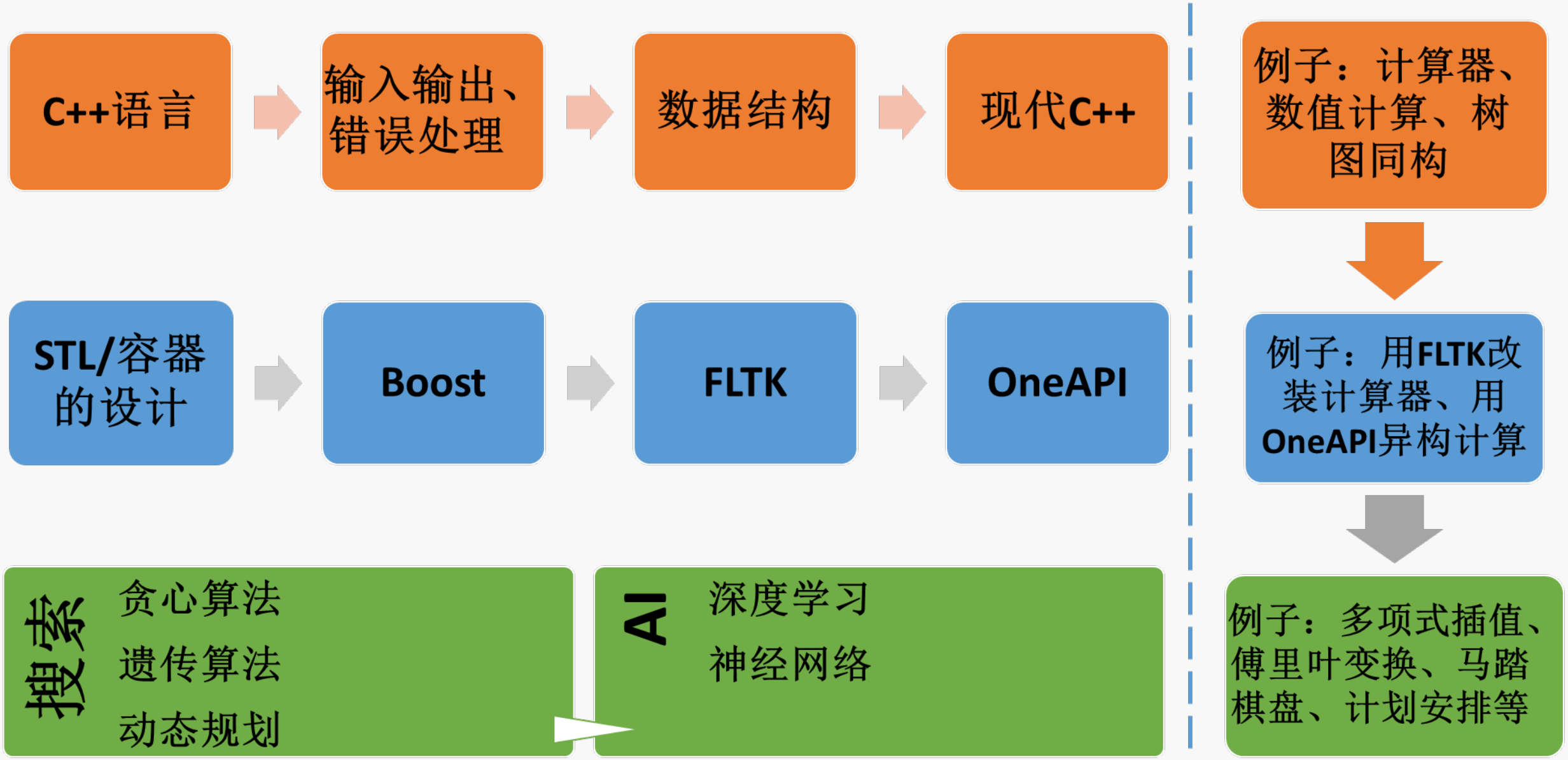

C++语言
输入输出、错误处理
数据结构
现代C++
STL/容器的设计
Boost
FLTK
OneAPI
贪心算法
遗传算法
动态规划
AI
深度学习
神经网络
例子：计算器、数值计算、树图同构
例子：用FLTK改装计算器、用OneAPI异构计算
例子：多项式插值、傅里叶变换、马踏棋盘、计划安排等

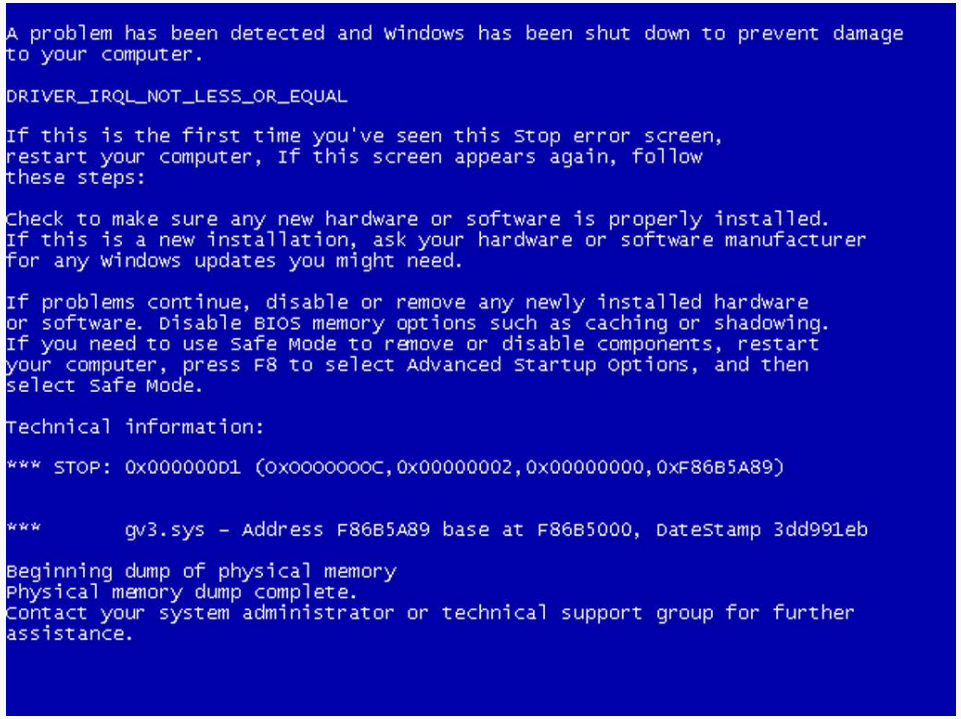
问题求解与实践

——C++回顾：错误处理

主讲教师： 陈雨亭、沈艳艳

Overview

- Kinds of errors
- Argument checking
  - Error reporting
  - Error detection
  - Exceptions
- Debugging
- Testing



So what is programming?

- Conventional definitions
  - Telling a **very** fast moron ***exactly*** what to do
  - A plan for solving a problem on a computer
  - Specifying the order of a program execution
    - But modern programs often involve millions of lines of code
    - And manipulation of data is central
- Definition from another domain (academia)
  - A … program is an organized and directed accumulation of resources to accomplish specific … objectives …
    - Good, but no mention of actually doing anything
- The definition we'll use
  - Specifying the structure and behavior of a program, and testing that the program performs its task correctly and with acceptable performance
    - Never forget to check that "it" works
- Software == one or more programs

Errors

Your Program

1. Should produce the desired results for all legal inputs
2. Should give reasonable error messages for illegal inputs
3. Need not worry about misbehaving hardware
4. Need not worry about misbehaving system software
5. Is allowed to terminate after finding an error

3, 4, and 5 are true for beginner's code; often, we have to worry about those in real software.

Sources of errors

- Poor specification
  - “What’s this supposed to do?”
- Incomplete programs
  - “but I’ll not get around to doing that until tomorrow”
- Unexpected arguments
  - “but **sqrt()** isn’t supposed to be called with **-1** as its argument”
- Unexpected input
  - “but the user was supposed to input an integer”
- Code that simply doesn’t do what it was supposed to do
  - “so fix it!”

Kinds of Errors

- Compile-time errors
  - Syntax errors
  - Type errors
- Link-time errors
- Run-time errors
  - Detected by computer (crash)
  - Detected by library (exceptions)
  - Detected by user code
- Logic errors
  - Detected by programmer (code runs, but produces incorrect output)

Check your inputs

- Before trying to use an input value, check that it meets your expectations/requirements
  - Function arguments
  - Data from input (**istream**)

Bad function arguments

- The compiler helps:
  - Number and types of arguments must match

```
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
```

How to report an error

- Return an “error value” (not general, problematic)

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

- So, “let the caller beware”

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

- Problems
  - What if I forget to check that return value?
  - For some functions there isn’t a “bad value” to return (e.g., max())

How to report an error

- Set an error status indicator (not general, problematic, don't!)

```
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;
}
```

- So, "let the caller check"

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

- Problems
  - What if I forget to check **errno**?
  - How do I pick a value for **errno** that's different from all others?
  - How do I deal with that error?

How to report an error

- Report an error by throwing an exception

```
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 {} – a value
   return length*width;
}
```

- Catch and deal with the error (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";
}
```

Exceptions

- Exception handling is general
  - You can’t forget about an exception: the program will terminate if someone doesn’t handle it (using a **try … catch**)
  - Just about every kind of error can be reported using exceptions
- You still have to figure out what to do about an exception (every exception thrown in your program)
  - Error handling is **never** really simple

Out of range

- Try this

```
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;
```

- vector's **operator[ ]** (subscript operator) reports a bad index (its argument) by throwing a **Range_error** if you use **#include "std_lib_facilities.h"**
  - The default behavior can differ
  - You can't make this mistake with a range-**for**

Exceptions – for now

- For now, just use exceptions to terminate programs gracefully, like this

```
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";
}
```

A function **error()**

- Here is a simple **error()** function as provided in **std_lib_facilities.h**
- This allows you to print an error message by calling **error()**
- It works by disguising throws, like this:

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

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

Using **error( )**

- Example

```
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
```

Debugging

How to look for errors

- When you have written (drafted?) a program, it'll have errors (commonly called "bugs")
  - It'll do something, but not what you expected
  - How do you find out what it actually does?
  - How do you correct it?
  - This process is usually called "debugging"

Program structure

- Make the program easy to read so that you have a chance of spotting the bugs
  - Comment
    - Explain design ideas
  - Use meaningful names
  - Indent
    - Use a consistent layout
    - Your IDE tries to help (but it can’t do everything)
      - You are the one responsible
  - Break code into small functions
    - Try to avoid functions longer than a page
  - Avoid complicated code sequences
    - Try to avoid nested loops, nested if-statements, etc.
      - (But, obviously, you sometimes need those)
  - Use library facilities

First get the program to compile

- Is every string literal terminated?

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

- Is every character literal terminated?

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

- Is every block terminated?

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

- Is every set of parentheses matched?

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

- The compiler generally reports this kind of error “late”
  - It doesn’t know you didn’t mean to close “it” later

First get the program to compile

- Is every name declared?
  - Did you include needed headers? (e.g., **std_lib_facilities.h**)

- Is every name declared before it's used?
  - Did you spell all names correctly?

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

- Did you terminate each expression statement with a semicolon?

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

Debugging

- Carefully follow the program through the specified sequence of steps
  - Pretend you're the computer executing the program
  - Does the output match your expectations?
  - If there isn't enough output to help, add a few debug output statements

    cerr << "x == " << x << ", y == " << y << '\n';
- Be very careful
  - See what the program specifies, not what you think it should say
    - That's much harder to do than it sounds
    - **for (int i=0; 0<month.size(); ++i) {** **// *oops!***
    - **for(int i = 0; i<=max; ++j) {** **// *oops! (twice)***

Debugging

- When you write the program, insert some checks (“sanity checks”) that variables have “reasonable values”
  - Function argument checks are prominent examples of this

```
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");
```

- Design these checks so that some can be left in the program even after you believe it to be correct
  - It’s almost always better for a program to stop than to give wrong results

Debugging

- Pay special attention to “end cases” (beginnings and ends)
  - Did you initialize every variable?
    - To a reasonable value
  - Did the function get the right arguments?
    - Did the function return the right value?
  - Did you handle the first element correctly?
    - The last element?
  - Did you handle the empty case correctly?
    - No elements
    - No input
  - Did you open your files correctly?
    - more on this in chapter 11
  - Did you actually read that input?
    - Write that output?

Pre-conditions

- What does a function require of its arguments?
  - Such a requirement is called a pre-condition
  - Sometimes, it's a good idea to check it

```
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;
}
```

Post-conditions

- What must be true when a function returns?
  - Such a requirement is called a post-condition

```
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;
}
```

Pre- and post-conditions

- Always think about them
- If nothing else write them as comments
- Check them “where reasonable”
- Check a lot when you are looking for a bug
- This can be tricky
  - How could the post-condition for area() fail after the pre-condition succeeded (held)?

Testing

Testing

- “A systematic way to search for errors”
- Real testers use a lot of tools
  - Unit test frameworks
  - Static code analysis tools
  - Fault injection tools
  - …
- When done well, testing is a highly skilled and most valuable activity
- “Test early and often”
  - Whenever you write a function or a class, think of how you might test it
  - Whenever you make a significant change, re-test (“regression testing”)
  - Before you ship (even after the most minor change), re-test

Testing

- Some useful sets of values to check (especially boundary cases):
  - the empty set
  - small sets
  - large sets
  - sets with extreme distributions
  - sets where “what is of interest” happens near the ends
  - sets with duplicate elements
  - sets with even and with odd number of elements
  - some sets generated using random numbers

Primitive test harness for binary_search()

```
int a1[ ] = { 1,2,3,5,8,13,21 };
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),1) == false)
   cout << "1 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),5) == false)
   cout << "2 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),8) == false)
   cout << "3 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),21) == false)
   cout << "4 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),-7) == true)
   cout << "5 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),4) == true)
   cout << "6 failed";
if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),22) == true)
   cout << "7 failed";
```

Primitive, but better, test harness for binary_search()

```
for (int x : { 1,2,3,5,8,13,21 })
  if (binary_search(a1,a1+sizeof(a1)/sizeof(*a1),x) == false)
        cout << x << " failed";
```

A Better Test Harness (still primitive)

Put the variables into a data file, *e.g.*, with a format of

{ 27 7 { 1 2 3 5 8 13 21} 0 }

meaning

{test_number value {sequence} result}

i.e., test #**27** calls our binary_search to look for the value **7** in the sequence **{ 1 2 3 5 8 13 21}** and checks that the result is **0** (false, that is, not found).

Now it's (relatively) easy to write lots of test cases, or even write another program to generate a data file with lots of (random) cases.

Loops

Most errors occur at the ends, *i.e.*, at the first case or the last case. Can you spot 3 problems in this code? 4? 5?

```
int do_loop(vector<int>& vec) // messy function
                              // undisciplined loop
{
   int i;
   int sum;
   while(i<=vec.size()) sum+=v[i];
   return sum;
}
```

Buffer Overflow

- Really a special type of loop error, *e.g.*, “storing more bytes than will fit” into an array—where *do* the “extra bytes” go? (probably not a good place)
- The premiere tool of virus writers and “crackers” (evil hackers)
- Some vulnerable functions (best avoided):
  - **gets, scanf** **//** *these are the worst: avoid!*
  - **sprintf**
  - **strcat**
  - **strcpy**
  - **…**

Buffer overflow

- Don’t avoid unsafe functions just as a fetish
  - Understand what can go wrong and don’t just write equivalent code
  - Even unsafe functions (e.g. **strcpy()**) have uses
    - if you really want to copy a zero terminated string, you can’t do better than **strcpy()** – just be sure about your “strings” (How?)

```
char buf[MAX];
char* read_line()    // harmless? Mostly harmless? Avoid like the plague?
{
   int i = 0;
   char ch;
   while (cin.get(ch) && ch!='\n')  buf(i++)=ch;
   buf[i+1]=0;
   return buf;
}
```

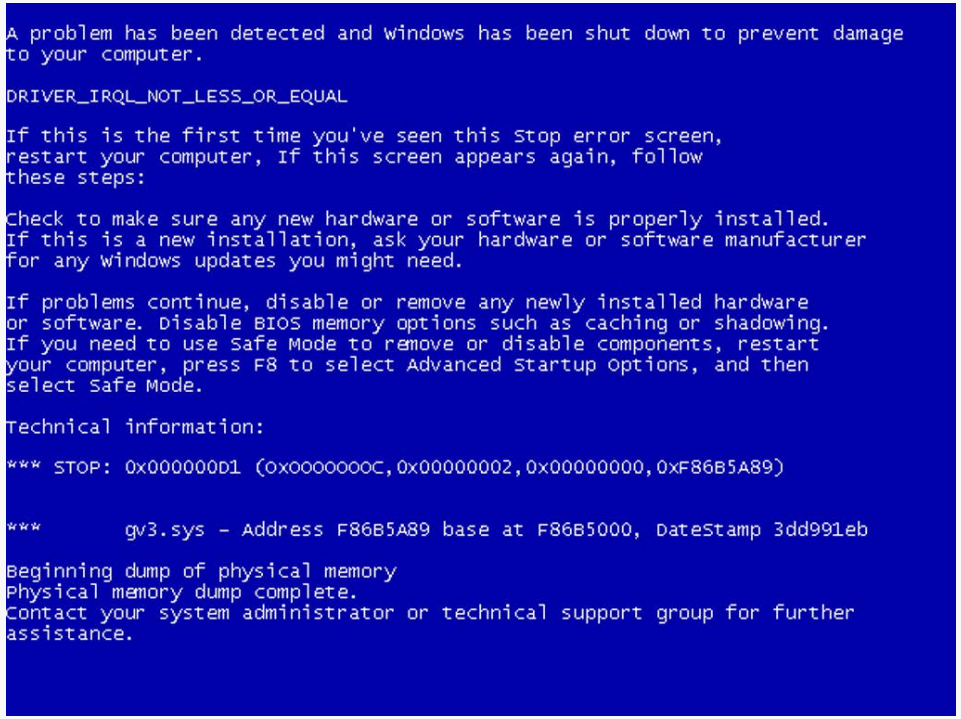
Buffer overflow

- Don’t avoid unsafe functions just as a fetish
  - Understand what can go wrong and don’t just write equivalent code
  - Write simple and safe code

string buf;
getline(cin,buf); // buf *expands to hold the newline terminated input*

Overview

- Programming in C++
- Kinds of errors
- Argument checking
  - Error reporting
  - Error detection
  - Exceptions
- Debugging
- Testing




问题求解与实践
——C++回顾：错误处理
THANKS
FOR YOUR WATCHING