USTC

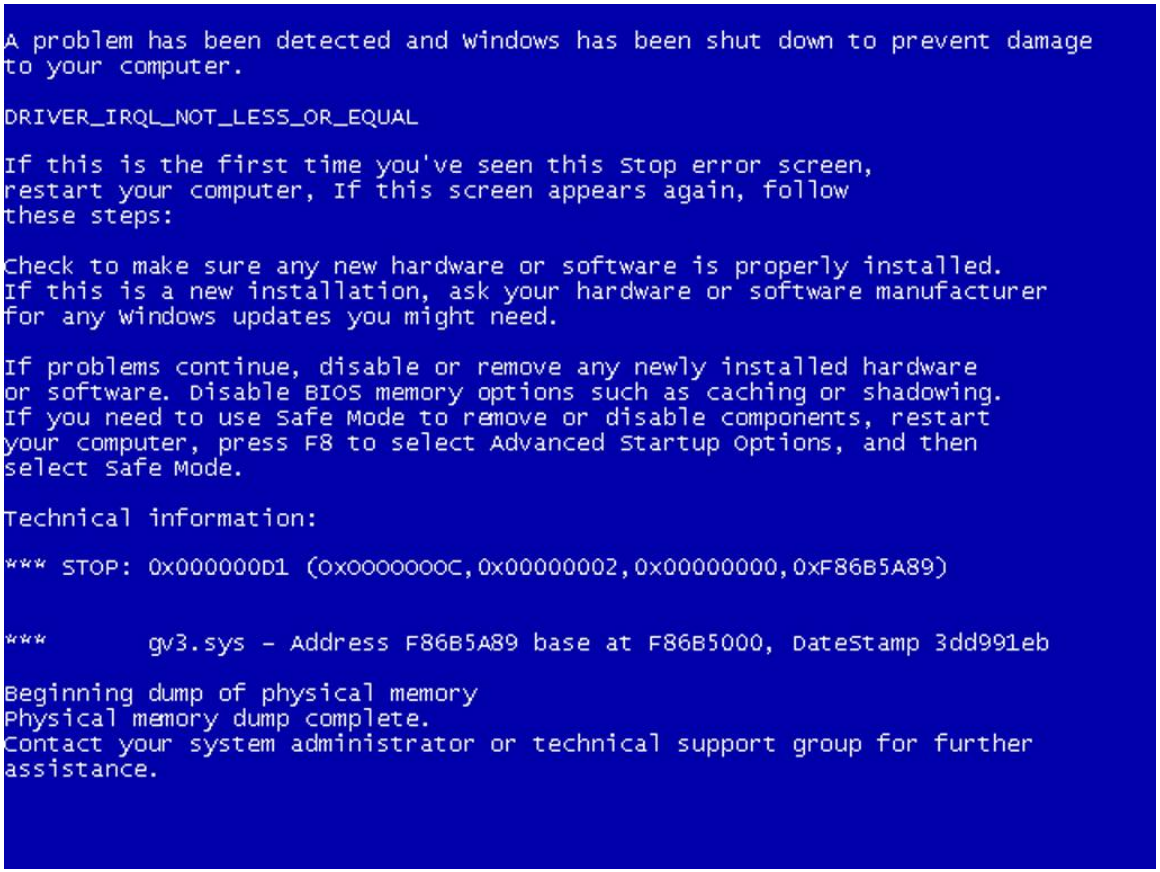
Chapter 5
Errors

王子磊 (Zilei Wang)
Email: zlwang@ustc.edu.cn
http://vim.ustc.edu.cn

Overview

- 编程时我们不得不处理各种错误，以期望达到正确性的目的

- Overview
  - 错误类型
  - 参数检查
    - 错误报告
    - 错误检查
    - 异常
  - 调试
  - 测试



错误 (Errors)

- “我已经意识到从现在开始我的大部分时间将花在查找和纠正自己的错误上”
  - Maurice Wilkes, 1949
- 当我们编写程序时，错误是自然不可避免的，现在的问题是：我们如何处理错误？
  - 精心组织软件结构以减少错误
  - 消除大部分程序错误
    - 调试 Debugging
    - 测试 Testing
  - 确保余下的错误是不重要的
- 在开发正式软件时，约95%的时间都是放在如何避免、查找和纠正错误上
  - 对小程序来说，你可以把错误处理做的好一些
    - 如果你很马虎，也可能做的更糟

程序

除非特别声明，我们假定你的程序：

1. 对于所有合法的输入应输出正确结果
2. 对于所有非法输入应输出错误消息
3. 不需要关心硬件故障
4. 不需要担心系统软件故障
5. 发现一个错误后，允许程序终止

- 对初学者来说，可认为3, 4, 5 是已经成立的(在实际程序中这些是需要认真考虑的)，因此，1和2是我们主要考虑的范畴

错误来源

- 规划不周
  - “我们想要程序做什么?”
  - 事先规划好，充分检查所有的“死角”
- 不完备的程序
  - “实际上，我们不能100%的达到*理想目标*”
- 意外的参数
  - “**sqrt()** 没有假定用参数**-1** 的调用情况”
- 意外的输入
  - “假定用户输入一个整数”
- 代码太简单，没有按照我们期望的那样运行(逻辑错误)
  - “那么修正它!”

错误类型

- 编译时错误 Compile-time errors
  - 语法错误
  - 类型错误
- 连接时错误 Link-time errors
- 运行时错误 Run-time errors
  - 计算机检查 (crash)
  - 程序库检查(exceptions)
  - 用户代码检查
- 逻辑错误 Logic errors
  - 程序员负责检查
  - 代码能够正常运行，但产生的结果不是期望的

注意：警告类错误也要排除！

检查输入

- 在开始使用一个输入值之前，检查它是否满足要求 (expectations/requirements)
  - 函数参数
  - 输入的数据 (e.g. **istream**)

错误的函数参数

- 编译器能够帮助你：
  - 匹配参数的个数和类型

```
int area(int length, int width)
{
    return length*width;
}

int x1 = area(7);                    // error: wrong number of arguments
int x2 = area("seven", 2);           // error: 1st argument has a wrong type
int x3 = area(7, 10);                // ok
int x5 = area(7.5, 10);   // ok, but dangerous: 7.5 truncated to 7;
                                     //      most compilers will warn you
int x = area(10, -7);                // this is a difficult case:
                                     // the types are correct,
                                     // but the values make no sense
```

错误的函数参数

- 那么，如何处理： **int x = area(10, -7);**
- 可能的解决方案
  - 什么都不做
    - 大部分情况下，这不是一个满意的答案
  - 调用者检查错误——多处
    - 但难以系统性地进行处理
  - 函数本身(被调用者)负责检查——一处
    - 返回一个“错误码”(不通用，可能有问题)
    - 设置一个错误标识(不通用，不要这样用)
    - 抛出一个异常exception
- 注意：有时我们不能修改一个函数，尽管它采用的错误处理不是我们期望的
  - 其他人写的，我们可能不能改或不想改这些代码

错误的函数参数——函数检查

- 为什么需要考虑这个问题?
  - 你想让你的程序正确执行
  - 典型情况下，一个函数的作者不能控制它被如何调用
    - 在手册或注释中写上“按这种方式使用”并不是一个解决方法 – 人们通常不读手册
  - 通常，函数开始处是进行检查的合适位置
    - 在计算开始变得“不知所措”之前
- 什么时候应该考虑?
  - 除非你有足够理由，否则参数检查应尽量在函数内部完成
  - 尽量测试每一个函数

如何报告一个错误？

- 返回一个错误码(可能是有问题的)

```
int area(int length, int width)    // return a negative value for bad input
{
    if(length <=0 || width <= 0) return -1;
    return length*width;
}
```

Unix-like系统(C语言)编程常用的方式！

- 在此情况下，需要调用者注意
  - 调用者和被调用者都需要进行错误检查

```
int z = area(x,y);
if (z<0) error("bad area computation");
// …
```

- 问题
  - 调用者可能忘记做错误检查
  - 许多函数并没有可以用作标记错误信息的额外返回值 (e.g. max())

如何报告一个错误？

- 设置一个错误状态标识 (有问题，不要用！!)

```
int errno = 0;        // used to indicate errors
int area(int length, int width)
{
    if (length<=0 || width<=0) errno = 7;        // || means or
    return length*width;
}
```

操作系统常用的方式！

- 在此情况下，需要调用者检查

```
int z = area(x,y);
if (errno==7) error("bad area computation");
// ...
```

- 问题
  - 我们可能忘记检查错误标识 errno？
  - 如何选择一个不同的错误码 errno — 每个标识的意义？
  - 如何处理这些错误？

如何报告一个错误？

- 通过异常报告

```
class Bad_area { };  // a class is a user defined type
                                    // Bad_area is a type to be used as an exception

int area(int length, int width)
{
    if (length<=0 || width<=0) throw Bad_area();   // note the ()
    return length*width;
}
```

- 捕捉和处理错误 (e.g., in **main()**)

```
try {
    int z = area(x,y);   //  if area() doesn't throw an exception
}                                          // make the assignment and proceed
catch(Bad_area) {     // if area() throws Bad_area(), respond
    cerr << "oops! Bad area calculation – fix program\n";
}
```

异常

- 异常处理是一种一般化的方法
  - 你不会忘记处理一个异常
    - 如果忘记处理(通过使用 try … catch)，程序将会不正常终止
  - 异常能够报告任何类型的错误
  - 异常将错误检查和错误处理分离
    - 以此可区分程序的处理分支和异常

- 现在，你仍需要考虑如何去处理一个异常 (每个异常都是程序抛出的)
  - 实际编程中，错误处理从来都不是简单的

范围错误：out of range

- 尝试下面的代码

```
vector<int> v(10);        // a vector of 10 ints,
                          // each initialized to the default value, 0,
                          // referred to as v[0] .. v[9]
for (int i = 0; i<v.size(); ++i) v[i] = i;        // set values
for (int i = 0; i<=10; ++i)                       // print 10 values (???)
    cout << "v[" << i << "] == " << v[i] << endl;
```

著名的“偏一位错误”！

- 向量操作符 operator[ ]（下标）通过抛出一个范围错误异常来报告一个错误的索引
  - 此处使用的是 #include "std_lib_facilities.h"
  - 程序默认的行为可能是不同的

异常处理

- 目前为止，我们能够使用异常来优雅地终止程序，如下：

```
int main()
try
{
  // ...
}
catch (out_of_range&) {        // out_of_range exceptions
  cerr << "oops – some vector index out of range\n";
}
catch (…) {                    // all other exceptions
  cerr << "oops – some exception\n";
}
```

许多可能的代码

万能异常处理器

error()函数

- error() 是我们在 std_lib_facilities.h 中提供的一个简单函数
  - 通过调用error()，我们能够打印一个错误消息
  - 它隐藏了异常的抛出操作，工作方式类似于：

```
void error(string s)              // one error string
{
        throw runtime_error(s);
}

void error(string s1,  string s2)          // two error strings
{
        error(s1 + s2);        // concatenates
}
```

使用 error()

❖ 示例：

```
cout << "please enter integer in range [1..10]\n";
int x = -1;       // initialize with unacceptable value (if possible)
cin >> x;
if (!cin)         // check that cin read an integer
   error("didn't get a value");
if (x < 1 || 10 < x)        // check if value is out of range
   error("x is out of range");
// if we get this far, we can use x with confidence
```

如何查找错误？

- 当我们写完一个程序后，它通常情况下是有错误的(称之为“Bugs”)

对大部分程序来说，它几乎是100%的！
否则，你可以开会庆祝了？！

  - 它能够工作，但没有按我们期望的那样工作
  - 如何发现程序实际做了哪些事情？
  - 如何纠正错误?
  - 这一过程通常称之为“调试”(“debugging”)

调试

- 尽量避免调试的方法：

while (*program doesn't appear to work***) {** *// pseudo code*

Randomly look at the program for something that "looks odd"

Change it to "look better"

}

伪代码，用人类语言来描述程序的执行
有利于计算过程和原理的理解！

- 调试中的关键问题是：

我如何知道程序是否真正运行正确呢？

程序结构

- 使程序更易读，你会有更多机会发现错误所在
  - 注释
    - 解释设计思想(代码本身表达清楚的不用注释！)
  - 使用有意义的名字
  - 缩进
    - 使用一致的代码层次结构
    - 集成开发环境IDE能够帮助但不能代替你做所有的事情
      - 你是代码的唯一负责人
  - 将代码分成许多小的功能函数
    - 尽量避免超过一页的函数
  - 避免使用复杂的程序语句
    - 尽量避免使用嵌套的循环，嵌套的 if 语句等(但有时候你必须这样做)
    - 复杂代码是最容易隐藏错误的地方！
  - 尽量使用标准库而不是自己的代码

首先，让程序编译通过

- 每个字面常量是否正常结束了？

```
cout << "Hello, << name << '\n';          // oops!
```

- 每个字符字面常量是否正常结束了？

```
cout << "Hello, "  << name << '\n;        // oops!
```

- 每个程序块是否正常结束了？

```
if (a>0) { /* do something */
else { /* do something else */ }  // oops!
```

- 每个括号是否完全匹配？

```
if (a                          // oops!
    x = f(y);
```

  - 编译器一般会“稍晚些”报告这类错误
    - 它不知道你不想稍晚点再输入一个右括号

首先，让程序编译通过

- 每个名字都声明了吗?
  - 你是否包含了所有必须的头文件? (e.g., **std_lib_facilities.h**)
- 每个名字都在使用前声明了吗?
  - 你是否正确拼写了所有名字?

```
int count;          /* ... */ ++Count;   // oops!
char ch;            /* ... */ Cin>>c;              // double oops!
```

- 你用分号终止了每个表达式语句吗?

```
x = sqrt(y)+2  // oops!
z = x+3;
```

调试技术

- 小心地跟踪程序，可依次采用下面的方法
  - 假定你自己是计算机来执行程序——单步执行
  - 输出是否与你期望的匹配？
  - 如果输出信息不足够调试，可添加一些调试语句

```
cerr << "x == " << x << ", y == " << y << '\n';
```

cerr类似于cout，专门用于输出错误信息

- 非常仔细地
  - 查看程序实际做了什么，而不是你希望它做的
    - 听起来容易，但做起来难！
    - `for (int i=0; 0<month.size(); ++i) {` // *oops!*
    - `for( int i = 0; i<=max; ++j) {` // *oops! (twice)*

调试技术

- 在编程过程中，插入一些“检查不变式”(变量应该有合理的值)
  - 其中，函数参数检查是最常用的情况

```
if (number_of_elements<0)
        error("impossible: negative number of elements");

if (largest_reasonable<number_of_elements)
        error("unexpectedly large number of elements");

if (x<y) error("impossible: x<y");
```

或使用断言assert

- 仔细设计这些检查
  - 即使你认为程序已经正确了，也可以留一些在程序中

调试技术

- 特别注意一些“两端情况”(开始和结束处)
  - 初始化每个变量了吗?
    - 赋于一个合理值
  - 函数是否输入了正确的参数?
    - 函数的返回值是否正确?
  - 你是否正确使用了第一个元素?
    - 最后一个元素呢?
  - 你是否正确处理了空值的情况?
    - 没有元素
    - 没有输入
  - 你是否正确打开了文件?
    - chapter 11中有更多细节
  - 你实际上读取那个输入了吗?
    - 写入那个输出了吗?

调试技术

- 现在，“如果你找不到错误，几乎可以肯定你找错地方了”
  - 常理方式：你知道问题之所在，就“顽固地”在错误的地方进行排查
  - 不要仅靠猜测，根据输出值来指导你
    - 从你能确认正确的地方开始前向检查
      - 接下来会发生什么？为什么？
    - 从某个具有错误输出的地方反向查看
      - 如何导致这种可能发生的情况?
- 一旦你找到了bug，小心的考虑纠正它是否整个程序就正确了
  - 通常情况下，解决一个bug会引入新的bug
- “我找到了最后一个bug”
  - 对程序员来说，这就是一个笑话

注意

- 几乎可以肯定，错误处理比编写普通代码更困难和繁琐
  - 基本上，只有一种方式能够正确的工作
  - 而错误的方式有千万种
- 程序的用户越多，错误处理必须做的越好
  - 如果仅仅破坏了你自己的代码，那是你自己的问题
    - 你的过程会更艰难
  - 如果你的朋友使用你的代码，没有捕获的错误会导致你失去朋友
  - 如果陌生人使用你的代码，没有捕获的错误会引起严重的危机
    - 你没有办法再恢复它所带来的损失

前置条件

- 函数对它的输入参数有什么要求呢？
  - 这样的要求称之为前置条件
  - 有时，直接检查它更好

```
int area(int length, int width)  // calculate area of a rectangle
    // length and width must be positive
{
        if (length<=0 || width <=0) throw Bad_area();
        return length*width;
}
```

在不容易处理的地方，必须通过注释或其他方式进行说明！

后置条件

- 当一个函数返回时，它的结果是否正确呢？
  - 这样的要求称之为后置条件

```
int area(int length, int width)  // calculate area of a rectangle
    // length and width must be positive
{
        if (length<=0 || width <=0) throw Bad_area();
        // the result must be a positive int that is the area
        // no variables had their values changed
        return length*width;
}
```

  - 在复杂计算情况下，这种检查更有用！

前置和后置条件

- 编程时，总是要考虑它们，不要忘记
- 如果不能用代码说明，就采用注释的方式
- 在每个可能的“合理地方”对它们进行检查
- 在debug过程中，要多多检查这些条件

- 有些情况可能是麻烦的 (如前例中)
  - 什么情况下，对area() 的前置检查是成功的而后置条件是失败的呢?

测试

- 如何测试一个程序？
  - 是个系统性的工程
    - “用键盘逐个试”对初始化好的小程序是可以的，但对实际的系统往往是不够的
  - 从新考虑测试和它的正确性
    - 在可能的情况下，独立测试程序的一部分
      - e.g., 当你写一个复杂函数时，编写一个小程序，使用一些参数来简单调用它，以在放入实际系统前独立检查它的功能行为
        - » 单元测试
  - Chapter 26将会重新讨论这一话题

Next

- 在接下来的两章中，我们将讨论一个小的完整程序的设计和实现
  - 一个简单的桌面计算器