C/C++的空指针

熟悉C++的都知道，为了避免“野指针”（即指针在首次使用之前没有进行初始化）的出现，我们声明一个指针后最好马上对其进行初始化操作。如果暂时不明确该指针指向哪个变量，则需要赋予NULL值。除了NULL之外，C++11新标准中又引入了nullptr来声明一个“空指针”，这样，我们就有下面三种方法来获取一个“空指针”：

如下：

```
int *p1 = NULL; // 需要引入cstdlib头文件
int *p2 = 0;
int *p3 = nullptr;
```

新标准中建议使用nullptr代替NULL来声明空指针。到这里，大家心里有没有疑问：为什么C++11要引入nullptr？它与NULL相比又有什么不同呢？这就是我们今天要解决的问题。

C/C++中的NULL到底是什么

我们查看一下C和C++的源码，不难发现：

1. NULL在C++中的定义，NULL在C++中被明确定义为整数0：

```
#ifndef NULL
   #ifdef __cplusplus
       #define NULL 0
   #else
       #define NULL ((void *)0)
   #endif
#endif
```

2.NULL在C中的定义.在C中，NULL通常被定义为如下：

```
#define NULL   ((void *)0)
```

也就是说NULL实质上是一个void *指针。

那么问题又来了，我们从一开始学习C++的时候就被告诫C++是兼容C的，为什么对于NULL C++却不完全兼容C呢？

简单地说，C++之所以做出这样的选择，根本原因和C++的函数重载机制有关。考虑下面这段代码：

```
void Func(char *);
void Func(int);

// char *p1 = NULL;
// int *p2 = 0;

//#define NULL 0
//#define NULL ((void*)0)

int main(){
    Func(NULL);
    return 0;
}
```

如果C++让NULL也支持void *的隐式类型转换，这样编译器就不知道应该调用哪一个函数。

为什么要引入nullptr

C++把NULL定义为0，解决了函数重载后的函数匹配问题，但是又引入了另一个“问题”，同样是上面这段代码：

```
void Func(char *);
void Func(int);

int main(){
    Func(NULL); // 调用Func(int)
    return 0;
}
```

由于我们经常使用NULL表示空指针，所以从程序员的角度来看，Func（NULL）应该调用的是Func（char *）但实际上NULL的值是0，所以调用了Func（int）。nullptr关键字真是为了解决这个问题而引入的。

另外我们还有注意到NULL只是一个宏定义，而nullptr是一个C++关键字。

nullptr如何使用

nullptr关键字用于标识空指针，是std::nullptr_t类型的（constexpr）变量。它可以转换成任何指针类型和bool布尔类型（主要是为了兼容普通指针可以作为条件判断语句的写法），但是不能被转换为整数。

```
char *p1 = nullptr;     // 正确
int *p2 = nullptr;      // 正确
bool b = nullptr;       // 正确. if(b)判断为false
int a = nullptr;        // error
```

C和C++中const用法总结

1. 修饰常量时

```
const int temp1;     // temp1为常量，不可变
int const temp2;     // temp2为常量，不可变
```

2. 修饰指针时

主要看const在*的前后，在前则指针指向的内容为常量，在后则指针本身为常量；

```
const int *ptr;            // *ptr为常量；
int const *ptr;            // *ptr为常量；
int* const ptr;            // ptr为常量；
const int *const ptr;      // *ptr、ptr均为常量；
```

3. const修饰类对象时

const修饰类对象时，其对象中的任何成员都不能被修改。const修饰的对象，该对象的任何非const成员函数都不能调用该对象，因为任何非const成员函数都会有修改成员变量的可能。

```
class TEMP{
public:
    void func1();
    void func2() const;
};
const TEMP temp;
temp.func1();     // 错误；
temp.func2();     // 正确；
```

4. const 修饰成员变量

const修饰的成员变量不能被修改，同时只能在初始化列表中被初始化，因为常量只能被初始化，不能被赋值；

赋值是使用新值覆盖旧值构造函数是先为其开辟空间然后为其赋值，不是初始化；而初始化列表开辟空间和初始化是同时完成的，直接给与一个值，所以const成员变量一定要在初始化列表中完成。

```
class TEMP{
public:
    const int val;
    TEMP(int x):val(x){};       // 只能在初始化列表中赋值；
};
```

5. const修饰类的成员函数

const成员函数不能修改类对象中的任何非const成员变量。一般const写在函数的后面，形如：void func() const;

如果某个成员函数不会修改成员变量，那么最好将其声明为const，因为const成员函数不会对数据进行修改，如果修改，编译器将会报错；

```
class TEMP{
public:
    void func() const;  // 常成员函数，不能修改对象中的成员变量，也不能调用类中任何非const成员函数；
    void add();
};
```

对于const类对象，只能调用类中的const成员函数，所以const修饰成员函数的作用主要就是限制对const对象的使用。

6. const在函数声明中的使用

在函数声明中，const可以修饰函数的返回值，也可以修饰具体某一个形参；

修饰形参时，用相应的变量初始化const常量，在函数体内，按照const所修饰的部分进行常量化；

修饰函数返回值时，一般情况下，const修饰返回值多用于操作符的重载。通常不建议用const修饰函数的返回值类型为某个对象或某个对象引用的情况；

7. const常量与define宏定义区别

1）处理阶段不同

define是在预处理阶段，define常量从未被编译器看见，因为在预处理阶段就已经替换了；

const常量在编译阶段使用。

2）类型和安全检查不同

define没有类型，不做任何检查，仅仅是字符替换，没有类型安全检查，并且在字符替换时可能会产生意料不到的错误

const常量有明确的类型，在编译阶段会进行类型检查；

3）存储方式不同

define是字符替换，有多少地方使用，就会替换多少次，不会分配内存；

编译器通常不会为const常量分配空间，只是将它们保存在符号表内，使他们成为一个编译期间的一个常量，没有读取内存的操作，效率也很高；

8. mutable关键字

在C++中，mutable是为了突破const的限制而设置的。被mutable修饰的变量，将永远处于可变的状态，即使在一个const函数中，甚至结构体变量或者类对象为const，其mutable成员也可以被修改：

```
class ST {
public:
    int a;
    mutable int showCount;
    void Show() const;
};

void ST::Show() const{
    // a = 1;         // 错误，不能在const成员函数中修改普通变量
   showCount++;       // 正确
}
```

mutable只能修饰非静态数据成员；

9. const_cast

用于修改类型的const或volatile属性。

```
const_cast<type_id> (expression)
```

该运算符用来修改类型的const或volatile属性。除了const 或volatile修饰之外， type_id和expression的类型是一样的。

1）常量指针被转化成非常量的指针，并且仍然指向原来的对象；

2）常量引用被转换成非常量的引用，并且仍然绑定原来的对象；

3）const_cast一般用于修改指针或引用。如const char *p形式。

建议：

1、应该尽可能使用const，它会允许你指定一个语义约束（也就是指定一个不能被改动的对象），而编译器会强制实施这项约束。它允许你告诉编译器和其他程序员某值应该保持不变。

2、将某些东西声明为const可帮助编译器侦测出错误用法。const可被施加于任何作用域内的对象、函数参数、函数返回类型、成员函数本体；

3、编译器强制实施bitwise constness，但你编写程序时应该使用“概念上的常量性”（conceptual constness）；

4、当const和non_const成员函数有着实质等价的实现时，令non-const版本调用const版本可避免代码重复；

int A[10] ------ A[0] - A[9]

4 --- A[4]

15 ---- A[15]

-2 ----- A[-2]

异常处理

在程序设计过程中，我们总是希望自己设计的程序是天衣无缝的，但这几乎又是不可能的。即使程序编译通过，同时也实现了所需要的功能，也并不代表程序就已经完美无缺了，因为运行程序时还可能会遇到异常，例如当我们设计一个为用户计算除法的程序时，用户很有可能会将除数输入为零，又例如当我们需要打开一个文件的时候确发现该文件已经被删除了……类似的这种情况很有很多，针对这些特殊的情况，不加以防范是不行的。

我们通常希望自己编写的程序能够在异常的情况下也能作出相应的处理，而不至于程序莫名其妙地中断或者中止运行了。在设计程序时应充分考虑各种异常情况，并加以处理。

在C++中，**一个函数能够检测出异常并且将异常返回，这种机制称为抛出异常。**当抛出异常后，函数调用者捕获到该异常，并对该异常进行处理，我们称之为异常捕获。

C++新增throw关键字用于抛出异常，新增catch关键字用于捕获异常，新增try关键字尝试捕获异常。通常将尝试捕获的语句放在 try{ } 程序块中，而将异常处理语句置于 catch{ } 语句块中。 异常处理的基本语法如下所述。首先说一下抛出异常的基本语法：

throw 表达式;

抛出异常由throw关键字加上一个表达式构成。抛出异常后需要捕获异常以及异常处理程序，其基本语法如下：

```
try
{
   // 可能抛出异常的语句
}
catch (异常类型1)
{
   // 异常类型1的处理程序
}
catch (异常类型2)
{
   // 异常类型2的处理程序
}
// ……
catch (异常类型n)
{
   // 异常类型n的处理程序
}
```

由try程序块捕获throw抛出的异常，然后依据异常类型运行catch程序块中的异常处理程序。catch程序块顺序可以是任意的，不过均需要放在try程序块之后。

[例1] C++异常处理示例:

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

enum index{
    underflow,
    overflow
};
```

```
int array_index(int *A, int n, int index);

int main(){
   int *A = new int[10];
   for(int i=0; i<10; i++)
       A[i] = i;
   try{
       cout << array_index(A, 10, 5) << endl;
       cout << array_index(A, 10, -1) << endl;
       cout << array_index(A, 10, 15) << endl;
   }
   catch(index e){
       if(e == underflow){
           cout << "index underflow!" << endl;
           exit(-1);
       }
       if(e == overflow){
           cout << "index overflow!" << endl;
           exit(-1);
       }
   }
   return 0;
}

int array_index(int *A, int n, int index){
    if(index < 0)
        throw underflow;
    if(index > n-1)
        throw overflow;

    return A[index];
}
```

本例展示了一个数组越界的异常捕获程序。array_index函数用于返回数组index下标的数值，如果出现异常则抛出异常。try程序块中的程序语句为可能出现异常情况的语句，catch则为针对异常的处理语句。在程序一开始我们定义了一个全局的枚举类型变量index，并且定义了两个值，分别为underflow和overflow，这两个值作为抛出异常的返回值。当在主函数要求输出越界的数组值时，调用array_index函数，一旦有预定异常抛出，则通过try捕获并根据catch语句针对异常情况作出处理。

在前面我们介绍了new和delete动态分配内存操作符，如果new或new[]不能成功分配所请求的，将会抛出一个bad_alloc异常。在使用new或new[]操作符分配动态内存，可以通过如下方式检测并捕获存储空间分配失败的异常。

[例2] 捕获new、new[] 抛出的异常:

```
int *p;
try{
   p = new int[10];
}
catch(bad_alloc){
   cerr << "allocate failure!" << endl;
   exit(-1);
}
```

在C语言中，异常通常是通过函数返回值获得，但这样一来，函数是否产生异常则需要通过检测函数的返回值才能得知。而在C++中，当函数抛出一个返回值时，即使不用try和catch语句，异常还是会被处理的，系统会自动调用默认处理函数unexpected来执行。

noexcept

noexcept修饰符

noexcept能够用来修饰函数，在函数的后面加上noexcept，表明这个函数不会抛出异常，若是抛出异常程序就会终止。这个特性在C++11中出现不少，主要是为了当程序不应出现异常的地方抛出异常时终止程序，例如delete函数、析构函数默认都是noexcept。

```
void my_exception(){
     throw 1;
}

void my_exception_noexcept() noexcept{
    throw 1;
}

void my_exception_noexcept_false() noexcept(false){
    throw 1;
}

int main(){
    try {
        my_exception();
    }
    catch (...) {
        std::cout << "throw" << std::endl;  // throw
    }
    try {
        my_exception_noexcept();            // terminate
    }
    catch (...) {
        std::cout << "throw" << std::endl;
    try {
        my_exception_noexcept_false();
    }
    catch (...) {
        std::cout << "throw" << std::endl;   // throw
    }
    return 0;
}
```

上面的代码最终会输出throw，若是在函数my_exception后加上noexcept，

void my_exception() noexcept，程序就会直接退出。noexcept还能加上常量表达式参数，例如noexcept(true)，当常量表达式结果为true时，标识该函数不能抛出异常，不然可以抛出异常。noexcept不带参数时默认为true。

noexcept运算符

noexcept还能做为运算符noexcept(expression)，noexcept 运算符不对 expression 求值。若 expression 含有至少一个下列潜在求值的构造则结果为 false：

一、调用无不抛出异常指定的任意类型函数，除非它是常量表达式。

二、throw 表达式。

三、目标类型是引用类型，且转换时须要运行时检查的 dynamic_cast 表达式

四、参数类型是多态类类型的 typeid 表达式

五、全部其余状况下结果是 true 。

```
void test(){}
void test_noexcept()noexcept(true){}
void test_noexcept_false()noexcept(false){  }
class Base{
public:
    virtual void f(){}
};

class Test : public Base { };

int main(){
    std::cout << noexcept(test()) << std::endl;                      \\ false
    std::cout << noexcept(test_noexcept()) << std::endl;             \\ false
    std::cout << noexcept(test_noexcept_false()) << std::endl;       \\ true
    std::cout << noexcept(throw) << std::endl;                       \\ false

    Test test;
    Base& base = test;
    std::cout << noexcept(dynamic_cast<Test&>(base)) << std::endl;     \\ false
    std::cout << noexcept(typeid(base)) << std::endl;                  \\ false
    return0;
}
```

另外noexcept运算符能用于模板，这样函数是否noexcept能够自定义，例如一下示例，定义了fun，其中第一个noexcept是修饰符，第二个为运算符，对模板参数T进行运算。

```
template <typename T> void fun()noexcept(noexcept(T())) {
    throw 1;
}

class Base {
public:
    virtual voidf(){}
};
```

```
class Test : public Base {
public:
    ~Test() noexcept(true) {}
};

class TestFalse : public Base {
public:
    ~TestFalse() noexcept(false) {}
};

int main(int argc, char **argv){
    std::cout << noexcept(TestFalse()) << std::endl;         // false
    std::cout << noexcept(Test()) << std::endl;              // true
    try{
        fun<TestFalse>();
    }catch (...){
        std::cout << "throw" << std::endl;                    // throw
    }try{
        fun<Test>();                                          // terminate
    }catch (...){
        std::cout << "throw" << std::endl;
    }
    getchar();
    return 0;
}
```

C++异常处理（try catch throw）

- 程序运行时常会碰到一些异常情况，例如：做除法的时候除数为 0；
- 用户输入年龄时输入了一个负数；
- 用 new 运算符动态分配空间时，空间不够导致无法分配；
- 访问数组元素时，下标越界；打开文件读取时，文件不存在。

这些异常情况，如果不能发现并加以处理，很可能会导致程序崩溃。

所谓“处理”，可以是给出错误提示信息，然后让程序沿一条不会出错的路径继续执行；也可能是不得不结束程序，但在结束前做一些必要的工作，如将内存中的数据写入文件、关闭打开的文件、释放动态分配的内存空间等。

一发现异常情况就立即处理未必妥当，因为在一个函数执行过程中发生的异常，在有的情况下由该函数的调用者决定如何处理更加合适。尤其像库函数这类提供给程序员调用，用以完成与具体应用无关的通用功能的函数，执行过程中贸然对异常进行处理，未必符合调用它的程序的需要。

此外，将异常分散在各处进行处理不利于代码的维护，尤其是对于在不同地方发生的同一种异常，都要编写相同的处理代码也是一种不必要的重复和冗余。如果能在发生各种异常时让程序都执行到同一个地方，这个地方能够对异常进行集中处理，则程序就会更容易编写、维护。

鉴于上述原因，C++ 引入了异常处理机制。其基本思想是：函数 A 在执行过程中发现异常时可以不加处理，而只是“抛出一个异常”给 A 的调用者，假定为函数 B。

抛出异常而不加处理会导致函数 A 立即中止，在这种情况下，函数 B 可以选择捕获 A 抛出的异常进行处理，也可以选择置之不理。如果置之不理，这个异常就会被抛给 B 的调用者，以此类推。

如果一层层的函数都不处理异常，异常最终会被抛给最外层的 main 函数。main 函数应该处理异常。如果main函数也不处理异常，那么程序就会立即异常地中止。

C++异常处理基本语法

C++ 通过 throw 语句和 try...catch 语句实现对异常的处理。throw 语句的语法如下：throw 表达式;

该语句抛出一个异常。异常是一个表达式，其值的类型可以是基本类型，也可以是类。

try...catch 语句的语法如下：

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

catch 可以有多个，但至少要有一个。

不妨把 try 和其后{}中的内容称作“try块”，把 catch 和其后{}中的内容称作“catch块”。

- try...catch 语句的执行过程是：执行 try 块中的语句，如果执行的过程中没有异常抛出，那么执行完后就执行最后一个 catch 块后面的语句，所有 catch 块中的语句都不会被执行；
- 如果 try 块执行的过程中抛出了异常，那么抛出异常后立即跳转到第一个“异常类型”和抛出的异常类型匹配的 catch 块中执行（称作异常被该 catch 块“捕获”），执行完后再跳转到最后一个 catch 块后面继续执行。

1. 例如下面的程序：

```
#include<iostream>
usingnamespace 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;
}
```

程序的运行结果如下：

```
9 6↙
before dividing.
1.5
after dividing.
finished
```

说明当 n 不为 0 时，try 块中不会抛出异常。因此程序在 try 块正常执行完后，越过所有的 catch 块继续执行，catch 块一个也不会执行。

程序的运行结果也可能如下：

```
9 0↙
before dividing.
catch\(int) -1
finished
```

当 n 为 0 时，try 块中会抛出一个整型异常。抛出异常后，try 块立即停止执行。该整型异常会被类型匹配的第一个 catch 块捕获，即进入catch(int e)

块执行，该 catch 块执行完毕后，程序继续往后执行，直到正常结束。

如果抛出的异常没有被 catch 块捕获，例如，将catch(int e)，改为catch(char e)，当输入的 n 为 0 时，抛出的整型异常就没有 catch 块能捕获，这个异常也就得不到处理，那么程序就会立即中止，try...catch 后面的内容都不会被执行。能够捕获任何异常的 catch 语句

如果希望不论抛出哪种类型的异常都能捕获，可以编写如下 catch 块：

```
catch(...) {
    ...
}
```

这样的 catch 块能够捕获任何还没有被捕获的异常。例如下面的程序：

```
#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(...){
        cout << "catch (...)" << endl;
    }
    cout << "finished" << endl;
    return 0;
}
```

程序的运行结果如下：

```
9 0↙
before dividing.
catch (...)
finished
```

当 n 为 0 时，抛出的整型异常被catch(...)捕获。

程序的运行结果也可能如下：

```
0 6↙
before dividing.
catch (double) -1
finished
```

当 m 为 0 时，抛出一个 double 类型的异常。虽然catch (double)和catch(...)都能匹配该异常，但是catch(double)是第一个能匹配的 catch 块，因此会执行它，而不会执行catch(...)块。

由于catch(...) 能匹配任何类型的异常，它后面的 catch 块实际上就不起作用，因此不要将它写在其他 catch 块前面。

异常的再抛出

1. 如果一个函数在执行过程中抛出的异常在本函数内就被 catch 块捕获并处理，那么该异常就不会抛给这个函数的调用者（也称为“上一层的函数”）；如果异常在本函数中没有被处理，则它就会被抛给上一层的函数。例如下面的程序：

```
#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;
}
```

程序的输出结果如下：

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

CountTa 函数抛出异常后自行处理，这个异常就不会继续被抛给调用者，即 main 函数。因此在 main 函数的 try 块中，CountTax 之后的语句还能正常执行，即会执行f = Devide(3, 0);。

第 35 行，Devide 函数抛出了异常却不处理，该异常就会被抛给 Devide 函数的调用者，即 main 函数。抛出此异常后，Devide 函数立即结束，第 14 行不会被执行，函数也不会返回一个值，这从第 35 行 f 的值不会被修改可以看出。

Devide 函数中抛出的异常被 main 函数中类型匹配的 catch 块捕获。第 38 行中的 e 对象是用复制构造函数初始化的。

如果抛出的异常是派生类的对象，而 catch 块的异常类型是基类，那么这两者也能够匹配，因为派生类对象也是基类对象。

虽然函数也可以通过返回值或者传引用的参数通知调用者发生了异常，但采用这种方式的话，每次调用函数时都要判断是否发生了异常，这在函数被多处调用时比较麻烦。有了异常处理机制，可以将多处函数调用都写在一个 try 块中，任何一处调用发生异常都会被匹配的 catch 块捕获并处理，也就不需要每次调用后都判断是否发生了异常。

1. 有时，虽然在函数中对异常进行了处理，但是还是希望能够通知调用者，以便让调用者知道发生了异常，从而可以作进一步的处理。在 catch 块中抛出异常可以满足这种需要。例如：

```
#include<iostream>
#include<string>

using namespace std;

int CountTax(int salary)
{
    try{
        if(salary <0)
            throw string("zero salary");
        cout << "counting tax" << endl;
    }
```

```
    catch(string s ){
        cout << "CountTax error : " << s << endl;
        throw;    // 继续抛出捕获的异常
    }
    cout << "tax counted " << endl;
    return salary *0.15;
}

int main()
{
    double f = 1.2;
    try{
        CountTax(-1);
        cout << "end of try block" << endl;
    }
    catch(string s){
        cout << s << endl;
    }
    cout << "finished" << endl;
    return 0;
}
```

程序的输出结果如下：

```
CountTax error:zero salary
zero salary
finished
```

第 14 行的throw;没有指明抛出什么样的异常，因此抛出的就是 catch 块捕获到的异常，即 string("zero salary")。这个异常会被 main 函数中的 catch 块捕获。

函数的异常声明列表

为了增强程序的可读性和可维护性，使程序员在使用一个函数时就能看出这个函数可能会抛出哪些异常，C++ 允许在函数声明和定义时，加上它所能抛出的异常的列表，具体写法如下：void func() throw (int, double, A, B, C);

或void func() throw (int, double, A, B, C){...}

上面的写法表明 func 可能抛出 int 型、double 型以及 A、B、C 三种类型的异常。异常声明列表可以在函数声明时写，也可以在函数定义时写。如果两处都写，则两处应一致。

如果异常声明列表如下编写：void func() throw ();

则说明 func 函数不会抛出任何异常。

一个函数如果不交待能抛出哪些类型的异常，就可以抛出任何类型的异常。

函数如果抛出了其异常声明列表中没有的异常，在编译时不会引发错误，但在运行时， Dev C++ 编译出来的程序会出错；用 Visual Studio 2010 编译出来的程序则不会出错，异常声明列表不起实际作用。

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 类派生而来的。常用的几个异常类如图 1 所示。

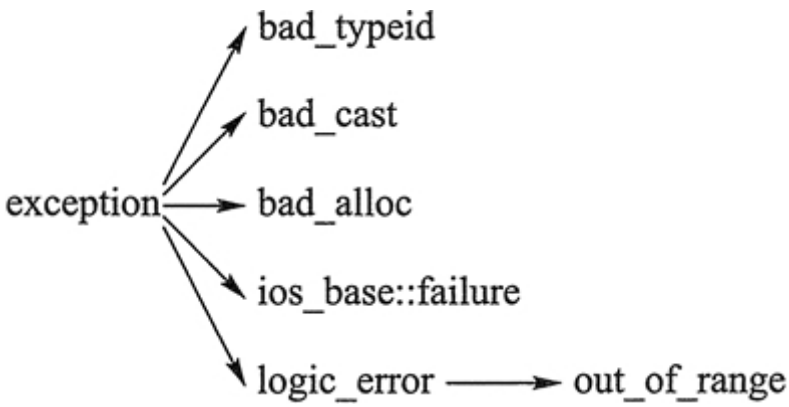

图1：常用的异常类

bad_typeid、bad_cast、bad_alloc、ios_base::failure、out_of_range 都是 exception 类的派生类。C++ 程序在碰到某些异常时，即使程序中没有写 throw 语句，也会自动抛出上述异常类的对象。这些异常类还都有名为 what 的成员函数，返回字符串形式的异常描述信息。使用这些异常类需要包含头文件 stdexcept。

下面分别介绍以上几个异常类。本节程序的输出以 Visual Studio 2010为准，Dev C++ 编译的程序输出有所不同。

1) bad_typeid

使用 typeid 运算符时，如果其操作数是一个多态类的指针，而该指针的值为 NULL，则会抛出此异常。

2) 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。程序示例如下：

```
#include<iostream>
#include<stdexcept>
usingnamespace 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;
}
```

程序的输出结果如下：

```
Bad dynamic_cast!
```

在 PrintObj 函数中，通过 dynamic_cast 检测 b 是否引用的是一个 Derived 对象，如果是，就调用其 Print 成员函数；如果不是，就抛出异常，不会调用 Derived::Print。

3) 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。程序示例如下：

```
#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;
}
```

程序的输出结果如下：

```
bad allocation
ios_base::failure
```

在默认状态下，输入输出流对象不会抛出此异常。如果用流对象的 exceptions 成员函数设置了一些标志位，则在出现打开文件出错、读到输入流的文件尾等情况时会抛出此异常。此处不再赘述。

4) 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，则会抛出此异常。例如：

```
#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;
}
```

程序的输出结果如下：

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

如果将v.at(100)换成v[100]，将s.at(100)换成s[100]，程序就不会引发异常（但可能导致程序崩溃）。因为 at 成员函数会检测下标越界并抛出异常，而 operator[] 则不会。operator [] 相比 at 的好处就是不用判断下标是否越界，因此执行速度更快。