cpp里
- 函数或类的声明放在 myclass.h里
- 编译之, 得到 myclass.obj文件

把myclass.h和myclass.obj提供给别人,

别人 #include <myclass.h>

然后在编译器的链接选项里指定要链接 myclass.obj即可

多文件工程的问题(3)

5) 如果有一个别人用C写的函数库我要在C++工程里用，怎么办?

在别人提供的函数声明前加 extern "C"

```
extern "C" void OthersFunction();
```

否则会在链接的时候找不到函数

```
extern "C" {
    void OthersFunctoin();
    …..
}
```

或者将所有别人的C函数的声明都写在上面的花括号中间

C++部分完美谢幕啦~