Introduction

程序系统的三要素：
1. 输入 Input：从键盘、文件、其它输入设备、其它程序（进程间通信）、程序的其它部分（线程间通信或函数调用）
2. 计算 Computation：程序从输入到产生输出过程中具体的操作
3. 输出 Output:到屏幕、文件、其它输出设备、其它程序、程序的其它部分

建立一个程序/应用的几个阶段（软件工程）
1. 分析 Analysis
   - 精炼我们对问题的理解 思考一下，程序/应用最终的用途是什么；给出需求集描述 问题是什么？用户想要什么？用户需要哪种可靠性？
2. 设计 Design
   - 创建程序（应用）的总体结构:如何解决问题？系统的整体结构将是什么？系统包括哪些部分？这些部分之间如何通信？用户接口？
   - 采用分解技术对问题进行分解，描述分解后模块间关系
3. 详细设计/实现 Implementation
   - 首先编写算法（伪码）表达问题求解的方法，然后再编写代码。
   - 系统化地进行调试、测试(单元、集成、系统)，保证这些代码的正确和可维护性。
4. 多次重复以上过程

计算（Computation）
- 程序员的任务：正确性（包括正常和异常流畅处理），简单性，高效性（容易忽略）
- 一种手段是分治 把一个大问题分为几个小问题分别解决 直到问题小到能够被我们很好地理解和解决为止 (原则)
- 另一种手段是抽象 提供一种高级的概念，而将具体实现细节隐藏在接口之后
- 数据的良好组织，是构成好代码的关键 输入/输出格式 Input/output formats 数据结构 Data structures 协议 Protocols
- 强调：结构和组织–非常重要

简单语句的堆砌并不能构成好的代码

数据处理相关的通用任务（抽象）
- 收集数据到容器中
- 组织数据输出：打印 快速访问
- 提取数据项：根据索引 根据值 根据属性/条件
- 修改容器中的数据：增加数据 删除数据 排序和搜索
- 简单的数值运算

编程计算需要—产品理念 与性能需求
- 对代码的要求：容易读、容易修改（易维护） 规范、简短、高效
- 对数据实现安全、一致的访问：数据维护（提取/增加/删除/修改）快速、方便 数据访问/维护独立于它是如何存储的(与具体存储方式无关)
- 对核心算法提供通用、标准实现：追求更好的规范、更大范围的重用和很好的性能 从具体到抽象，逐步求精，提炼概念、算法 泛型编程技术：接口、泛类型、泛数据容器、泛处理条件

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

文件注释、类注释、函数注释、语句注释、参数注释

编程不仅仅是功能实现和工作结果正确，还要强调优美性 正确性 correctness ——包括正常和异常流程 高效性 Efficiency 简单性 simplicity 可维护性 maintainability

类型
- 明确: 该类变元需占用内存空间的大小和可执行的操作
- 语言本身提供了一些类型：内建类型 (“built-in types”) e.g. C 语言内建类型：bool, char, int, double
- 程序员自定义类型类型 (“user-defined types”)：语言提供自定义类型的多种可支持形式 e.g. C 语言可支持的自定义形式：enum, struct, union C++增加了全新的自定义类型：class, 并对 struct 进行了类似 class 形式扩展
- C++标准库提供了一些常用类型：e.g. string, vector, complex 本质上说, 这些属于用户自定义类型

声明与定义
- 声明将一个名称引入到作用域中，并指定名字的类型：在 C++中，名称只有声明以后才能够使用
- 若声明还给出了声明实体(实现)的完整描述，则称之为定义：定义一定是声明，但声明不一定都是定义。
- 变量/对象是有名字的、按某种类型分配的内存空间。Type 明确: 需占用内存空间的大小和可执行的操作 Name 用于检索内存空间 编译器 (符号表) Value 内存空间上实际的数值。

不能定义一个对象两次：但可以声明一个对象多次：声明通常是通过一个"头(文件)"引入到一个程序中 这为"抽象"提供了可能

C++的基本语言要素
- 常量及其表示
- 类型与表达式(算术/逻辑)：内建/自定义类型
- 变量定义与初始化：声明与定义关系
- 语句：分类（赋值计算、判断选择、控制转移） 逻辑组织结构（顺序、选择与循环）
- 函数：声明（要素有哪些?） 参数传入、传出方式如何表示? 定义与应用语法

作用域
- 作用域是一个程序文本的有效范围区域 — 编译器特性 类型有：全局作用域 (整个程序) 类作用域 (在一个类内) 局部作用域 (在花括号 {} 之间) 语句作用域 (e.g. for 语句)
- 一个作用域的名称对该作用域和嵌套内的作用域是可见的：当然，必须在名称声明之后 (受"不能先用" 规则限制)
- 作用域使"实体 (things)" 是局部的：避免我的变量、函数等实体与你的发生冲突 谨记：实际程序中有成千上万个实体 局部性是最好的！ 尽量使名称局部化 审慎选用全局变量

错误报告和异常处理
- 通过异常报告错误

```
class Bad_area  : exception {...};  // 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();
  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";
} catch (...) {   // all other exceptions
  cerr << "oops-some exception\n";
}
```

error() 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
}
```

```
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
```

```
Expression :
  Term
  Expression '+' Term     e.g., 1+2,   (1-2)+3, 2*3+1
  Expression '-' Term

Term :
  Primary
  Term '*' Primary     e.g., 1*2,   (1-2)*3.5
  Term '/' Primary
  Term '%' Primary

Primary :
  Number       e.g., 1,   3.5
  '(' Expression ')'    e.g., (1+2*3)

Number :
  floating-point literal     e.g., 3.14, 0.274e1, or 42 – as defined for C++
```

编译器/自然语言理解的第一步，是基于分词 (Token) (e.g., 数字和运算符)切分一行输入

```
class Token { // define a type called Token
public:
  char kind;                  // what kind of token
  double value; // used for numbers (only): a value
  Token (char ch):kind(ch), value(0) {};
  Token (char ch, double val):kind(ch), value(val) {};
};

class Token_stream {
  // representation: not directly accessible to users:
  bool full;        // is there a Token in the buffer?
  Token buffer;     // here is where we keep a Token put back using putback()
public:    // user interface:
  Token get();    // get a Token
  void putback(Token); // put a Token back into the Token_stream
  Token_stream(); // constructor: make a Token_stream
};
void Token_stream::putback(Token t) { if (full) error("putback() into a full buffer");
  buffer=t;
  full=true;
}

Token Token_stream::get() {   // read a Token from the Token_stream
  if (full) { full=false; return buffer; }  // check if we already have a Token ready

  char ch;
  cin >> ch;  // note that >> skips whitespace (space, newline, tab, etc.)

  switch (ch) {
  case '(': case ')': case ';': case 'q': case '+': case '-': case '*': case '/':
    return Token(ch);   // let each character represent itself
  case '.':
  case '0': case '1': case '2': case '3': case '4': case '5': case '6': case '7': case '8': case '9':
  { cin.putback(ch);    // put digit back into the input stream
    double val;
    cin >> val;   // read a floating-point number
    return Token('8',val);  // let '8' represent "a number"
  }
  default:
    error("Bad token");
  }
}
```

```
Expression:
  Term
  Expression '+' Term // Note: every Expression starts with a Term
  Expression '-' Term

double expression() // read and evaluate: 1   1+2.5   1+2+3.14  etc. {
  double left = term();       // get the Term
  while (true) {
    Token t = get_token();    // get the next token…
    switch (t.kind) {     // … and do the right thing with it
    case '+':     left += term(); break;
    case '-':     left -= term(); break;
    default:            ts.putback(t);  // put the unused token back
      return left;
    }
  }
}

Term :
  Primary
  Term '*' Primary  // Note: every Term starts with a Primary
  Term '/' Primary
double term() // exactly like expression(), but for * and  / {
  double left = primary();    // get the Primary
  while (true) {
    Token t = get_token();  // get the next Token
    switch (t.kind) {
    case '*':   left *= primary();
      break;
    case '/':     double d = primary();
      if  (d==0) error("divide by zero");
      left /= d;
      break;

    default:
      return left;  // return the value
    }
  }
}

double primary()  // Number or '(' Expression ')' {
  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':   // we use '8' to represent the "kind" of a number
    return t.value; // return the number's value
  case '-':                   //处理负数
    return 0- primary();
  default:
    ts.putback(t);  // put unused token back into input stream
    error("primary expected");
  }
}

int main()  {
  double val = 0;
  cout << '>';  // print 输出提示符 (prompt)
  try {
    while (cin) {
      Token t = ts.get(); // rather than get_token()
      if (t.kind == 'q') break;   // 'q' for "quit"
      if (t.kind == ';')      // ';' for "print now"
        cout <<  val << '\n'; // print result
      else
        ts. putback(t); // put a token back into the input stream
      val = expression(); // evaluate
    }
    keep_window_open();
    return 0;
  } catch(exception& e) {
    cerr << "error:"<<e.what()<<"\n";   // cerr 专用于错误信息输出,类似 cout
    keep_window_open();
    return 1;
  catch (...) {
    cerr << "Oops:unknown exception!\n";
    keep_window_open();
    return 2;
  }
}
```

错误报告

```
void calculate() {
  while (cin)
  try {
    cout << prompt;
    Token t = ts.get();
    while (t.kind == print) t=ts.get();
    if (t.kind == quit) return;
    ts.putback(t);
    cout << result << expression() << endl;
  }
  catch (exception& e) {
    cerr << e.what() << endl;
          //在此处嵌入错误恢复处理功能,也是一个函数!
    clean_up_mess();
  }
}
```

典型问题：一个程序的上层不能恢复好底层产生的错误 i.e., “bad tokens”产生的错误 只有 Token_stream 了解字符串 因此，我们必须在字符处理(低)层次上进行处理 解决方案是必须在 Token_stream 中修改

```
class Token_stream {
  bool full;          // is there a Token in the buffer?
  Token buffer; // here is where we keep a Token put back using putback()
public:
  Token get();    // get a Token
  void putback(Token t);  // put back a Token
  Token_stream();   // make a Token_stream that reads from cin
  void ignore(char c);  // discard tokens up to and including a c
};
```

I/O 流及其错误处理
- C++扩展支持 I/O 流模式访问数据文件

```
int main() {
cout << "Please enter input & output file name: ";
string iname,oname;
cin >> iname>>oname;
ifstream ifs(name.c_str()); // ifstream 是基于文件的输入流
if (!ifs) error("can't open input file ", name);
    ofstream ofs(name.c_str());          // ofstream 是基于文件的输出流
    if (!ofs) error("can't open output file ", name);     ……
}
```

- iostream 将所有的错误归结为四类流状态

good() eof() fail() bad()

I/O 读数据示例

```
struct Reading {  // a temperature reading
  int hour;   // hour after midnight [0:23]
  double temperature; // in Fahrenheit
  // (How would you define a Reading that could handle both Fahrenheit and Celsius?)
  Reading(int h, double t) :hour(h), temperature(t) { }
};

vector<Reading> temps;  // create a vector to store the readings
int hour;
double temperature;
while (ifs >> hour >> temperature) {        // read
  if (hour < 0 || 23 <hour) error("hour out of range");   // check
  temps.push_back( Reading(hour,temperature) );   // store
}
```

I/O 错误处理示例

```
void fill_vector(istream& ist, vector<int>& v, char terminator) {     // read integers from ist into v until we reach eof() or terminator
  int i = 0;
  while (ist >> i) v.push_back(i);      // read and store in v until "some failure"
  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!
      //fail 态可读–可恢复,只是格式不对或遇到了 terminator，尝试进一步处理
  if (ist.fail()) {       // clean up the mess as best we can and report the problem
    ist.clear();          // clear stream state 先清流状态,以便再读入一个字符
    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()
    }
                  // if (c == terminator)
return;  //调用者检测到的态是读到特殊终止符的 good 返回态
  }
}
```

指针和引用
- 你可以把引用看做为一种自动解引用的常量指针，或者是对象别名
- 对指针赋值改变的是指针
- 对引用赋值改变的是对象的值
- 你不能让一个引用指向不同的对象

```
int a = 10;
int* p = &a;  // you need & to get a pointer
*p = 7; // 通过 p 的解引用，给 a 赋值
      // 用* (或 [ ]) 取指针指向元素的值
int x1 = *p;  // read a through p

int& r = a;   // r is a synonym for a
int x2 = r; // read a through r
r = 9;  // assign to a through r ---对引用赋值改变的是对象的值
r = x1; // error: 你不能赋值改变一个常量解引用的绑定(指向)
p = &x1;          // ok:而指针允许多次赋值修改指向同类型的不同元素或数组
```

& 操作语义

取地址操作：获取指向任意对象的指针

```
int a;
char ac[20];

void f(int n) {
  int b;
  int* p = &b;  // pointer to individual variable
  p = &a;
  char* pc = ac;  // the name of an array names a pointer to its first element
  pc = &ac[0];  // equivalent to pc = ac
  pc = &ac[n];  // pointer to ac's nth element (starting at 0th)
    // ...  }
```

定义变量的别名

```
int& r = a;   // r is a synonym for a
int x2 = r; // read a through r
r = 9;  // assign to a through r ---对引用赋值改变的是对象的值
r = x1; // error: 你不能赋值改变一个常量解引用的绑定(指向)
p = &x1;          // ok:
```

*操作语义
- 定义指针

```
void f(int pi[ ]) { // equivalent to void f(int* pi)
  int a[ ] = { 1, 2, 3, 4 };
  int b[ ] = a; // error: copy isn't defined for arrays 数组---没有定义拷贝赋值!
  b = pi;   // error: 原因同上.

  pi = a;                 // ok: but it doesn't copy: pi now points to a's first element
            // 这里可能有内存泄露

  int* p = a; // p points to the first element of a
  int* q = pi;  // q points to the first element of a
}
```
- 引用指针值—解引用指针

```
int x = 7;
int x = 7;
int* p = &x;
int y = *p;           // 一元 * , 解引用,取右值
*p = 9;                 // 一元 * , 解引用,取右值
```

空指针 void* void* 表示指向内存的一个指针，但编译器不知道它的类型：无类型的指针
- 当需要在不同程序块之间传输地址，而不知道它们的确切类型时，需要使用 void* 例如：回调函数的参数
- 没有 void 类型的对象 ⟶ why ? (类型的作用)

```
void v; // error
void f(); // f() returns nothing - f() does not return an object of type void
```

任何对象的指针都可以赋值 为 void*

```
int* pi = new int;
double* pd = new double[10];
void* pv1 = pi;
void* pv2 = pd;
```

OOP

C++面向对象
- C++ 是一种支持多重编程范式的通用程序设计语言。在兼容支持 C 语言的过程化程序设计基础上，扩展支持面向对象程序设计、泛型程序设计等多种程序设计风格。
- 面向对象(Object Oriented)主要特性：封装（Encapsulation） 继承（Inheritance） 重载（Overloading） 基于虚函数的多态（Polymorphism based on virtual function）

基于类的封装
- “类” 是 C++表达用户自定义类型、组织程序的重要机制：可用来表达程序设计要实现的概念：struct 演伸为所有成员都属 public 一种特殊类。
- 可基于类将一些数据信息及其相关的处理方法的实现细节，封装、隐藏起来。 类内存储信息的数据项，称为数据成员(data members)。 类内处理/操作数据的函数成员，也称为方法(methods)。 包括构造函数、析构函数、存取函数、其它操作函数…… 要访问类内的代码和数据，必须通过严格控制的开放接口。 规范的开放接口形式：限于抽象的函式接口。
- 类的实例化：调用构造函数，生成→对象 调用析构函数，销毁→对象

类的方法
- 基本操作
  1. 默认构造函数：无定义情况下，编译器提供一个默认构造函数 (什么都不做) 如果定义了其他任何构造函数，则编译器不会再默认产生
  2. 拷贝构造函数：默认产生的是拷贝每一个成员
  3. 拷贝赋值操作：默认产生的是拷贝每一个成员
  4. 析构函数：默认情况下什么都不做。规范建议：在此集中安排—执行释放构造对象时所申请的各类资源相关代码，或子函数
- 辅助函数与操作符重载
- 其它操作

类的接口 为什么要严格区分 public/private 呢？ 为什么不让所有的属性都是 public 的呢?
- 提供更加简洁的接口 (对使用者)：数据和复杂的实现函数是 private 的
- 保持不变式约束：只有有限的函数能够访问数据
- 便于调试 (对开发者)：只有有限的函数能够访问数据 称之为“round up the usual suspects” 技术
- 允许你改变它的描述 (对维护者)：你只需改动几个固定的函数即可 你不知道究竟谁在使用 public 成员

好的接口是什么样的呢?
- 最小化：尽可能的小，这样才能更清晰 简化理解、调试和维护
- 完整：保证够用，也不能太小
- 类型安全：小心让人迷惑的参数顺序
- 使用 const 纠正引用参数传递

接口与实现的分离

为了保持类接口的简洁，可能引入需要额外的"辅助函数"

类烟花——不变式
- 此处的"合法日期"是类"合法值思想"的一个重要特例
- 我们尽量仔细地设计类，以保证值的合法性

或者,我们时常检查它的合法性，oops!
- 对合法值的约束规则称为"不变式"
- 如果我们不能构思一个好的不变式，就将它按普通数据结构进行处理：这种情况下，使用结构 (struct) 尽力为你的类设计好的不变式 如此，可将你从烦人的 bug 代码中解脱出来

枚举（Enumerations） 是一种简单的用户自定义类型，它指定了一个值的集合(枚举量， enumerators) 实现：将相关的一组常数，集中且简洁地进行定义 无名枚举量—定义一些简单的相关常量列表

```
enum { red, green };     // the enum { } doesn't define a scope
int a = red;       // red is available here
```

```
enum { red, blue, purple }; // error: red defined twice
```
有名枚举量，提供了附加的类型安全检查，例如：
```
enum Month { jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec };
enum Color { red, green, blue, /* … */  };
Month m1 = jan;                 //ok!
Month m2 = red; // error red isn't a Month ——类型安全

Month m3 = 7; // error 7 isn't a Month

int n = m;     // ok: we can get the numeric value of a Month
Month mm = Month(7);  // convert int to Month (unchecked)
```
一个枚举量总是表示一个整数，但不能把一个整数赋值给一个枚举量！

运算符重载
- 你可以在自己的类型 (类、枚举)上定义几乎所有的 C++运算符

这通常称为"运算符重载"
```
enum Month {
  jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month operator++(Month& m)  // prefix increment operator {
  m = (m==dec) ? jan : Month(m+1);  // "wrap around"
  return m;
}

Month m = nov;
++m;  // m becomes dec
++m;  // m becomes jan
```
- 你只能定义已经存在的运算符而 (不能定义新的运算符)： e.g., + - * / % [] () ^ ! & < <= > >=
- 你定义的运算符必须与已有的 C++语法一致 (操作数个数) e.g., 不能定义一元的<= (小于等于)和二元的 ! (非)
- 一个重载的运算符至少有一个用户自定义类型的操作数 `int operator+(int,int); // error: you can't overload built-in +` `Vector operator+(const Vector&, const Vector &);  // ok`
- 建议 (非语言规则) 重载运算符应该保持其原有意义 +应该是加法， * 应该是乘， [] 应该是访问， () 应该是调用, 等等
- 建议 (非语言规则)：除非你真正确定重载运算符能大大改善代码，否则不要为你的类型定义运算符 (非必要不要重载)！

拷贝构造与赋值构造
```
class vector {  // an almost real vector of doubles
public:
  vector() : sz(0), elem(0), space(0) { }    // default constructor
  vector(const vector&);       // copy constructor 拷贝构造
  vector& operator=(const vector&); // copy assignment 拷贝赋值
};

vector v(10),  v2(10);
vector v3 = &v;  //调用拷贝构造
v2 = &v;                              //调用拷贝赋值
```
继承，重载，虚函数与纯虚函数
```
class Base {  //定义基类
public:
  /*用关键字 virtual 声明虚函数，子类必须重载---实现虚函数
  用虚函数加文法 v=0 ，进一步声明纯虚函数 */
  产含纯虚函数的基类，称为抽象基类--实现虚函数
  virtual void DoSomething() =0 ;
  /* 含纯虚函数的基类，称为抽象基类
  抽象基类---相当于接口类，不能直接实例化创建对象 */
};
class Drived:public Base {  //定义继承基类 Base 的一个派生类
public:
    void DoSomething()  {    //所有派生类必须重载实现基类中定义的"虚函数" 。
        cout<<"virtual function"<<endl;
    }
};
```
虚函数与多态
```
class Shape { //定义基类--形状
public:
  virtual double area() const = 0;  //纯虚函数
};
class Circle:public Shape { //圆类，继承形状类
private:
  double radius;
public:
  Circle(double r) :radius(r) {}
  virtual double area()const { return radius * radius * 3.14; }
};
class Rectangle :public Shape { ... } ;   //长方形类，继承形状类
class Triangle :public Shape { ... };     //三角形类，继承形状类
void printArea(Shape& s) {
  cout << "面积为: " << s.area() << endl;}
```
OO 特性的优势与局限性
- OO 特性：封装、继承、重载、多态等。
- 多态是 OO 语言的一种特征，支持以类似的方式处理不同类型的对象：通过继承和虚函数实现子类型多态。
- OO 编程实现，通常由大量对象类组成，同应用的不同类之间，根据继承关系，形成类树形的层次化结构。
- 基于类实例化生成的各类对象，即使是同类的不同对象，具有不同的状态。它们通过协作[相互调用]完成面向过程中的任务。
  1. 这种主动的相互调用，不可避免会形成相互间强依赖与强耦合。
  2. 不利于实现—模块间松散耦合—的应用规范。
  3. 不利于实现—易修改、易维护—的应用品质。

克服 OO 局限性的方法
- 采用基于接口编程，使得不同对象间的相互依赖，变成对更为抽象的接口的依赖。
- 采用控制反转技术（IOC – Inversion of Control）
  1. 将设计好的类交给系统去控制，而不是在自己的类内部控制。
  2. 将对象的创建和获取—集中提取到类的外部，由外部（系统运行支撑平台的）容器自动创建，并提供需要的组件。
- 采用面向切面编程技术（AOP）
  1. 类似于引入代理、wrap 管理类方法，剥离/分离非核心业务的其它辅助功能处理。
  2. 把在不同逻辑顺序点执行的辅助功能，分别独立实现，并通过执行逻辑链的"相关切点"，把这些辅助功能函数关联在一起。

GUI

3.1 12 Display
```
int main() {
  using namespace Graph_lib;  // use our graphics interface library

  Point tl(100,200);       // a point (obviously)

  Simple_window win(tl,600,400,"Canvas");   // make a simple window

  Polygon poly;   // make a shape (a polygon, obviously)

  poly.add(Point(300,200)); // add three points
  poly.add(Point(350,100));
  poly.add(Point(400,200));

  poly.set_color(Color::red);     // make the polygon red (obviously)

  win.attach(poly);   // connect poly to the window

  win.wait_for_button();  // give control to the display engine
}
```
Graphics/GUI 库，our code → our interface library → FLTK → OS
```
// Getting access to the graphics system (don't forget to install):
// Demo 1
#include "Simple_window.h"  // stuff to deal with your system's windows
#include "Graph.h"    // graphical shapes
using namespace Graph_lib;  // make names available
// in main():
Simple_window win(Point(100,100),600,400,"Canvas");
    // screen coordinate (100,100) top left of window
    // window size(600*400)
    // title: Canvas
win.wait_for_button();  // Display!

// Demo2
Axis xa(Axis::x, Point(20,300), 280, 10, "x axis");
  // make an Axis
    // an axis is a kind of Shape
    // Axis::x means horizontal
    // starting at (20,300)
    // 280 pixels long
    // 10 "notches"
    // text "x axis
win.set_label("Canvas #2");
win.attach(xa); // attach axis xa to the window
win.wait_for_button();
```

3.2 13 Graphics Classes

Line
```
struct Shape {
};
class Line : Shape {
public:
  Line(Point p1, Point p2);
};

Line::Line(Point p1, Point p2) {  // construct a line from p1 to p2
  add(p1);  // add p1 to this shape (add() is provided by Shape)
  add(p2);  // add p2 to this shape
}
Line horizontal(Point(100,100),Point(200,100));
Line vertical(Point(150,50),Point(150,150));
```

STL

4.1 vector
```
class vector {
  int sz;
  double* elem;
public:
  vector(const vector&);
};
vector::vector(const vector& a):sz(a.sz),  elem(new double[a.sz]) {
  for (int i = 0; i<sz; ++i) elem[i] = a.elem[i];
}
```
拷贝赋值
```
vector& vector::operator=(const vector& a)
  // like copy constructor, but we must deal with old elements
  // make a copy of a then replace the current sz and elem with a's {
  double* p = new double[a.sz];     // allocate new space
  for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i];  // copy elements
  delete[ ] elem;       // deallocate old space
  sz = a.sz;          // set new size
  elem = p;         // set new elements
  return *this;     //  return a self-reference
        // The this pointer is explained in Lecture 19
        // and in 17.10
}
```
浅拷贝：仅拷贝指针，因此两个指针可能指向同一个对象 指针和引用都是如此 深拷贝：拷贝指针及其指向的数据，因此两个指针分别指向两个不同的 就是 vector, string, etc. 的处理方式 对容器类，需要拷贝构造函数和拷贝赋值函数
```
vector glob(10);       // global vector-"lives" forever
vector* some_fct(int n) {
  vector v(n);       // local vector-"lives" until the end of scope
  vector* p = new vector(n);  // free-store vector-"lives" until we delete it
  return p;
}
void f() {
  vector* pp = some_fct(17);
  delete pp;  // deallocate the free-store vector allocated in some_fct()
}
```
很容易忘记 delete 分配的自由存储 因此，尽量避免使用 new/delete
```
vector& vector::operator=(const vector& a) {
  if (this==&a) return *this; // self-assignment, no work needed
  if (a.sz<=space) {    // enough space, no need for new allocation
    for (int i = 0; i<a.sz; ++i) elem[i] = a.elem[i]; // copy elements
    sz = a.sz;                                  //不可或缺
    return *this;                               // a key statement
  }
  double* p = new double[a.sz];  //没有足够条件,只好copy and swap
  for (int i = 0; i<a.sz; ++i) p[i] = a.elem[i];
  delete[ ] elem;
  sz = a.sz;
  space = a.sz;
  elem = p;
  return *this;
}
```
模板是 C++ 中泛型编程的基础 有时称为"参数化多态" 通过将类型和整数指定模板类型参数的参数化 当性能要求高时使用它 (e.g., 硬实时和数值计算) 当灵活性要求高时使用它 (e.g., C++ 标准库) 模板定义 (语法) `template<class T, int N> class Buffer { /* … */ };`//类型模板 `template<class T, int N> void fill(Buffer<T,N>& b) { /*...*/  }`//函数模板

STL: vector, list, map, set(平衡二叉树), queue, stack, priority_queue, deque, iterator, algorithm, function object, etc. 序列容器：vector,list,deque 关联容器： map,set,multimap,multiset 近似容器：array,string,stack,queue,priority_queue
```
template<class T> class vector {
  T& at(int n);      // checked access
  T& operator[ ](int n);    // unchecked access
};
template<class T>  T& vector<T>::at (int n) {
  if (n<0 || sz<=n) throw out_of_range();
  return elem[n];
}
template<class T>  T& vector<T>::operator[ ](int n)
{
  return elem[n];
}
```
泛型编程 对终端用户，其目的是
- 提高正确性：通过更好地规范化
- 更大的使用范围：可能的重用，有良好的适应性
- 更好的性能：通过使用完善的库 不必要的慢速代码将会逐渐被取消
- 路线策略：从具体到抽象：其他方式通常会导致实现膨胀

模板是 C++ 中泛型编程的基础—"参数化多态"：泛元素类型、泛容器、泛操作条件（函数对象）

迭代器：size(),insert();erase(),front(),back(), push_back(),pop_back() 等操作
```
template<class T> class list {
  Link* elements;
public:
  class iterator;
  iterator begin();   // points to first element
  const_iterator begin() const;
  iterator end();   // points one beyond the last element
  const_iterator end() const;
  iterator erase(iterator p);   // remove element pointed to by p
  iterator insert(iterator p, const T& v);  // insert a new element v before p
        Elem& front();
        Elem& back();
};
```
List 迭代器
```
template<class Elem> class list<Elem>::iterator {
  Link* curr;       //current link
public:
  iterator(Link* p):curr(p){};
  iterator& operator++(){curr=curr->succ; return *this};  // forward
  iterator& operator--()  {curr=curr->prec; return *this};  // backward
  Elem& operator*(){return cur->val}; // get value (dereference)

  bool operator==(const iterator& b) const {return curr==b.curr;}
  bool operator!=(const iterator& b) const {return curr!=b.curr;}
};
```
vector 类型
```
template<class T, class A=allocator<T> >class vector
{
    int  sz;                          // the size
    T*  elem;                       // a pointer to the elements
     int space;
    A    alloc;                       // use allocate to handle memory for elements
Public:
  typedef  unsigned long size_type;
  typedef T   value_type;
  typedef T* iterator;  // the type of an iterator is implementation defined
        // a vector iterator could be a pointer to an element
  typedef  const T* const_iterator;
  iterator begin();   // points to first element
  const_iterator begin() const;
  iterator end();   // points one beyond the last element
  const_iterator end() const;
  iterator erase(iterator p);   // remove element pointed to by p
  iterator insert(iterator p, const T& v);  // insert a new element v before p
};
```
vector 快速字典
```
int main() {
  string from, to;
  cin >> from >> to;  // get source and target file names
  ifstream is(from.c_str()); // open input stream
  ofstream os(to.c_str());  // open output stream
  istream_iterator<string> ii(is);  // make input iterator for stream
  istream_iterator<string> eos; // input sentinel 结束标志 (defaults to EOF)
  ostream_iterator<string> oo(os,"\n"); // make output iterator for stream
          // append "\n" each time
  vector<string> b(ii,eos);   // b is a vector initialized from input
  sort(b.begin() ,b.end());     // sort the buffer
  unique_copy(b.begin() ,b.end() ,oo);  // copy buffer to output,
    // discard replicated values
}
```
erase()操作
```
template<class T, class A>
vector<T,A>::iterator vector<T,A>::erase (iterator p) {
  if (p==end()) return p;
  for (iterator pos = p+1; pos!=end(); ++pos)
      *(pos-1) = *pos;      //拷贝元素到其左边的一个位置
  alloc.destroy(&*(end()-1));  //连带拷贝地销毁最后一个单元
  return p;
};
```
find
```
template<class In, class T>
In find(In first, In last, const T& val)  { //风格一
  while (first!=last && *first != val) ++first;
  return first;
}
```

4.2 map
```
template<class Key, class Value> class map {
  typedef pair<Key,Value> value_type; // a map deals in (Key,Value) pairs

  typedef ??? iterator;   // probably a pointer to a tree node
  typedef ??? const_iterator;

  iterator begin();   // points to first element
  iterator end();   // points to one beyond the last element

  Value& operator[ ](const Key&);

  iterator find(const Key& k);  // is there an entry for k?

  void erase(iterator p);   // remove element pointed to by p
  pair<iterator, bool> insert(const value_type&);  // insert a new pair before p

};
dow_weight.insert(make_pair("MMM", 5.8549));
```
快速字典(set)
```
int main() {
  string from, to;
  cin >> from >> to;                          // get source and target file names

  ifstream is(from.c_str());                  // make input stream
  ofstream os(to.c_str());      // make output stream

  istream_iterator<string> ii(is);           // make input iterator for stream
  istream_iterator<string> eos;              // input sentinel (defaults to EOF)
  ostream_iterator<string> oo(os,"\n"); // make output iterator for stream
          // append "\n" each time
  set<string> b(ii,eos);       // b is a set initialized from input
  copy(b.begin() ,b.end() ,oo);     // copy buffer to output
}
```

八股文

- 封装：把客观事物抽象为类，包含自己的属性和方法。
- 继承：使用现有类的所有功能，在无需重新编写原有类的情况下对类的功能进行拓展。被继承的类成为父类或基类，继承的类成为子类或派生类。
- 多态：一种形式，多种状态，分为静态多态和动态多态。静态多态指编译时多态，如函数重载、模板；动态多态指运行时多态，特指 virtual 虚函数机制形成的多态。

类中可以直接访问自己类的 public、protected、private 成员，但类对象只能访问自己类的 public 成员。

- 构造函数：一个类对象创建时自动调用的方法，用来完成初始化得工作。没有返回值，函数名与类名相同。
- 析构函数：当类对象生命周期结束时，自动销毁对象占有的内存。
- 拷贝构造函数：参数为类的引用，即用一个类对象复制构造一个新的类对象。
- 赋值函数：将一个类对象成员的值赋给当前对象。

1. 拷贝构造函数的形参是一个左值引用，而移动构造函数的形参是一个右值引用
2. 拷贝构造函数完成的是整个对象或变量的拷贝，而移动构造函数是生成一个指针指向源对象或变量的地址，接管源对象的内存，相对于大量数据的拷贝节省时间和内存空间。

- 浅拷贝：利用类提供的默认拷贝构造函数，将一个对象的成员所在内存的数据复制给另一个对象的成员。
- 深拷贝：显式定义类的拷贝构造函数，不仅会将原对象的成员变量复制给新对象，还会在堆中为新对象分配一块新的内存，并将原对象持有的动态内存资源也拷贝过来。

空类有哪些函数，空类的大小：
- 一个空类包括默认构造函数、拷贝构造函数、默认赋值运算符、默认取值运算符、析构函数。
- 空类也可以实例化，在内存会有独一无二的地址，编译器规定空类的大小为 1 字节；
- 仅含有一个虚函数的类的大小为 4 字节，因为虚表指针占有 4 个字节地址。

C++内存分区：
- 全局区：存储全局变量和静态变量，编译时分配，存在整个程序运行期间。
- 堆区：程序员手动管理的内存区间，运行时分配，手动开辟、释放内存。
- 栈区：存储局部变量，运行时分配，系统自动管理内存的开辟和释放。
- 常量区：存放常量，编译时分配，存在整个程序运行期间。
- 代码区：存放程序运行的 cpu 指令，编译时分配，系统自动管理内存的开辟和释放。

extern C {} ：告诉编译器这段代码是用 C 语言写的，不要进行 C++的编译处理。

C++ struct 和 class 的唯一区别： struct 中成员默认的访问权限是 public，class 中成员默认的访问权限是 private。

define 和 const 的区别：
- 处理时机不同：宏定义是"编译时"概念，在预处理阶段展开，生命周期结束于编译时期；const 是"运行时"概念，在程序运行时使用。
- 存储方式不同：宏定义直接替换，不会分配内存，存储在程序的代码区中；const 变量需要进行内存分配。
- 类型和安全检查不同：宏定义是字符替换，没有数据类型的区别，没有类型安全检查；const 是常量的声明，有类型区别，编译时会进行安全检查。

四种类型转换：
1. static_cast：用于编译器无法自动执行的类型转换；
2. dynamic_cast：将一个基类指针（或引用）转换为派生类指针（或引用），一般用于有继承关系的类中；
3. const_cast：将 const 修饰的常量转换为非常量，一般用于重载函数中；
4. reinterpret_cast：将变量从类型 1 转换为类型 2，编译器会当作类型 2，真实类型为类型 1。（不建议使用）

vector, list, deque 对比：
1. 相当于一个动态数组，可实现容器长度的变化，在内存上占有一段连续的地址空间；

擅长随机访问以及在容器尾部添加或删除元素，效率较高； 在容器头部及中间位置插入或删除元素，需要移动该位置以后的所有元素，效率较低；
1. 容器中各个元素的前后顺序靠指针联系在一起，在内存中不一定是连续的地址空间；

擅长随机插入或删除元素，只改变前后两个元素的联系，效率高；不擅于随机访问元素，遍历容器需要访问大量指针，效率极低；
1. 将多个连续内存空间通过指针数组连接在一起，不能保证所有元素在同一段连续的地址空间中；

容器的地址空间是动态变化的，可以向两端进行伸缩，内部实现比较复杂： 擅长在容器首部、尾部插入或删除元素，效率高；