异常

比如函数传进来一个空指针，或者并不是我们希望的类型，这个时候函数就处理不下去了，就会把一个错误报出去。

解决办法：

1. assert() 断言终止程序运行

缺点是没有给别人修复的机会，另外只有程序员看得懂报错信息。

2. 错误码机制

比如说`sqrt(-1)`，传出一个错误码。优点是可读性好（可视宏）

但是有的时候错误发生的值域不好预测。这个时候可以使用**全局错误码**

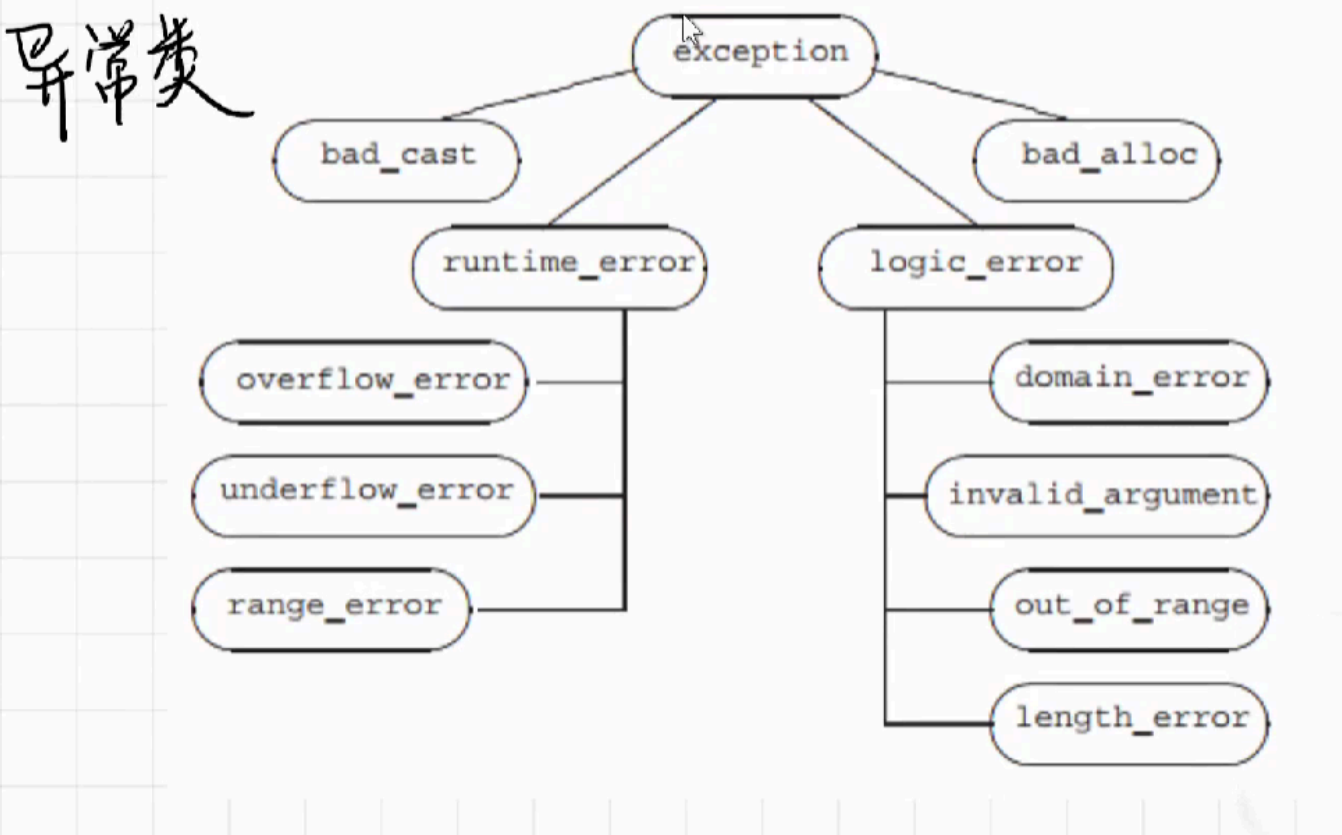
全局错误码**GetLastError**函数（windows）：每当执行一个可能会出错的函数之后，你就立刻调用一下这个函数来检验错误码，如果错误码改变了，说明发生了异常，你就要去检查一下。

但是在多线程的场景下，这样会导致代码冗长复杂。

3. C++自带异常处理机制

通过try...catch这样的语法，在可能出问题的地方对代码进行包装，当代码触发异常的时候，函数会在栈上逐个地寻找出现异常的代码。如果找到了就调用处理异常函数，没找到就一层层向上汇报（raise/throw）。就好像你出现了工作搞不定交给领导，领导搞不定交给领导的领导。如果这个问题抛到最上方还是解决不了的话，这个异常就会被终结掉。

> 还有一种if ...else的异常处理机制，通常在小项目，但是在大项目的时候会把发现问题和解决问题分离开。就是即使你发现了这个异常，但是那块内容不是你实现的。你没法去处理不由你决定。



bad_cast：转换时错误
runtime_error：运行时错误
logic_error：逻辑错误
bad_alloc：分配内存时错误。

我们可以从系统的异常类中继承。

overflow_error：向上溢出
range_error：范围溢出
out_of_range：数组越界

```
#include<stdexcept>
#include<exception>
// C++提供的异常都是从这俩头文件出来的
```

下面是一个异常处理的代码示例：

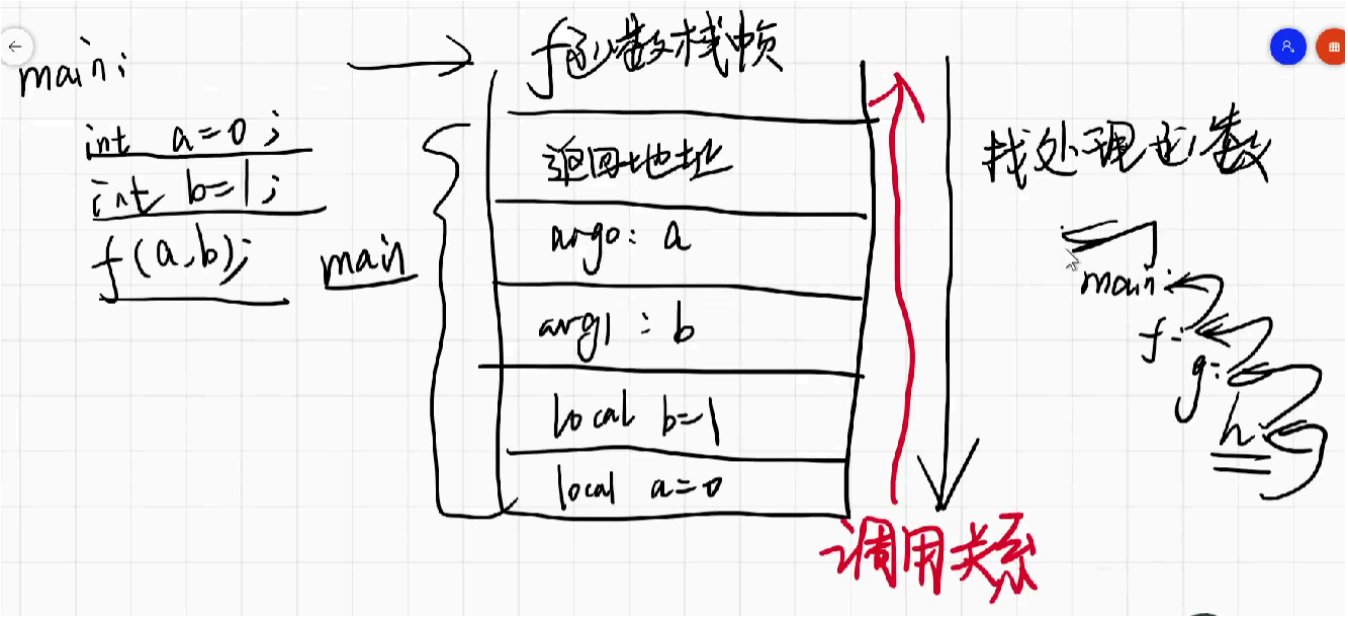
```cpp
#include<iostream>
#include<cstring>
using namespace std;

class MyException : public runtime_error {
public:
    MyException(const string &s) : runtime_error(s) {
        cout << "MyException ctor" << endl;
    }
    const char *what() const noexcept override {
        return "213";
        // noexcept 关键字表示这个函数本身不会再抛出异常了，这是为了防止无线递归的情况。如果还是抛出了异常
    }
};
int main() {

    try {
        // 可能含有异常的代码
        cout << "throwing" << endl;
        throw (MyException("hello world!"));
        throw ((string)"dsafdavd");
        throw (1);
        cout << "will not excute" << endl; // unreachable code
    } catch (runtime_error &re) {
        // 一般用引用来接，一层层传
        cout << re.what() << endl;
    } catch (string &e) {
        cout << "caught an string" << endl;
    } catch (int &e) {
        cout << e << " caught an integer" << endl;
    } catch(...) { //捕获全部异常
        cout << "exception caught" << endl;
    }

    cout << "continue from try..." << endl;
    return 0;
}
```

```
throwing
MyException ctor
213
continue from try...
```

异常/函数调用

异常的实现一般都是放在栈上的。而异常处理查找的顺序和函数调用的顺序是相反的。那么函数调用的过程到底是怎么发生的？

考虑f(a,b)，对大多数函数来说，函数是先把参数放在栈上，再往下，新开辟一块栈的空间，而这块空间实际上才是f函数调用的时候所用的空间每个函数用的区域叫做**栈帧**，下边这块是外层函数代码所占的空间，栈帧之间应该是互不影响的。



异常处理机制保证了，**分配在站上的变量可被回收**，

分配在函数内部，没有使用new malloc堆式分配的在栈上的变量都是在栈上的。**他们的析构函数都会自动地调用**。

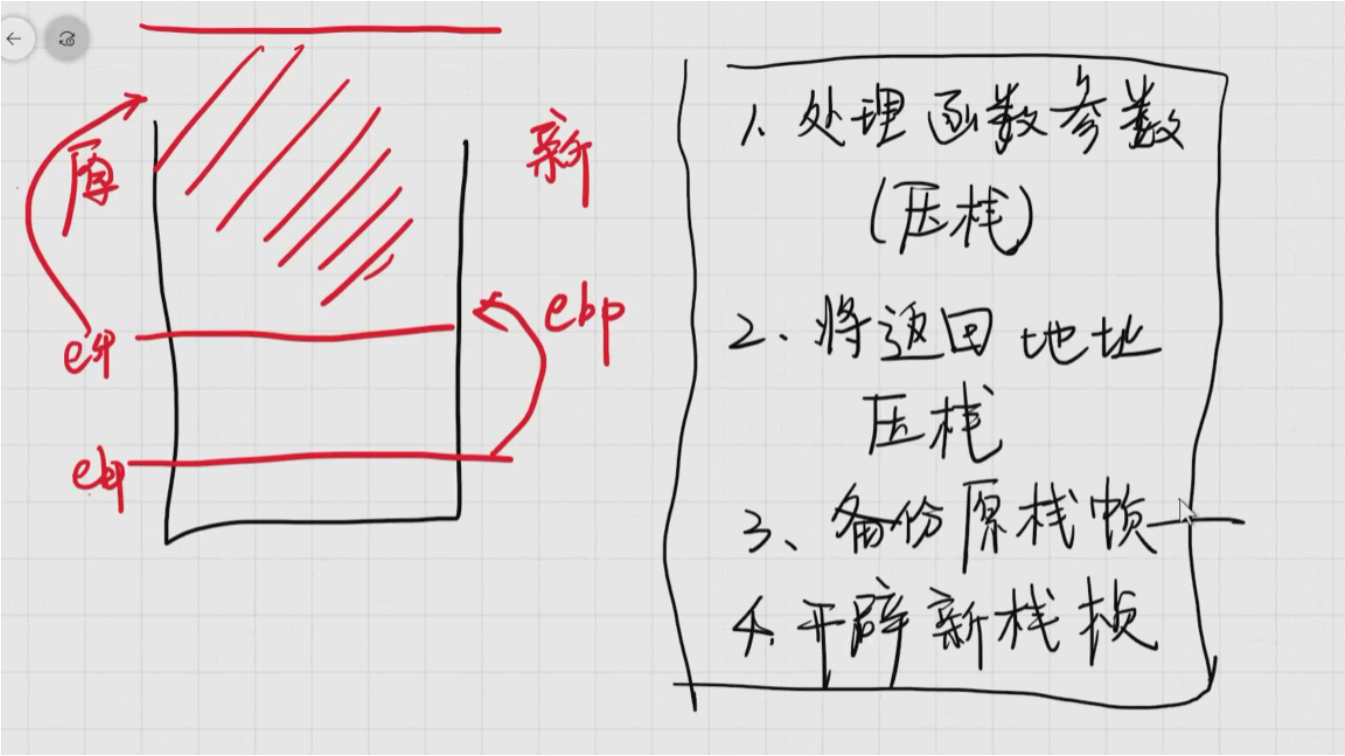
在堆上申请内存：

```
Helper *h = new Helper[3]
```

主动调用析构：

```
delete[] h;
```

函数调用过程



vscode怎么查看汇编？

```
-exec disassemble /m main
```

```cpp
using namespace std;
int main() {

    try {
        // 可能含有异常的代码
        cout << "throwing" << endl;
        throw ((string)"dsafdavd");
        throw (1);
        cout << "will not excute" << endl; // unreachable code
    } catch (string &e) {
        cout << "caught an string" << endl;
    } catch (int &e) {
        cout << e << " caught an integer" << endl;
    } catch(...) { //捕获全部异常
        cout << "exception caught" << endl;
    }

    cout << "continue from try..." << endl;
    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
Loaded 'C:\WINDOWS\SysWOW64\msvcrt.dll'. Symbols loaded.
Loaded 'D:\Appgallery\MinGW\bin\libgcc_s_dw2-1.dll'. Symbols loaded.
Loaded 'D:\Appgallery\MinGW\bin\libstdc++-6.dll'. Symbols loaded.
throwing
Execute debugger commands using "-exec <command>", for example "-exec info registers" will list registers in us
-exec disassemble /m main
Dump of assembler code for function main():
6       int main() {
   0x00401430 <+0>:     lea     ecx,[esp+0x4]
```

或者去这个网站：compile explorer