2020/05/27_C++_第16课_异常(Exception)、文件流

笔记本: C++
创建时间: 2020/5/27 星期三 15:35
作者: ileemi
标签: C++文件流, C++异常处理

- 前言
- catch 在匹配过程中的类型转换
- 赋值运算符在类中使用及一些区别
- 异常
  - 返回值示例
  - C++ 异常
  - 程序中有异常错误，但是没有使用 try catch做对应的的处理
  - 异常错误代码的匹配
  - 程序抛出异常后，后面的代码是否继续执行
  - try块内的局部对象会调用析构
  - 在类的构造和析构中抛出异常
  - 异常是可以嵌套的
  - 抛出类对象
- 文件流（文件操作）

前言

异常（Exception）：为解决运行时错误而引入的。运行时错误是指程序在运行期间发生的错误，例如除数为 0、内存分配失败、数组越界、文件不存在等。

语法：

```
try{
    // 可能抛出异常的语句}
catch(exceptionType variable)
{
    //exceptionType variable：catch可以处理的异常类型
    // 处理异常的语句
}
```

注意：

- 发生异常时必须将异常明确地抛出，try 才能检测到；如果不抛出来，即使有异常 try 也检测不到
- 发生异常后，程序的执行流会沿着函数的调用链往前回退，直到遇见 try 才停止。在这个回退过程中，调用链中剩下的代码（所有函数中未被执行的代码）都

会被跳过，没有执行的机会了

C++异常类型以及多级catch匹配

catch 在匹配过程中的类型转换

C/C++ 中存在多种多样的类型转换，以普通函数（非模板函数）为例，发生函数调用时，如果实参和形参的类型不是严格匹配，那么会将实参的类型进行适当的转换，以适应形参的类型，这些转换包括：

- 算数转换：例如 int 转换为 float，char 转换为 int，double 转换为 int 等。
- 向上转型：也就是派生类向基类的转换，请猛击《C++向上转型（将派生类赋值给基类）》了解详情。
- const 转换：也即将非 const 类型转换为 const 类型，例如将 char * 转换为 const char* 。
- 数组或函数指针转换：如果函数形参不是引用类型，那么数组名会转换为数组指针，函数名也会转换为函数指针。
- 用户自定的类型转换。

赋值运算符在类中使用及一些区别

拷贝构造和赋值运算符重载的使用以及调用的时机

```
#include <iostream>

class CInteger
{
public:
    CInteger(int nVal)
    {
        m_nVal = nVal;
    }
    // 拷贝构造
    CInteger(const CInteger& obj)
    {
        m_nVal = obj.m_nVal;
    }
    // 当程序中没有声明定义对应的赋值运算符的时候，编译器会默认生成

    //CInteger& operator=(int nVal)
    //{
    //  m_nVal = nVal;
    //  return *this;
    //}
```

```
    CInteger& operator=(const CInteger& obj)
    {
        m_nVal = obj.m_nVal;
        return *this;
    }

    int GetVal() const
    {
        return m_nVal;
    }
    void SetVal(int val)
    {
        m_nVal = val;
    }
private:
    int m_nVal;
};
int main()
{
    CInteger obj(999);
    // 调用拷贝构造， 赋初值语句，相当于 obj1.CInteger(obj);
    // 生命周期开始，要进行初始化
    CInteger obj1 = obj;
    CInteger obj2(666);

    // 调用 = 号 运算符重载，赋值语句，展开后相当于 obj2.operator=(obj);
    // 生命周期中间
    obj2 = obj;
    /*
    在给obj2 对象赋值的时候，给的是类对象进行的赋值，但是上面的类中
    在运算符重载的时候，为其数据成员赋值的是 int 类型的数值，编译器没有进行调用
    这里编译器其实自动生成了一个 重载函数（默认的重载函数）
    */

    /*
    注意观察，= 符号的位置，是在类对初始化前，还是类对象初始化后进行调用
    在初始化前，使用，调用拷贝构造
    在类对象初始化后，在次使用，就会调用重载的赋值语句
    */
    return 0;
}
```

异常

异常：意料之外的情况
程序中的异常： 异常 --> 错误

常见的错误处理的两种方法：

1、使用全局错误码（C语言库，Windows操作系统）
示例：

```
int g_nError;
   0  成功
   1  加法溢出
   2  空指针
   3  除法溢出
   4  乘法溢出

   ...(出错的错误情况等)

根据错误码，做出相应的处理

缺点：随着代码量的增多，出错的情况也就随之增多，维护成本随之增加
```

2、返回值表示

示例：

```
  0     成功
  1     各种错误信息（例：加法溢出等）
```

返回值示例

缺点：当程序函数调用嵌套很深的时候，返回值错误会将出现错误信息的模块往上传，传递到使用者调用的模块中，也就是说，出现错误问题，使用者不方便定位错误。

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

// 两数相加
int Add(int nVal1, int nVal2, int& nResult)
{
    nResult = nVal1 + nVal2;

    // 两个整数相加，结果为负，加法溢出
    if (nVal1 > 0 && nVal2 > 0 && nResult < 0)
    {
        // cout << "加法溢出" << endl;
```

```
        // exit(0);
        return 1;
    }
    return 0;
}

// 两数相减
int Sub(int nVal1, int nVal2, int& nResult)
{
    nResult = nVal1 - nVal2;

    // 正数减负数，结果为负， 减法溢出
    if (nVal1 > 0 && nVal2 < 0 && nResult < 0)
    {
        // cout << "减法溢出" << endl;
        // exit(0);
        return 1;
    }
    return 0;
}

int  Func(int nVal1, int nVal2)
{
    int nResult1 = 0;
    int nRet1 = Add(nVal1, nVal2, nResult1);
    // nRet1 = -2
    int nResult2 = 0;
    int nRet2 = Sub(nVal1, nVal2, nResult1);
    // nRet2 = 0
    return nResult1 * nResult2;
}

int main()
{
    Func(0x7FFFFFFF, 0x7FFFFFFF);
    return 0;
}
```

C++ 异常

关键字：

- **抛异常：throw**
- **接异常：catch**

C++ 异常处理的流程：
抛出（throw）--> 检测（try） --> 捕获（catch）

相对于返回值传出错误来说，C++的异常，处理错误的基本做法就是，哪里出错，哪里返回错误信息，顺着调用链往上传递错误信息。所有调用链都可以处理该错误信息，直到第一次调用哪里。

基本用法：

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

// 两数相加
int Add(int nVal1, int nVal2)
{
    int nResult = nVal1 + nVal2;

    // 两个整数相加，结果为负，加法溢出
    if (nVal1 > 0 && nVal2 > 0 && nResult < 0)
    {
        throw 1;
    }
    return nResult;
}


// 两数相减
int Sub(int nVal1, int nVal2)
{
    int nResult = nVal1 - nVal2;

    // 正数减负数，结果为负， 减法溢出
    if (nVal1 > 0 && nVal2 < 0 && nResult < 0)
    {
        throw 2;
    }
    return nResult;
}

int  Func(int nVal1, int nVal2)
{
    // 哪个调用方法的函数都可以通过 try catch 来捕获错误信息
    /*
    try
    {
        int nResult1 = Add(nVal1, nVal2);
        int nResult2 = Sub(nVal1, nVal2);
        return nResult1 * nResult2;

    }
```

```
    catch (int nError)
    {
        //cout << nError << endl;
        switch (nError)
        {
        case 1:
            cout << "加法溢出，错误码：" << nError << endl;
            break;
        case 2:
            cout << "减法溢出，错误码：" << nError << endl;
            break;
        }
    }
    return 0;
    */

    int nResult1 = Add(nVal1, nVal2);
    int nResult2 = Sub(nVal1, nVal2);
    return nResult1 * nResult2;
}

int main()
{
    // 同样 main函数也可以捕获错误信息
    try
    {
        Func(0x7FFFFFFF, 0x80000000);
    }
    catch (int nError)
    {
        //cout << nError << endl;
        switch (nError)
        {
        case 1:
            cout << "加法溢出，错误码：" << nError << endl;
            break;
        case 2:
            cout << "减法溢出，错误码：" << nError << endl;
            break;
        }
    }
    return 0;

    //Func(0x7FFFFFFF, 0x7FFFFFFF);
    //Func(0x7FFFFFFF, 0x80000000);
```

```
    return 0;
}
```

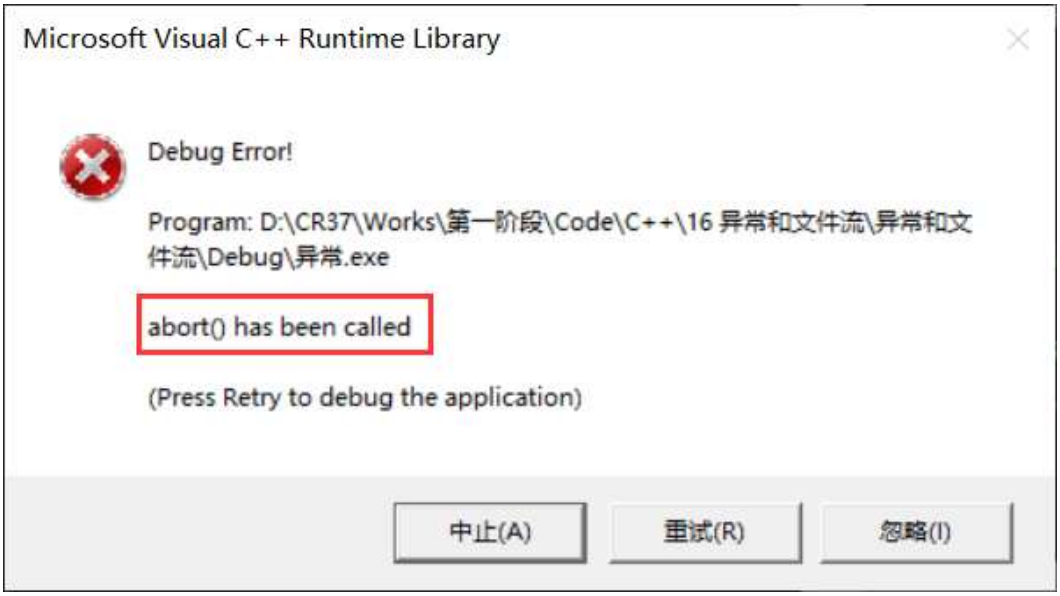
程序中有异常错误，但是没有使用 try catch做对应的的处理

如果异常没有被接收， 最终会调用abort函数， 退出程序（debug还会弹框）

示例代码：

```
int Add(int nVal1, int nVal2)
{
    int nResult = nVal1 + nVal2;

    //两个整数相加，结果为负， 加法溢出
    if (nVal1 > 0 && nVal2 > 0 && nResult < 0)
    {
        throw 1;
    }
    return nResult;
}

int Sub(int nVal1, int nVal2)
{
    int nResult = nVal1 - nVal2;

    //正数减负数，结果为负， 减法溢出
    if (nVal1 > 0 && nVal2 < 0 && nResult < 0)
    {
        throw 2;
    }
    return nResult;
}

int Func(int nVal1, int nVal2)
{
    int nResult1 = Add(nVal1, nVal2);
    int nResult2 = Sub(nVal1, nVal2);
    return nResult1 * nResult2;
}

int main()
{
    //abort();

    Func(0x7fffffff, 0x80000000);
    printf("hello world");
```

```
    return 0;
}
```



异常错误代码的匹配

程序中的异常错误代码中不会做隐式转换， 其错误信息的类型必须精确匹配， try下可以出现搓个catch，如果没有与之匹配的 catch 错误代码的类型，程序终止，**显示未经处理的异常。**

示例代码：

```
int Add(int nVal1, int nVal2)
{
    int nResult = nVal1 + nVal2;

    //两个整数相加，结果为负， 加法溢出
    if (nVal1 > 0 && nVal2 > 0 && nResult < 0)
    {
        throw 1;
    }
    return nResult;
}

int Sub(int nVal1, int nVal2)
{
    int nResult = nVal1 - nVal2;

    //正数减负数，结果为负， 减法溢出
    if (nVal1 > 0 && nVal2 < 0 && nResult < 0)
    {
        throw 9.99;
    }
```

```
    return nResult;
}

int Func(int nVal1, int nVal2)
{
    int nResult1 = Add(nVal1, nVal2);
    int nResult2 = Sub(nVal1, nVal2);
    return nResult1 * nResult2;
}

int main()
{
    try
    {
        Func(0x7fffffff, 0x80000000);
        printf("hello world");
    }
    catch (int nError)
    {
        cout << "错误码: " << nError << endl;
    }
    /*
    如果此处没有下面的catch语句，上面对应的 错误代码 9.99，
    catch 就受不到，程序会终止，显示未经处理的异常
    */
    catch (double dblError)
    {
        cout << "错误码: " <<dblError << endl;
    }
    return 0;
}
```

程序抛出异常后，后面的代码是否继续执行

从抛出异常处， 跳转到catch， catch后面的代码继续执行

代码示例:

```
int main()
{
    try
    {
        int nVal = 666;
        throw 1;
        /*
        throw 1;
```

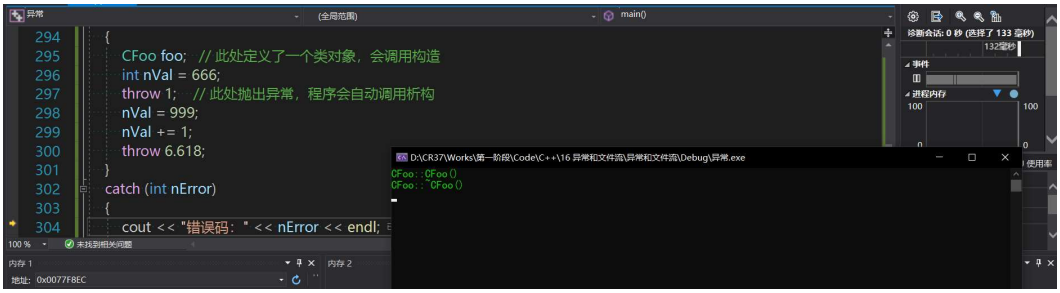
```
        这里抛出异常，  后面的代码不会被执行，
        说明上面的代码就有问题，防止后面代码使用有问题的代码
        */
        nVal = 999;
        nVal += 1;
        throw 6.618;
    }
    catch (int nError)
    {
        cout << "错误码: " << nError << endl;
        // 此处的代码可以继续执行
        cout << "Hello World" << endl;
    }
    catch (double dblError)
    {
        cout << "错误码: " << dblError << endl;
        cout << "Hello World" << endl;
    }

    // 此处的代码可以继续执行
    cout << "Hello World" << endl;
    return 0;
}
```

try块内的局部对象会调用析构

代码示例:

```
int main()
{
    try
    {
        CFoo foo;    // 此处定义了一个类对象，会调用构造
        int nVal = 666;
        throw 1;     // 此处抛出异常，程序会自动调用析构
        nVal = 999;
        nVal += 1;
        throw 6.618;
    }
    catch (int nError)
    {
        cout << "错误码: " << nError << endl;
        // 此处的代码可以继续执行
        cout << "Hello World" << endl;
    }
```

```
    catch (double dblError)
    {
        cout << "错误码：" << dblError << endl;
        cout << "Hello World" << endl;
    }

    // 此处的代码可以继续执行
    cout << "Hello World" << endl;
    return 0;
}
```



在类的构造和析构中抛出异常

构造函数抛出异常：不会调用析构，直接进入对应的 catch 块
析构函数抛出异常：程序终止不会走catch块， 直接调用abort函数
代码示例：

```
class CFoo
{
public:
    CFoo()
    {
        cout << "CFoo::CFoo()" << endl;
        //throw 1; // 不会调用析构，直接进入对应的 catch 块
    }
    ~CFoo()
    {
        cout << "CFoo::~CFoo()" << endl;
        throw 1;    // 不会走catch块， 直接调用abort函数
    }
};

int main()
{
    try
    {
        CFoo foo;   // 此处定义了一个类对象，会调用构造
    }
```

```
    catch (int nError)
    {
        cout << "错误码: " << nError << endl;
        // 此处的代码可以继续执行
        cout << "Hello World" << endl;
    }
    catch (double dblError)
    {
        cout << "错误码: " << dblError << endl;
        cout << "Hello World" << endl;
    }

    // 此处的代码可以继续执行
    cout << "Hello World" << endl;
    return 0;
}
```

异常是可以嵌套的

catch (...) --> 可以接受任意类型的异常错误代码

代码示例:

```
int main()
{
    try
    {
        // 异常可以嵌套
        try
        {
            throw 'a'; // 内层接不住， 外层继续接异常
        }
        catch (double dblError)
        {
            cout << dblError << endl;
        }
    }
    catch (int nError)
    {
        cout << "错误码: " << nError << endl;
        // 此处的代码可以继续执行
        cout << "Hello World" << endl;
    }
    catch (double dblError)
    {
        cout << "错误码: " << dblError << endl;
```

```
        cout << "Hello World" << endl;
    }
    // 用于程序体面的退出
    catch (...)
    {
        cout << "接受任意类型的异常错误代码" << endl;
    }

    // 此处的代码可以继续执行
    cout << "Hello World" << endl;
    return 0;
}
```

抛出类对象

简单使用:

```
#include <iostream>
using namespace std;
class CExceptionInfo
{
public:
    CExceptionInfo(int nVal) : m_nVal(nVal)
    {
    }
    void SetVal(int nVal)
    {
        m_nVal = nVal;
    }
    int GetVal() const
    {
        return m_nVal;
    }
private:
    int m_nVal;
};
int main()
{
    try
    {
        throw CExceptionInfo(99);
    }
    catch (const CExceptionInfo& obj)
    {
        cout << obj.GetVal() << endl;
```

```
    }
    return 0;
}
```

一般不会像上面的简单的是使用，一般会输出一些有用的程序信息，文件名，出错行号，所属函数、出错原因等。

代码展示：

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

// 父类
class CExceptionInfo
{
public:
    // 带参构造
    CExceptionInfo(string strFileName, string strFuncName, string
strError, int nLineNumber):
        m_strFileName(strFileName),
        m_strFuncName(strFuncName),
        m_strError(strError),
        m_nLineNumber(nLineNumber)
    {
    }
    string GetFileName() const { return m_strFileName; }
    void SetFileName(string val) { m_strFileName = val; }
    string GetFuncName() const { return m_strFuncName; }
    void SetFuncName(string val) { m_strFuncName = val; }
    string GetError() const { return m_strError; }
    void SetError(string val) { m_strError = val; }
    int GetLineNumber() const { return m_nLineNumber; }
    void SetLineNumber(int val) { m_nLineNumber = val; }
private:
    string  m_strFileName;  // 文件名
    string  m_strFuncName;  // 函数名
    string  m_strError; // 错误信息
    int     m_nLineNumber;  // 错误行数
};

/*
* 当多个类需要处理不同情况的数据是，可以先定义一个父类，让它子类去继承
该父类
* 使其拥有相同的功能
*/
```

```
// 子类
class CAddOverFlowException : public CExceptionInfo
{
public:
    CAddOverFlowException(string strFileName, string strFuncName, string
strError, int nLineNumber):
        CExceptionInfo(strFileName, strFuncName, "展示加法溢出" ,
nLineNumber)
    {
    }
};

void Func()
{
    throw CAddOverFlowException(__FILE__, "Func", "展示加法溢出",
__LINE__);
    throw CExceptionInfo(__FILE__, "Func", "展示类对象抛出错误",
__LINE__);
}
int main()
{
    try
    {
        Func();
    }
    // 父类引用(指针) 接受子类对象
    catch (const CExceptionInfo& obj)
    {
        cout << obj.GetFileName() << endl
            << obj.GetFuncName() << endl
            << obj.GetError() << endl
            << obj.GetLineNumber()
            <<endl;
    }
    return 0;
}
```

文件流（文件操作）

文件操作，抽象出来的操作方法在不同的平台，不同的系统，不同的框架是通用的。

文件的一般操作：
打开（只读、只写、二进制、等等各种标志）、读、写、文件大小、文件指针、关闭文件

ofstream -->写
ifstream -->读

fstream -->继承（ofstream、ifstream）

使用示例：

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

int main()
{
#if 0
    ofstream ofs;
    ofs.open("ReadMe.txt", ios_base::out | ios_base::app);
    ofs.seekp(0, ios_base::beg); //将文件指针移动到文件头部
    ofs << "hello test"
        << " " << 4
        << " " << 4.5
        << " " << 3.141592653f
        << endl;
    ofs.close();
#endif // 0
#if 0
    fstream fs;
    fs.open("Readme.txt", ios_base::out | ios_base::in);
    fs.seekp(0, ios_base::beg); //将文件指针移动到文件头部
    fs << "hello test"
        << " " << 4
        << " " << 4.5
        << " " << 3.141592653f
        << endl;

    fs.close();
#endif // 0

    // 推荐使用 fstream
    fstream fs;
    fs.open("Readme.txt", ios_base::out | ios_base::in |
ios_base::binary);

    char szBuff[] = { "hello world fstream xxxxxxx" };
    fs.write(szBuff, sizeof(szBuff));

    fs.close();
    return 0;
}
```