Chapter 10
Input/Output Streams

Zhengwei QI

Most slides are from Bjarne Stroustrup

www.stroustrup.com/Programming

Recap

```
// simple Date (use enum class Month)
class Date {
public:
        Date(int y, Month m, int d);  // check for valid date and initialize
    // ...
private:
    int y;          // year
    Month m;
    int d;          // day
};

Date my_birthday(1950, 30, Month::dec); // error: 2nd argument not a Month
Date my_birthday(1950, Month::dec, 30);  // OK
```

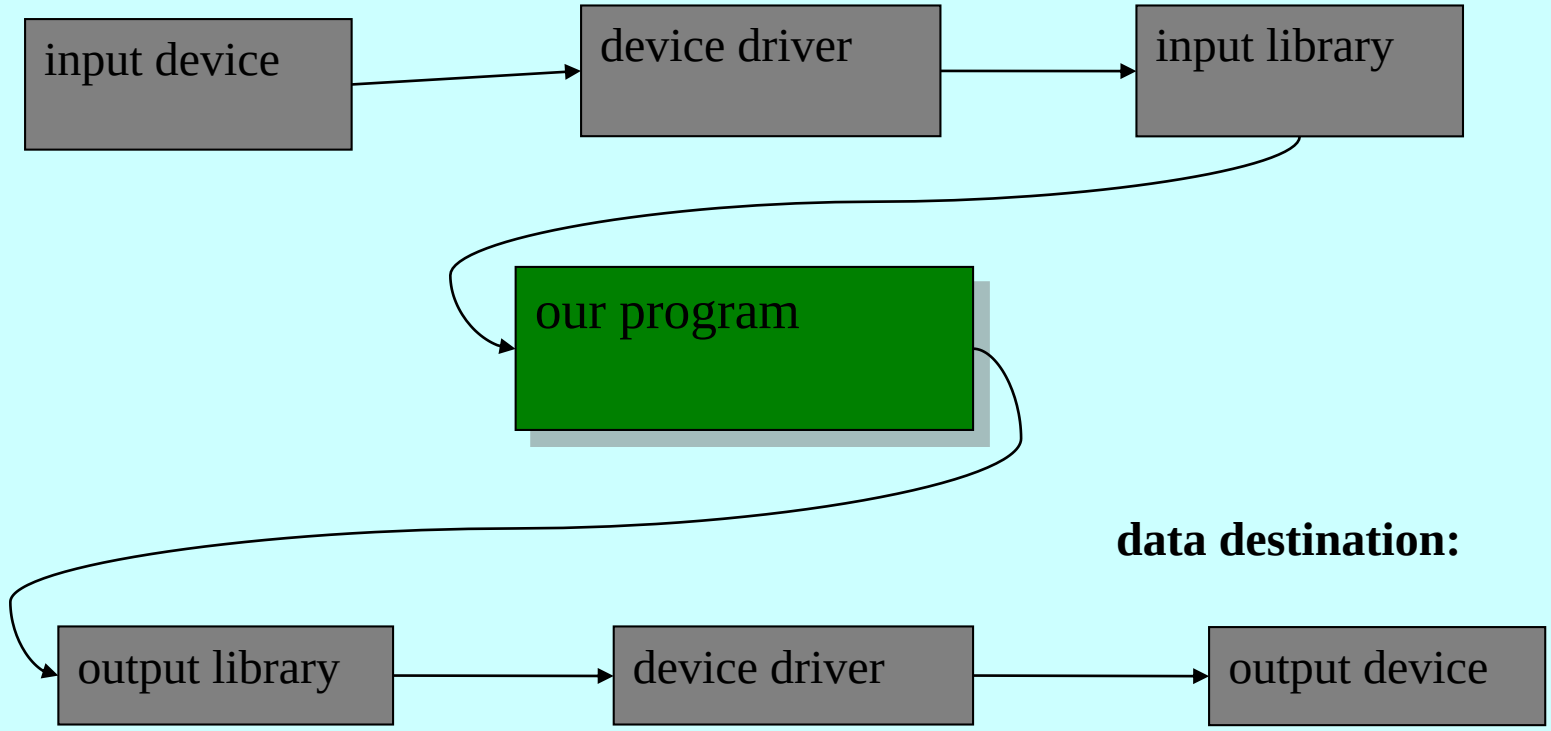
Overview

- Fundamental I/O concepts
- Files
  - Opening
  - Reading and writing streams
- I/O errors
- Reading a single integer

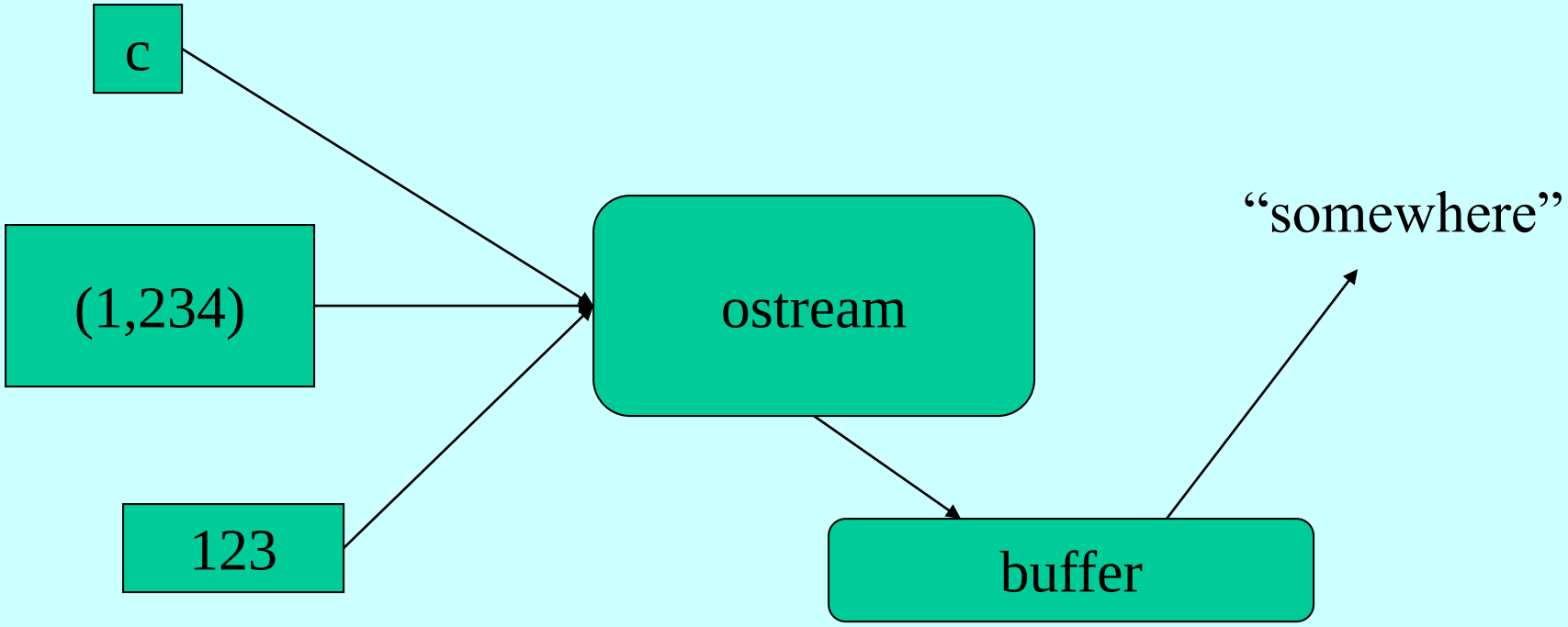
```c
#include<stdio.h>
struct stu{
    char name[10];
    int num;
    int age;
    char addr[15];
}boya[2],boyb[2],*pp,*qq;
main(){
    FILE *fp;
    char ch;
    int i;
    pp=boya;
    qq=boyb;
    if((fp=fopen("d:\\jrzh\\example\\stu_list","wb+"))==NULL){
        printf("Cannot open file strike any key exit!");
        getch();
        exit(1);
    }
    printf("\ninput data\n");
    for(i=0;i<2;i++,pp++)
        scanf("%s%d%d%s",pp->name,&pp->num,&pp->age,pp->addr);
    pp=boya;
    fwrite(pp,sizeof(struct stu),2,fp);
    rewind(fp);
    fread(qq,sizeof(struct stu),2,fp);
    printf("\n\nname\tnumber      age      addr\n");
    for(i=0;i<2;i++,qq++)
        printf("%s\t%5d%7d     %s\n",qq->name,qq->num,qq->age,qq->addr);
    fclose(fp);
} « end main »
```

Input and Output

data source:

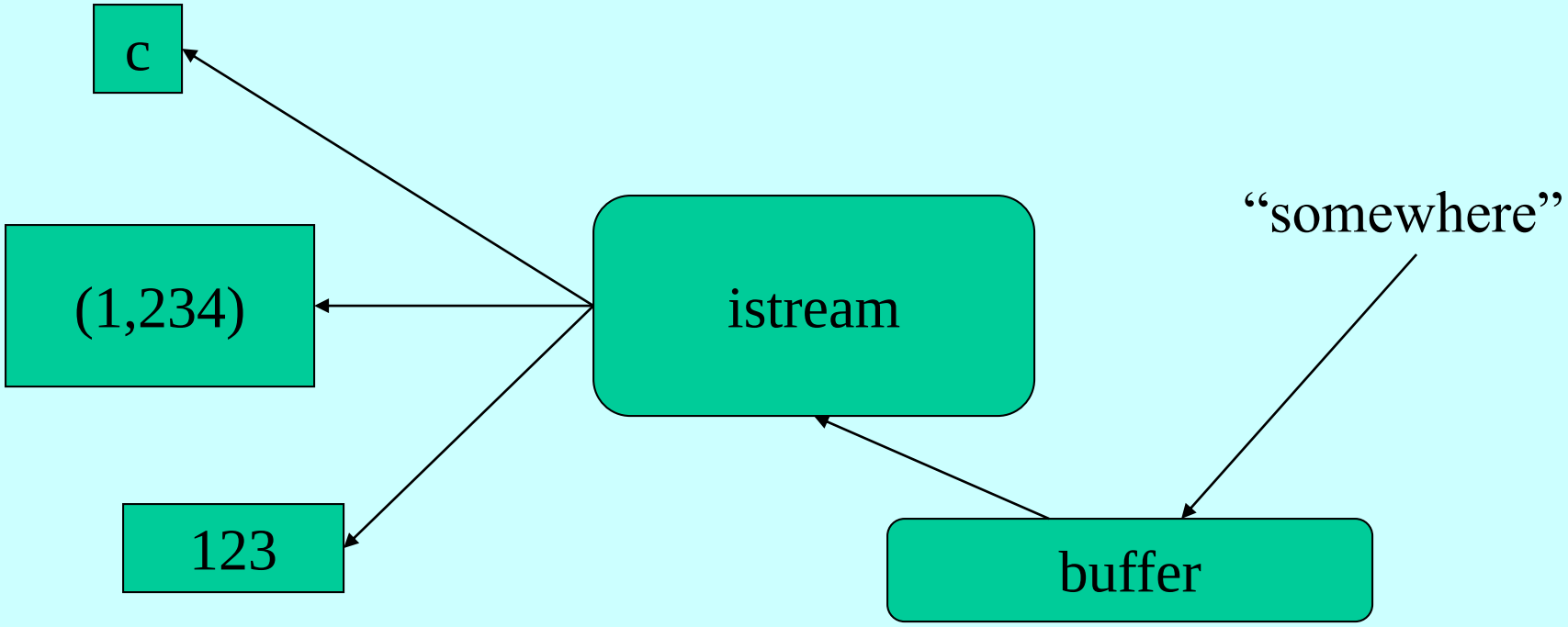


The stream model



- An **ostream**
  - turns values of various types into character sequences
  - sends those characters somewhere
    - *E.g.*, console, file, main memory, another computer

The stream model



- An **istream**
  - turns character sequences into values of various types
  - gets those characters from somewhere
    - *E.g.*, console, file, main memory, another computer

The stream model

- Reading and writing
  - Of typed entities
    - << (output) and >> (input) plus other operations
    - Type safe
    - Formatted
  - Typically stored (entered, printed, etc.) as text
    - But not necessarily (see binary streams in chapter 11)
  - Extensible
    - You can define your own I/O operations for your own types
  - A stream can be attached to any I/O or storage device

Files

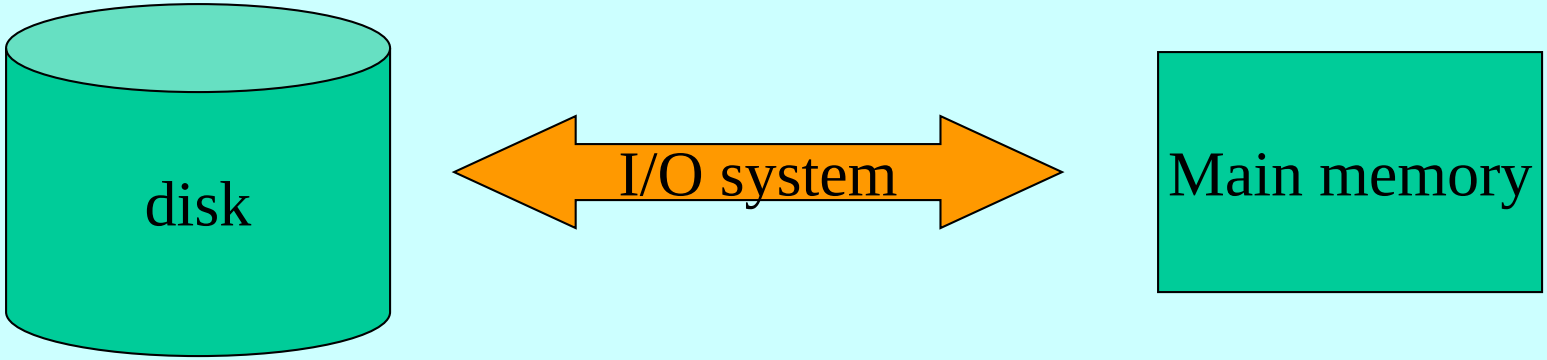
- We turn our computers on and off
  - The contents of our main memory is transient
- We like to keep our data
  - So we keep what we want to preserve on disks and similar permanent storage
- A file is a sequence of bytes stored in permanent storage
  - A file has a name
  - The data on a file has a format
- We can read/write a file if we know its name and format

A file

- At the fundamental level, a file is a sequence of bytes numbered from 0 upwards
- Other notions can be supplied by programs that interpret a “file format”
  - For example, the 6 bytes "123.45" might be interpreted as the floating-point number 123.45

Files

- General model



Files
(sequences of bytes)

iostreams

Objects
(of various types)

Files

- To read a file
  - We must know its name
  - We must open it (for reading)
  - Then we can read
  - Then we must close it
    - That is typically done implicitly
- To write a file
  - We must name it
  - We must open it (for writing)
    - Or create a new file of that name
  - Then we can write it
  - We must close it
    - That is typically done implicitly

Opening a file for reading

```
// …
int main()
{
    cout << "Please enter input file name: ";
    string iname;
    cin >> iname;
    ifstream ist {iname};  // ifstream is an "input stream from a file"
                           // defining an ifstream with a name string
                           // opens the file of that name for reading
    if (!ist) error("can't open input file ", iname);
    // …
```

Opening a file for writing

```
    // …
    cout << "Please enter name of output file: ";
    string oname;
    cin >> oname;
    ofstream ofs {oname};        // ofstream is an "output stream from a file"
                                 // defining an ofstream with a name string
                                 // opens the file with that name for writing
    if (!ofs) error("can't open output file ", oname);
    // …
}
```

Reading from a file

- Suppose a file contains a sequence of pairs representing hours and temperature readings

  0 60.7
  1 60.6
  2 60.3
  3 59.22

- The hours are numbered 0..23
- No further format is assumed
  - Maybe we can do better than that (but not just now)
- Termination
  - Reaching the end of file terminates the read
  - Anything unexpected in the file terminates the read
    - *E.g.*, q

Reading a file

```
struct Reading {        // a temperature reading
        int hour;       // hour after midnight [0:23]
        double temperature;
};

vector<Reading> temps;      // create a vector to store the readings

int hour;
double temperature;
while (ist >> hour >> temperature) {                        // read
        if (hour < 0 || 23 <hour) error("hour out of range");   // check
        temps.push_back( Reading{hour,temperature} );       // store
}
```

I/O error handling

- Sources of errors
  - Human mistakes
  - Files that fail to meet specifications
  - Specifications that fail to match reality
  - Programmer errors
  - Etc.

- iostream reduces all errors to one of four states
  - good() // *the operation succeeded*
  - eof() // *we hit the end of input ("end of file")*
  - fail() // *something unexpected happened*
  - bad() // *something unexpected and serious happened*

Sample integer read "failure"

- Ended by "terminator character"
  - 1 2 3 4 5 *
  - State is **fail()**
- Ended by format error
  - 1 2 3 4 5.6
  - State is **fail()**
- Ended by "end of file"
  - 1 2 3 4 5 end of file
  - 1 2 3 4 5 Control-Z (Windows)
  - 1 2 3 4 5 Control-D (Unix)
  - State is **eof()**
- Something really bad
  - Disk format error
  - State is **bad()**

I/O error handling

```
void fill_vector(istream& ist, vector<int>& v, char terminator)
{           // read integers from ist into v until we reach eof() or terminator
    for (int i; ist >> i; )      // read until "some failure"
            v.push_back(i);  // store in v
    if (ist.eof()) return;                      // fine: we found the end of file
    if (ist.bad()) error("ist is bad");   // stream corrupted; let's get out of here!

    if (ist.fail()) {      // clean up the mess as best we can and report the problem
            ist.clear();        // clear stream state,  so that we can look for terminator
            char c;
            ist >> c;           // read a character, hopefully terminator
            if (c != terminator) {                          // unexpected character
                    ist.unget();                            // put that character back
                    ist.clear(ios_base::failbit); // set the state back to fail()
            }
    }
}
```

Throw an exception for bad()

```
// How to make ist throw if it goes bad:
ist.exceptions(ist.exceptions()|ios_base::badbit);

// can be read as
//  "set ist's exception mask to whatever it was plus badbit"
//  or as "throw an exception if the stream goes bad"
```

Given that, we can simplify our input loops by no longer checking for **bad**

Simplified input loop

```
void fill_vector(istream& ist, vector<int>& v, char terminator)
{   // read integers from ist into v until we reach eof() or terminator
    for (int i; ist >> i; )
        v.push_back(i);
    if (ist.eof()) return;      // fine: we found the end of file

    // not good() and not bad() and not eof(), ist must be fail()
    ist.clear();                // clear stream state
    char c;
    ist >> c;                   // read a character, hopefully terminator
    if (c != terminator) {      // ouch: not the terminator, so we must fail
        ist.unget();            // maybe my caller can use that character
        ist.clear(ios_base::failbit);       // set the state back to fail()
    }
}
```

Reading a single value

```
// first simple and flawed attempt:

cout << "Please enter an integer in the range 1 to 10
        (inclusive):\n";
int n = 0;
while (cin>>n) {                        // read
        if   (1<=n && n<=10) break;             // check range
        cout << "Sorry, "
                << n
                << " is not in the [1:10] range; please try again\n";
}
// use n here
```

- Three kinds of problems are possible
  - the user types an out-of-range value
  - getting no value (end of file)
  - the user types something of the wrong type (here, not an integer)

Reading a single value

- What do we want to do in those three cases?
  - handle the problem in the code doing the read?
  - throw an exception to let someone else handle the problem (potentially terminating the program)?
  - ignore the problem?

  - Reading a single value
    - Is something we often do many times
    - We want a solution that's very simple to use

Handle everything: What a mess!

```
cout << "Please enter an integer in the range 1 to 10 (inclusive):\n";
int n = 0;
while (cin >> n) {
    if (cin) {          // we got an integer; now check it:
        if (1<=n && n<=10) break;
        cout << "Sorry, " << n << " is not in the [1:10] range; please try
    again\n";
    }
    else if (cin.fail()) {          // we found something that wasn't an integer
        cin.clear();            // we'd like to look at the characters
        cout << "Sorry, that was not a number; please try again\n";
        for (char ch; cin>>ch && !isdigit(ch); )          // throw away non-digits
                /* nothing */ ;
        if (!cin) error("no input");    // we didn't find a digit: give up
        cin.unget();// put the digit back, so that we can read the number
    }
    else
        error("no input"); // eof or bad: give up
}
// if we get here n is in [1:10]
```

The mess: trying to do everything at once

- Problem: We have all mixed together
  - reading values
  - prompting the user for input
  - writing error messages
  - skipping past "bad" input characters
  - testing the input against a range

- Solution: Split it up into logically separate parts

What do we want?

- What logical parts do we what?
  - int get_int(int low, int high); // *read an int in [**low**..**high**] from **cin***
  - int get_int(); // *read an int from **cin***
// *so that we can check the range **int***
  - void skip_to_int(); // *we found some "garbage" character*
// *so skip until we find an int*
- Separate functions that do the logically separate actions

Skip "garbage"

```
void skip_to_int()
{
    if (cin.fail()) {           // we found something that wasn't an integer
        cin.clear();            // we'd like to look at the characters
        for(char ch; cin>>ch; ) {// throw away non-digits
            if (isdigit(ch) || ch=='-') {
                cin.unget();        // put the digit back,
                                    // so that we can read the number
                return;
            }
        }
    }
    error("no input");      // eof or bad: give up
}
```

Get (any) integer

```
int get_int()
{
    int n = 0;
    while (true) {
        if (cin >> n) return n;
        cout << "Sorry, that was not a number; please try again\n";
        skip_to_int();
    }
}
```

Get integer in range

```
int get_int(int low, int high)
{
    cout << "Please enter an integer in the range "
         << low << " to " << high << " (inclusive):\n";
    while (true) {
         int n = get_int();
         if (low<=n && n<=high) return n;
         cout << "Sorry, "
                   << n << " is not in the [" << low << ':' << high
                   << "] range; please try again\n";
    }
}
```

Use

```
int n = get_int(1,10);
cout << "n: " << n << endl;

int m = get_int(2,300);
cout << "m: " << m << endl;
```

- Problem:
  - The "dialog" is built into the read operations

What do we *really* want?

```
// parameterize by integer range and "dialog"

int strength = get_int(1, 10,
                       "enter strength",
                       "Not in range, try again");
cout << "strength: " << strength << endl;

int altitude = get_int(0, 50000,
                       "please enter altitude in feet",
                       "Not in range, please try again");
cout << "altitude: " << altitude  << "ft. above sea level\n";
```

- That's often the really important question
- Ask it repeatedly during software development
- As you learn more about a problem and its solution, your answers improve

Parameterize

```
int get_int(int low, int high, const string& greeting, const string& sorry)
{
    cout << greeting << ": [" << low << ':' << high << "]\n";
    while (true) {
        int n = get_int();
        if (low<=n && n<=high) return n;
        cout << sorry  << ": [" << low << ':' << high << "]\n";
    }
}
```

- Incomplete parameterization: **get_int()** still "blabbers"
  - "utility functions" should not produce their own error messages
  - Serious library functions do not produce error messages at all
    - They throw exceptions (possibly containing an error message)

User-defined output: operator<<()

- Usually trivial

```
ostream& operator<<(ostream& os, const Date& d)
{
    return os << '(' << d.year()
              << ',' << d.month()
              << ',' << d.day() << ')';
}
```

- We often use several different ways of outputting a value
  - Tastes for output layout and detail vary

Use

```
void do_some_printing(Date d1, Date d2)
{
    cout << d1;          // means operator<<(cout,d1) ;

    cout << d1 << d2;
                    // means (cout << d1)  <<  d2;
                                // means (operator<<(cout,d1)) <<  d2;
                    // means operator<<((operator<<(cout,d1)), d2) ;
}
```

User-defined input: operator>>()

```
istream& operator>>(istream& is, Date& dd)
    // Read date in format: ( year , month , day )
{
    int y, d, m;
    char ch1, ch2, ch3, ch4;
    is >> ch1 >> y >> ch2 >> m >> ch3 >> d >> ch4;
    if (!is) return is;          // we didn't get our values, so just leave
    if (ch1!='(' || ch2!=',' || ch3!=',' || ch4!=')') {      // oops: format error
        is.clear(ios_base::failbit);    // something wrong: set state to fail()
        return is;                      // and leave
    }
    dd = Date{y,Month(m),d};      // update dd
    return is;                    // and leave with is in the good() state
}
```

Review


/2017/腾/讯/代/码/报/告
提交语言
TOP10
{{ 2017 }}
C/C++稳居榜首 前端语言火热
C/C++
14646万行
JS
5628万行
▲1
Java
4752万行
▼1
HTML/CSS
3870万行
▲1
C#
2994万行
▼1
PHP
2922万行
Obj-C/C++
2010万行
Python
1854万行
Lua
1146万行
ASP.NET
439万行
1
2
3
4
5
6
7
8
9
10
— 4 —
头条号 / 聪明

The Aims

- Teach/learn
  - Fundamental programming concepts
  - Key useful techniques
  - Basic Standard C++ facilities
- After the course, you'll be able to
  - Write small colloquial C++ programs
  - Read much larger programs
  - Learn the basics of many other languages by yourself
  - Proceed with an "advanced" C++ programming course
- After the course, you will ***not*** (yet) be
  - An expert programmer
  - A C++ language expert
  - An expert user of advanced libraries

Chapter 1-2 Programming

- Why C++?
- Why software?
- Where is C++ used?
- Hello World program
- Computation & Linking
- What is programming?
- Integrated Development Environment

```
#include "std_lib_facilities.h " //header

int main() // where a C++ programs start
{
    cout << "Hello, world\n";// output

    keep_window_open();    // wait
    return 0;          // return success
}
```

Chapter 3 Types

- Builtin types: int, double, bool, char
- Library types: string, complex
- Input and output
- Operators—“overloading”
- Variable names in C++
- Simple computations
- Literals
- Declaration & initialization
- Type safety
- Programming philosophy

```
// inch to cm and cm to inch conversion:
int main()
{
    const double cm_per_inch = 2.54;
    int val;
    char unit;
    while (cin >> val >> unit) {// keep reading
        if (unit == 'i')          // 'i' for inch
            cout << val << "in == “
                 << val*cm_per_inch << "cm\n";
        else if (unit == 'c')     // 'c' for cm
            cout << val << "cm == "
                 << val/cm_per_inch << "in\n";
        else
            return 0; // terminate on a “bad
                                   // unit”, e.g. 'q'
    }
}
```

Chapter 4 Computation

- Expressing computations
  - Correctly, simply, efficiently
  - Divide and conquer
  - Use abstractions
  - Organizing data, **vector**
- Language features
  - Expressions
    - Boolean operators (e.g. ||)
    - Short cut operators (e.g. +=)
  - Statements
  - Control flow
  - Functions
- Algorithms

```
// Eliminate the duplicate words; copying unique words
vector<string> words;
string s;
while (cin>>s && s!= "quit")
        words.push_back(s);
sort(words.begin(), words.end());
vector<string>w2;
if (0<words.size()) {
        w2.push_back(words[0]);
        for (int i=1; i<words.size(); ++i)
                if(words[i-1]!=words[i])
                w2.push_back(words[i]);
 }
cout<< "found " << words.size()-w2.size()
        <<  " duplicates\n";
for (int i=0; i<w2.size(); ++i)
         cout << w2[i] << "\n";
```

Chapter 5 Errors

- Errors (“bugs”) are unavoidable in programming
  - Sources of errors?
  - Kinds of errors?
- Minimize errors
  - Organize code and data
  - Debugging
  - Testing
- Do error checking and produce reasonable messages
  - Input data validation
  - Function arguments
  - Pre/post conditions
- Exceptions– **error()**

```
int main()
{
    try
    {
      // …
    }
    catch (out_of_range&) {
      cerr << "oops – some vector "
                  " index out of  range\n";
    }
    catch (…) {
      cerr << "oops – some exception\n";
    }
    return 0;
}
```

Chapter 6 Writing a Program

- Program a simple desk calculator
  - Process of repeatedly analyzing, designing, and implementing
- Strategy: start small and continually improve the code
- Use pseudo coding
- Leverage prior work
  - Expression Grammar
  - Functions for parsing
- Token type
- Program organization
  - Who calls who?
- Importance of feedback

```
double primary()     // Num or '(' Expr ')'
{
    Token t = get_token();
    switch (t.kind) {
    case '(':                                   //
    handle '('expression ')'
    {       double d = expression();
            t = get_token();
            if (t.kind != ')') error("')'  expected");
            return d;
    }
    case '8': // '8' represents number "kind"
            return t.value; // return value
    default:
            error("primary expected");
    }
}
```

Chapter 7 Completing a Program

- Token type definition
  - Data members
  - Constructors
- Token_stream type definition
  - Function members
  - Streams concept
- "Grow" functionality: eg. prompts, and error recovery
- Eliminate "magic" constants

```
class Token_stream {
    bool full;        // is there a Token in the buffer?
    Token buffer; // here is where we keep a Token
public:
    Token get();      // get a Token
    void putback(Token); // put back a Token
    // the buffer starts empty:
    Token_stream() :full(false), buffer(0) { }

};

void Token_stream::putback(Token t)
{
    if (full) error("putback() into a full buffer");
    buffer=t;
    full=true;
}
```

Chapter 8 Functions

- Declarations and definitions
- Headers and the preprocessor
- Scope
  - Global, class, local, statement
- Functions
- Call
  - by value,
  - by reference, and
  - by **const** reference
- Namespaces
  - Qualification with :: and **using**

```
namespace Jack {//  in Jack's header file
        class Glob{ /*...*/ };
        class Widget{ /*...*/ };
}


#include "jack.h";    // this is in your code
#include "jill.h";    // so is this

void my_func(Jack::Widget p)
{       // OK, Jack's Widget class will not
        // clash with a different Widget
        // …
}
```

Chapter 9 Classes

- User defined types
  - **class** and **struct**
  - **private** and **public** members
    - Interface
  - **const** members
  - constructors/destructor
  - operator overloading
  - Helper functions
  - Enumerations **enum**
- **Date** type

```
// simple Date (use Month type)
class Date {
public:
    enum Month {
        jan=1, feb, mar, apr, may, jun, jul,
        aug, sep, oct, nov, dec
    };
    Date(int y, Month m, int d);  // check for valid
                                  // date and initialize
    // …
private:
    int y;              // year
    Month m;
    int d;              // day
};

Date my_birthday(1950, 30, Date::dec);      // error:
                                  // 2nd argument not a Month
```

Chapter 10 Streams

- Devices, device drivers, libraries, our code
- The stream model,
  - type safety, buffering
  - operators << and >>
- File types
  - Opening for input/output
  - Error handling
    - check the stream state
- Code logically separate actions as individual functions
- Parameterize functions
- Defining >> for **Date** type

```
struct Reading {  // a temperature reading
        int hour;   // hour after midnight [0:23]
        double temperature;
        Reading(int h, double t) :hour(h),
        temperature(t) { }
};

string name;
cin >> name;
ifstream ist(name.c_str());
vector<Reading> temps;     // vector of readings
int hour;
double temperature;
while (ist >> hour >> temperature) {   // read
        if (hour < 0 || 23 <hour)
                error("hour out of range");
        temps.push_back(
        Reading(hour,temperature) ); // store
}
```

Chapter 11 Customizing I/O

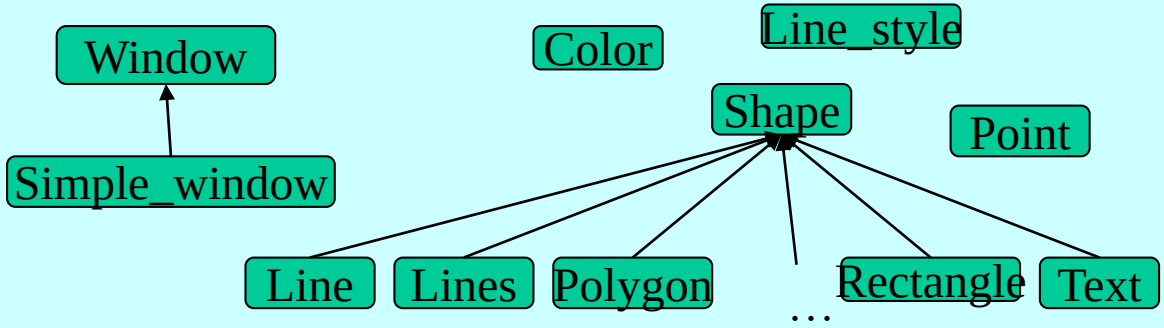
- Formatted output—manipulators for **int** and **double**
- File open modes
- Text vs binary files
- Positioning in a filestream
- **stringstream**s
- Line and **char** input/output
- Character classification functions

```
double str_to_double(string s)
// if possible, convert characters
// in s to floating-point value
{
    istringstream is(s);          // make a stream
    double d;
    is >> d;
    if (!is) error("double format error");
    return d;
}

double d1 = str_to_double("12.4"); // testing
double d2 = str_to_double("1.34e-3");
// will call error():
double d3 = str_to_double("twelve point four");
```

Chapter 12 Graphics

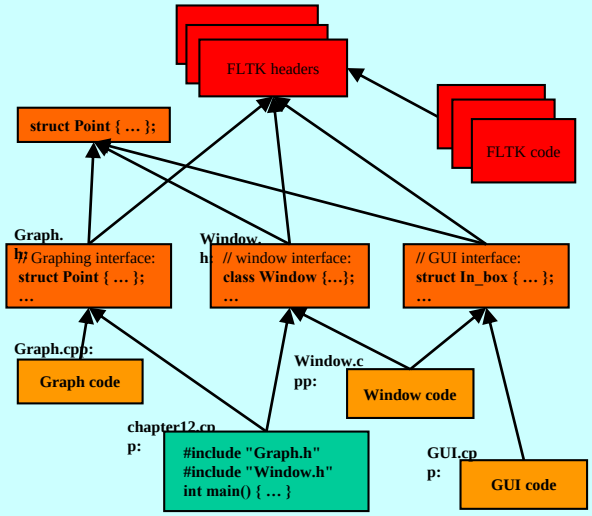
- Why Graphics/GUI?
- WYSIWYG
- Display model
  - Create a Window
  - Create Shapes
  - Attach objects
  - Draw
- 2D Graphics/GUI library
  - FLTK
  - Layered architecture
  - Interface classes

```
int main()
{
    using namespace Graph_lib; // use graph library
    Point tl(100,200);              // a point (obviously)
    Simple_window win(tl,600,400,"Canvas");
    Polygon poly;     // make a polygon shape
    poly.add(Point(300,200));    // add three points
    poly.add(Point(350,100));
    poly.add(Point(400,200));
    poly.set_color(Color::red);     // make it red
    win.attach(poly);     // connect poly to the window
    win.wait_for_button(); // give up control
}
```



Chapter 13 Graphics Classes

- Code organization




```
Simple_window win20(pt,600,400,"16*16 color  matrix");

Vector_ref<Rectangle> vr;  // use like vector
    // but imagine that it holds references to objects
for (int i = 0; i<16; ++i) { // i is the horizontal
                             // coordinate
    for (int j = 0; j<16; ++j) {    // j is the vertical
                                    // coordinate
        vr.push_back(
            new Rectangle(  Point(i*20,j*20),20,20)
            );
        vr[vr.size()-1].set_fill_color(i*16+j);
        win20.attach(vr[vr.size()-1]);
    }
```

- Implementation of **Point**, **Line,Color**, **Line_style**, **Polylines**, **Text**, etc.
- Object-oriented programming

Chapter 14: Design Principles for Programming a Class Library

- Implement types used in the application domain
- Derived classes inherit from a few key abstractions
- Provide a minimum number of operations, access functions
- Use a consistent, regular style, appropriate naming
- Expose the interface only
  - encapsulation
- Virtual functions
  - dynamic dispatching

```
void Shape::draw() const
        // The real heart of class Shape
        // called by Window (only)
{
        Fl_Color oldc = fl_color();  // save old color
           // there is no good portable way of
           // retrieving the current style (sigh!)
        fl_color(line_color.as_int()); // set color and
                                       // style
        fl_line_style(ls.style(),ls.width());

        // call the appropriate draw_lines():
        draw_lines();  // a "virtual call"
                       // here is what is specific for a
                       // "derived class" is done

        fl_color(oldc);   // reset color to previous
        fl_line_style(0);// (re)set style to default
}
```

Chapter 15 Graphing

- Graphing functions
- Labeling, use of color
- Scaling
- typedef
- Standard mathematical functions
- Function approximation
- Rounding errors
- Graphing data

```
Function::Function( Fct f,
        double r1, double r2, //range
        Point xy,             // screen location of (0, 0)
        int count,            // number of points
        double xscale,        // location (x,f(x)) is
         double yscale        // (xscale*x,yscale*f(x))
        )
{
        if (r2-r1<=0)
                error("bad graphing range");
        if (count <=0)
                error("non-positive graphing count");
        double dist = (r2-r1)/count;
        double r = r1;
        for (int i = 0; i<count; ++i) {
                add(Point(xy.x+int(r*xscale),
                                xy.y-int(f(r)*yscale)));
                r += dist;
        }
}
```

Chapter 16 GUI

- Graphical I/O
- Layered architecture
- Control inversion
  - Callbacks
  - Wait loops
  - Event oriented actions
- Buttons
- Input/output boxes

```
Button start_button(Point(20,20), 100, 20,
      "START", cb_start);
…
static void cb_start(Address, Address addr) {
      reference_to<Window>(addr).start();
}
void start(void) { start_pushed = true; }
….
void wait_for_start(void){
      while (!start_pushed) Fl::wait();
      start_pushed = false;
      Fl::redraw();
}
….
Window win (Point(10,10), “My Window");
….
win.wait_for_start();
```

Chapter 17 Free Store

- Built vector type
- Pointer type
- The **new** operator to allocate objects on the free store (heap)
- Why use free store?
- Run-time memory organization
- Array indexing
- Memory leaks
- **void***
- Pointers vs references

```
class vector {
    int sz;             // the size
    double* elem;       // a pointer to the elements
public:
    // constructor (allocate elements):
    vector(int s) :sz(s), elem(new double[s]) { }
    // destructor (deallocate elements):
    ~vector() { delete[ ] elem; }
    // read access:
    double get(int n) { return elem[n]; }
    // write access:
    void set(int n, double v) { elem[n]=v; }
    // the current size:
    int size() const { return sz; }
};
vector v(10);
for (int i=0; i<v.size(); ++i) {
    v.set(i,i); cout << v.get(i) << ' ';
 }
```

Chapter 18 Arrays

- Vector copy constructor
- Vector copy assignment
- Shallow and deep copy
- Arrays—avoid if possible
- Overloading [ ]
  - i.e. defining [] for **vector**

```
class vector {
    int sz;             // the size
    double* elem;       // pointer to elements
public:
    // constructor:
    vector(int s) :sz(s), elem(new double[s]) { }
    // …
    // read and write access: return a reference:
    double& operator[ ](int n) { return elem[n]; }
};

vector v(10);
for (int i=0; i<v.size(); ++i) {    // works and
                                    // looks right!
    v[i] = i;                       // v[i] returns a
                        // reference to the ith element
    cout << v[i];
}
```

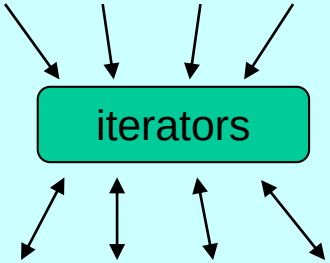
Chapter 19 Vector

- Changing vector size
- Representation changed to include free *space*
- Added
  - **reserve(int n)**,
  - **resize(int n)**,
  - **push_back(double d)**
- The *this* pointer
- Optimized copy assignment
- Templates
- Range checking
- Exception handling

```
// an almost real vector of Ts:
template<class T> class vector { // "for all types T"
        int sz;                 // the size
        T* elem;                // a pointer to the elements
        int space;              // size+free_space
public:
        // default constructor:
        vector() : sz(0), elem(0), space(0);
        // constructor:
        explicit vector(int s)
                :sz(s), elem(new T[s]), space(s) {
        // copy constructor:
        vector(const vector&);
        // copy assignment:
        vector& operator=(const vector&);
        ~vector() { delete[ ] elem; } // destructor
        // access: return reference
        T& operator[ ] (int n) { return elem[n]; }
        int size() const { return sz; } // the current size

        // ...
};
```

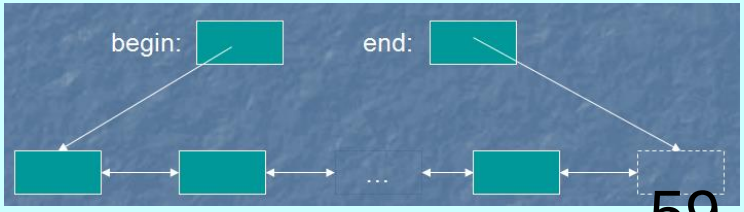
Chapter 20 STL

- Generic programming
  - "lifting an algorithm"
- Standard Template Library
- 60 Algorithms
  - sort, find, search, copy, …



  - vector, list, map, hash_map,…
- 10 Containers
- iterators define a sequence
- Function objects

```
// Concrete STL-style code  for a more
// general version of summing values

template<class Iter, class T>     // Iter should be an
                                  // Input_iterator
                                  // T should be
                                  // something we can
                                  // + and =
T sum(Iter first, Iter last, T s) // T is the
                                  // "accumulator type"
{
      while (first!=last) {
            s = s + *first;
            ++first;
      }
      return s;
}
```



Word counting example (C++ version)

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

int main()
{
    map<string,int> m;
    for (string s; cin>>s; )
        m[s]++;
    for(const auto& p : m)
        cout << p.first << " : " << p.second << "\n";
}
```

Word counting example (C version)

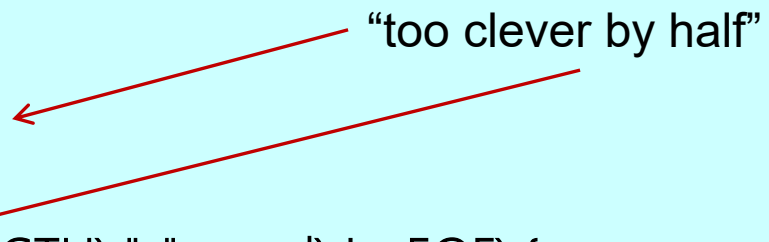
```
// word_freq.c
// Walter C. Daugherity

#include <stdio.h>
#include <stdlib.h>
 #include <string.h>

#define MAX_WORDS 1000 /* max unique words to count */
#define MAX_WORD_LENGTH 100

#define STR(s) #s              /* macros for scanf format */
#define XSTR(s) STR(s)

typedef struct record {
    char word[MAX_WORD_LENGTH + 1];
    int count;
} record;
```

Word counting example (C version)

```
int main()
{
    // … read words and build table …
    qsort(table, num_words, sizeof(record), strcmp);
    for (iter=0; iter<num_words; ++iter)
            printf("%s  %d\n",table[iter].word,table[iter].count);
    return EXIT_SUCCESS;
}
```

Word counting example (most of main)

```
record table[MAX_WORDS + 1];
int num_words = 0;
char word[MAX_WORD_LENGTH + 1];
int iter;
while (scanf("%" XSTR(MAX_WORD_LENGTH) "s", word) != EOF) {
    for (iter = 0; iter < num_words && strcmp(table[iter].word, word); ++iter);
    if (iter == num_words) {
        strncpy(table[num_words].word, word, MAX_WORD_LENGTH + 1);
        table[num_words++].count = 1;
    }
    else table[iter].count++;
    if (num_words > MAX_WORDS){
        printf("table is full\n");
        return EXIT_FAILURE;
    }
}
```



Next

- Data structure
  - Stanford PA