Advanced Programming

ACSE-5: Lecture 5 – Overview Slides

Adriana Paluszny

These slides

- This is a selection of Bjarne Stroustrup’s C++ slides
- They should function as a revision tool for the content covered in class thus far
- For those that prefer a shorter version of the book content
- Except for content on “Errors” all the material should seem familiar, and even “easy”, and should serve as a mechanism to check your basic understanding of the language
- In class we will cover new material

Computation

Bjarne Stroustrup's start of slides

Original can be found in www.stroustrup.com

- Our job is to express computations
  - Correctly
  - Simply
  - Efficiently
- One tool is called Divide and Conquer
  - to break up big computations into many little ones
- Another tool is Abstraction
  - Provide a higher-level concept that hides detail
- Organization of data is often the key to good code
  - Input/output formats
  - Protocols
  - Data structures
- Note the emphasis on structure and organization
  - You don't get good code just by writing a lot of statements

Expressions

```
// compute area:
int length = 20;                    // the simplest expression: a literal (here, 20)
                                    // (here used to initialize a variable)
int width = 40;
int area = length*width;            // a multiplication
int average = (length+width)/2;     // addition and division
```

The usual rules of precedence apply:

a*b+c/d means **(a*b)+(c/d)** and not **a*(b+c)/d**.

If in doubt, parenthesize. If complicated, parenthesize.

Don't write "absurdly complicated" expressions:

```
a*b+c/d*(e-f/g)/h+7                 // too complicated
```

Choose meaningful names.

Expressions

- Expressions are made out of operators and operands
  - Operators specify what is to be done
  - Operands specify the data for the operators to work with

- Boolean type: **bool** (**true** and **false**)
  - Equality operators: **= =** (equal), **!=** (not equal)
  - Logical operators: **&&** (and), **||** (or), **!** (not)
  - Relational operators: **<** (less than), **>** (greater than), **<=, >=**
- Character type: **char** (e.g., **'a'**, **'7'**, and **'@'**)
- Integer types: **short, int, long**
  - arithmetic operators: **+, -, *, /, %** (remainder)
- Floating-point types: e.g., **float, double** (e.g., **12.45** and **1.234e3**)
  - arithmetic operators: **+, -, *, /**

Concise Operators

- For many binary operators, there are (roughly) equivalent more concise operators
  - For example
    - **a += c** means **a = a+c**
    - **a *= scale** means **a = a*scale**
    - **++a** means **a += 1** or **a = a+1**
  - “Concise operators” are generally better to use (clearer, express an idea more directly)

Statements

- A statement is
  - an expression followed by a semicolon, or
  - a declaration, or
  - a “control statement” that determines the flow of control
- For example
  - **a = b;**
  - **double d2 = 2.5;**
  - **if (x == 2) y = 4;**
  - **while (cin >> number) numbers.push_back(number);**
  - **int average = (length+width)/2;**
  - **return x;**
- You may not understand all of these just now, but you will …

Selection

- Sometimes we must select between alternatives
- For example, suppose we want to identify the larger of two values. We can do this with an **if** statement

```
if (a<b)                // Note:  No semicolon here
    max = b;
else                    // Note:  No semicolon here
    max = a;
```

- The syntax is

```
if (condition)
    statement-1     // if the condition is true, do statement-1
else
    statement-2     // if not, do statement-2
```

Iteration (while loop)

- The world’s first “real program” running on a stored-program computer (David Wheeler, Cambridge, May 6, 1949)

```
// calculate and print a table of squares 0-99:
int main()
{
    int i = 0;
    while (i<100) {
        cout << i << '\t' << square(i) << '\n';
        ++i ;       // increment i
    }
}
// (No, it wasn’t actually written in C++ ☺.)
```

Iteration (while loop)

- What it takes
  - A loop variable (control variable); here: **i**
  - Initialize the control variable; here: **int i = 0**
  - A termination criterion; here: if **i<100** is false, terminate
  - Increment the control variable; here: **++i**
  - Something to do for each iteration; here: **cout << …**

```
int i = 0;
while (i<100)  {
        cout << i << '\t' << square(i) << '\n';
        ++i ;       // increment i
}
```

Iteration (for loop)

- Another iteration form: the **for** loop
- You can collect all the control information in one place, at the top, where it's easy to see

```
for (int i = 0; i<100; ++i) {
    cout << i << '\t' << square(i) << '\n';
}
```

That is,

for (initialize**;** condition **;** increment **)**
controlled statement

Note: what is **square(i)**?

Functions

- But what was **square(i)?**
  - A call of the function **square()**

```
int square(int x)
{
    return x*x;
}
```

  - We define a function when we want to separate a computation because it
    - is logically separate
    - makes the program text clearer (by naming the computation)
    - is useful in more than one place in our program
    - eases testing, distribution of labor, and maintenance

Control Flow

```
int main()
{
        i=0;

    while (i<100)
    {
i<100
            square(i);

    }
}       i==100
```

```
int square(int x)
{
    // compute square root
    return x * x;
}
```

Functions

- Our function

```
int square(int x)
{
    return x*x;
}
```

is an example of

```
Return_type  function_name ( Parameter list )
                                              // (type name, etc.)
{
    // use each parameter in code
    return some_value;                        // of Return_type
}
```

Another Example

- Earlier we looked at code to find the larger of two values. Here is a function that compares the two values and returns the larger value.

```
int max(int a, int b)  // this function takes 2 parameters
{
    if (a<b)
        return b;
    else
        return a;
}

int x = max(7, 9); // x becomes 9
int y = max(19, -27);       // y becomes 19
int z = max(20, 20);        // z becomes 20
```

Data for Iteration - Vector

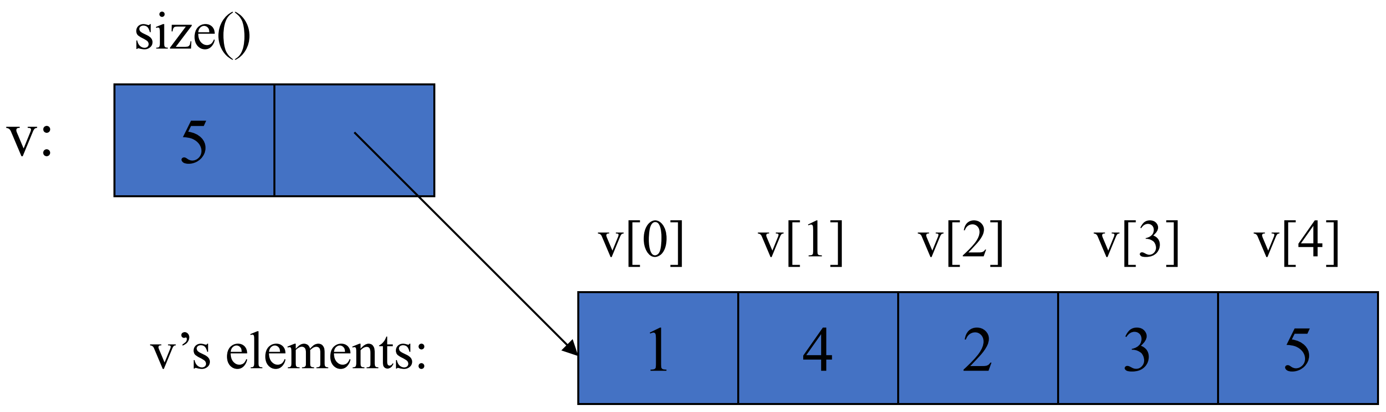
- To do just about anything of interest, we need a collection of data to work on. We can store this data in a **vector**. For example:

```
// read some temperatures into a vector:
int main()
{
    vector<double> temps;        // declare a vector of type double to store
                                 // temperatures – like 62.4
    double temp;                 // a variable for a single temperature value
    while (cin>>temp)    // cin reads a value and stores it in temp
        temps.push_back(temp);          // store the value of temp in the vector
    // … do something …
}
// cin>>temp  will return true until we reach the end of file or encounter
// something that isn't a double: like the word "end"
```

Vector

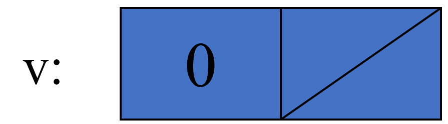
- Vector is the most useful standard library data type
  - a **vector<T>** holds an sequence of values of type **T**
  - Think of a vector this way

    A vector named **v** contains 5 elements: {1, 4, 2, 3, 5}:

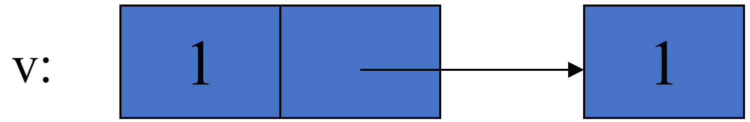


Vectors

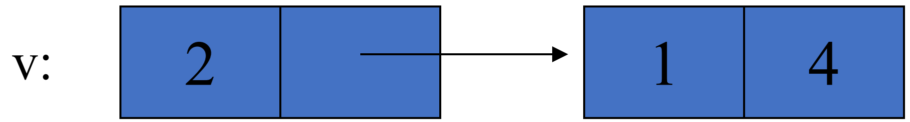
vector<int> v; // start off empty



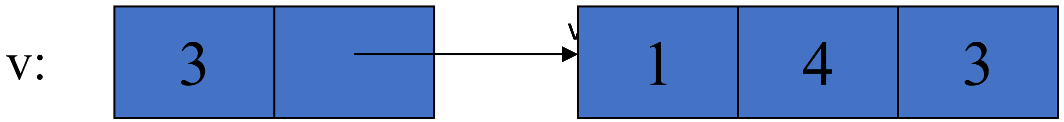
v.push_back(1); // add an element with the value **1**



v.push_back(4); // add an element with the value **4** at end (“the back”)



v.push_back(3); // add an element with the value **3** at end (“the back”)



Vectors

- Once you get your data into a vector you can easily manipulate it

```
// compute mean (average) and median temperatures:
int main()
{
   vector<double> temps;        // temperatures in Fahrenheit, e.g. 64.6
   double temp;
   while (cin>>temp)  temps.push_back(temp); // read and put into vector

   double sum = 0;
   for (int i = 0; i< temps.size(); ++i) sum += temps[i];  // sums temperatures

   cout << "Mean temperature: " << sum/temps.size() << '\n';
   sort(temps);             // from std_lib_facilities.h
                            // or sort(temps.begin(), temps.end();
   cout << "Median temperature: " << temps[temps.size()/2] << '\n';
}
```

Traversing a vector

- Once you get your data into a vector you can easily manipulate it
- Initialize with a list
  - **vector<int> v = { 1, 2, 3, 5, 8, 13 };** // *initialize with a list*
- *often we want to look at each element of a vector in turn:*

```
for (int i = 0; i< v.size(); ++i) cout << v[i] << '\n';  // list all elements

// there is a simpler kind of loop for that (a range-for loop):

for (int i : v) cout << x << '\n';  // list all elements

// for each x in v …
```

Combining Language Features

- You can write many new programs by combining language features, built-in types, and user-defined types in new and interesting ways.
  - So far, we have
    - Variables and literals of types **bool, char, int, double**
    - **vector, push_back(), [ ]** (subscripting)
    - **!=, ==, =, +, -, +=, <, &&, ||, !**
    - **max( ), sort( ), cin>>, cout<<**
    - **if, for, while**
  - You can write a lot of different programs with these language features! Let’s try to use them in a slightly different way…

Example – Word List

// “boilerplate” left out

```
vector<string> words;
for (string s; cin>>s && s != "quit“; )                 // && means AND
        words.push_back(s);

sort(words);                                            // sort the words we read

for (string s : words)
        cout << s << '\n';

/*
   read a bunch of strings into a vector of strings, sort
   them into lexicographical order (alphabetical order),
   and print the strings from the vector to see what we have.
*/
```

Word list – Eliminate Duplicates

// Note that duplicate words were printed multiple times. For
// example "the the the". That's tedious, let's eliminate duplicates:

```
vector<string> words;
for (string s; cin>>s && s!= "quit"; )
        words.push_back(s);

sort(words);

for (int i=1; i<words.size(); ++i)
        if(words[i-1]==words[i])
                "get rid of words[i]"     // (pseudocode)
for (string s : words)
        cout << s << '\n';
```

// there are many ways to "get rid of words[i]"; many of them are messy
// (that's typical). Our job as programmers is to choose a simple clean
// solution – given constraints – time, run-time, memory.

Example (cont.) Eliminate Words!

```
// Eliminate the duplicate words by copying only unique words:
    vector<string> words;
    for (string s; cin>>s && s!= "quit"; )
            words.push_back(s);
    sort(words);
    vector<string>w2;
    if (0<words.size()) {                          // note style { }
            w2.push_back(words[0]);
            for (int i=1; i<words.size(); ++i)    // note: not a range-for
                    if(words[i-1]!=words[i])
                     w2.push_back(words[i]);
    }
    cout<< "found " << words.size()-w2.size() <<  " duplicates\n";
    for (string s : w2)
            cout << s << "\n";
```

Algorithm

- We just used a simple algorithm
- An algorithm is (from Google search)
  - "a logical arithmetical or computational procedure that, if correctly applied, ensures the solution of a problem." – *Harper Collins*
  - "a set of rules for solving a problem in a finite number of steps, as for finding the greatest common divisor." – *Random House*
  - "a detailed sequence of actions to perform or accomplish some task. Named after an Iranian mathematician, Al-Khawarizmi. Technically, an algorithm must reach a result after a finite number of steps, …The term is also used loosely for any sequence of actions (which may or may not terminate)." – *Webster's*
- We eliminated the duplicates by first sorting the vector (so that duplicates are adjacent), and then copying only strings that differ from their predecessor into another vector.

Errors

- " … I realized that from now on a large part of my life would be spent finding and correcting my own mistakes."
  - Maurice Wilkes, 1949
- When we write programs, errors are natural and unavoidable; the question is, how do we deal with them?
  - Organize software to minimize errors.
  - Eliminate most of the errors we made anyway.
    - Debugging
    - Testing
  - Make sure the remaining errors are not serious.
- My guess is that avoiding, finding, and correcting errors is 95% or more of the effort for serious software development.
  - You can do much better for small programs.
    - or worse, if you're sloppy

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)

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
```

Bad Function Arguments

- So, how about **int x = area(10, -7);** ?
- Alternatives
  - Just don't do that
    - Rarely a satisfactory answer
  - The caller should check
    - Hard to do systematically
  - The function should check
    - Return an "error value" (not general, problematic)
    - Set an error status indicator (not general, problematic – don't do this)
    - Throw an exception
- Note: sometimes we can't change a function that handles errors in a way we do not like
  - Someone else wrote it and we can't or don't want to change their code

Bad function arguments

- Why worry?
  - You want your programs to be correct
  - Typically the writer of a function has no control over how it is called
    - Writing “do it this way” in the manual (or in comments) is no solution – many people don’t read manuals
  - The beginning of a function is often a good place to check
    - Before the computation gets complicated
- When to worry?
  - If it doesn’t make sense to test every function, test some

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
```

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"

Debugging

- How ***not*** to do it

```
while (program doesn't appear to work) { // pseudo code
        Randomly look at the program for something that "looks odd"
        Change it to "look better"
}
```

- Key question

  How would I know if the program actually worked correctly?

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?

Debugging

- "If you can't see the bug, you're looking in the wrong place"
  - It's easy to be convinced that you know what the problem is and stubbornly keep looking in the wrong place
  - Don't just guess, be guided by output
    - Work forward through the code from a place you know is right
      - so what happens next? Why?
    - Work backwards from some bad output
      - how could that possibly happen?
- Once you have found "the bug" carefully consider if fixing it solves the whole problem
  - It's common to introduce new bugs with a "quick fix"
- "I found the last bug"
  - is a programmer's joke

Note

- Error handling is fundamentally more difficult and messy than “ordinary code”
  - There is basically just one way things can work right
  - There are many ways that things can go wrong
- The more people use a program, the better the error handling must be
  - If you break your own code, that’s your own problem
    - And you’ll learn the hard way
  - If your code is used by your friends, uncaught errors can cause you to lose friends
  - If your code is used by strangers, uncaught errors can cause serious grief
    - And they may not have a way of recovering

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

- How do we test a program?
  - Be systematic
    - “pecking at the keyboard” is okay for very small programs and for very initial tests, but is insufficient for real systems
  - Think of testing and correctness from the very start
    - When possible, test parts of a program in isolation
      - E.g., when you write a complicated function write a little program that simply calls it with a lot of arguments to see how it behaves in isolation before putting it into the real program (this is typically called “unit testing”)
  - We’ll return to this question in Chapter 26

A mystery

- Expect “mysteries”
- Your first try rarely works as expected
  - That’s normal and to be expected
    - Even for experienced programmers
  - If it looks as if it works be suspicious
    - And test a bit more
  - Now comes the debugging
    - Finding out why the program misbehaves
  - And don’t expect your second try to work either

Remove "magic constants"

- But what's wrong with "magic constants"?
  - Everybody knows **3.14159265358979323846264**, **12, -1**, **365**, **24, 2.7182818284590**, **299792458**, **2.54**, **1.61,** **-273.15**, **6.6260693e-34, 0.5291772108e-10,** **6.0221415e23** and **42**!
  - No; they don't.
- "Magic" is detrimental to your (mental) health!
  - It causes you to stay up all night searching for bugs
  - It causes space probes to self destruct (well … it can … sometimes …)
- If a "constant" could change (during program maintenance) or if someone might not recognize it, use a symbolic constant.
  - Note that a change in precision is often a significant change;

    3.14 !=3.14159265
  - **0** and **1** are usually fine without explanation, **-1** and **2** sometimes (but rarely) are.
  - **12** can be okay (the number of months in a year rarely changes), but probably is not (see Chapter 10).
- If a constant is used twice, it should probably be symbolic
  - That way, you can change it in one place

Language technicalities

- Are a necessary evil
  - A programming language is a foreign language
  - When learning a foreign language, you have to look at the grammar and vocabulary
  - We will do this in this chapter and the next
- Because:
  - Programs must be precisely and completely specified
    - A computer is a very stupid (though very fast) machine
    - A computer can't guess what you "really meant to say" (and shouldn't try to)
  - So we must know the rules
    - Some of them (the C++14 standard is 1,358 pages)
- However, never forget that
  - What we study is programming
  - Our output is programs/systems
  - A programming language is only a tool

Technicalities

- Don’t spend your time on minor syntax and semantic issues. There is more than one way to say everything
  - Just like in English
- Most design and programming concepts are universal, or at least very widely supported by popular programming languages
  - So what you learn using C++ you can use with many other languages
- Language technicalities are specific to a given language
  - But many of the technicalities from C++ presented here have obvious counterparts in C, Java, C#, etc.

Declarations

- A declaration introduces a name into a scope.
- A declaration also specifies a type for the named object.
- Sometimes a declaration includes an initializer.
- A name must be declared before it can be used in a C++ program.
- Examples:
  - **int a = 7;** // *an int variable named 'a' is declared*
  - **const double cd = 8.7;** // *a double-precision floating-point constant*
  - **double sqrt(double);** // *a function taking a double argument and* // *returning a double result*
  - **vector<Token> v;** // *a vector variable of **Token**s (variable)*

Declarations

- Declarations are frequently introduced into a program through “headers”
    - A header is a file containing declarations providing an interface to other parts of a program
- This allows for abstraction – you don’t have to know the details of a function like **cout** in order to use it. When you add

```
#include "std_lib_facilities.h"
```

  to your code, the declarations in the file **std_lib_facilities.h** become available (including **cout**, etc.).

For example

- At least three errors:

```
int main()
{
    cout << f(i) << '\n';
}
```

- Add declarations:

```
#include "std_lib_facilities.h"     // we find the declaration of cout in here

int main()
{
    cout << f(i) << '\n';
}
```

For example

- Define your own functions and variables:

```
#include ”std_lib_facilities.h”     // we find the declaration of cout in here

int f(int x ) { /* … */ }     // declaration of f

int main()
{
    int i = 7;     // declaration of i
    cout << f(i) << ’\n‘;
}
```

Definitions

A declaration that (also) fully specifies the entity declared is called a definition

- Examples

```
int a = 7;
int b;                        // an (uninitialized) int
vector<double> v;             // an empty vector of doubles
double sqrt(double) { … };    // a function with a body
struct Point { int x; int y; };
```

- Examples of declarations that are not definitions

```
double sqrt(double);          // function body missing
struct Point;                 // class members specified elsewhere
extern int a;                 // extern means "not definition"
                              // "extern" is archaic; we will hardly use it
```

Declarations and definitions

- You can't *define* something twice
  - A definition says what something is
  - Examples

```
int a;                       // definition
int a;                       // error: double definition
double sqrt(double d) { … } // definition
double sqrt(double d) { … } // error: double definition
```

- You can *declare* something twice
  - A declaration says how something can be used

```
int a = 7;                   // definition (also a declaration)
extern int a;                // declaration
double sqrt(double);         // declaration
double sqrt(double d) { … } // definition (also a declaration)
```

Why both declarations and definitions?

- To refer to something, we need (only) its declaration
- Often we want the definition “elsewhere”
  - Later in a file
  - In another file
    - preferably written by someone else
- Declarations are used to specify interfaces
  - To your own code
  - To libraries
    - Libraries are key: we can’t write all ourselves, and wouldn’t want to
- In larger programs
  - Place all declarations in header files to ease sharing

Kinds of declarations

- The most interesting are
  - Variables
    - **int x;**
    - **vector<int> vi2 {1,2,3,4};**
  - Constants
    - **void f(const X&);**
    - **constexpr int = isqrt(2);**
  - Functions (see §8.5)
    - **double sqrt(double d) { /*** … ***/ }**
  - Namespaces (see §8.7)
  - Types (classes and enumerations; see Chapter 9)
  - Templates (see Chapter 19)

Header Files and the Preprocessor

- A header is a file that holds declarations of functions, types, constants, and other program components.
- The construct

  #include "std_lib_facilities.h"

  is a “preprocessor directive” that adds declarations to your program
  - Typically, the header file is simply a text (source code) file
- A header gives you access to functions, types, etc. that you want to use in your programs.
  - Usually, you don’t really care about how they are written.
  - The actual functions, types, etc. are defined in other source code files
    - Often as part of libraries

Source files

token.h:

```
// declarations:
class Token { … };
class Token_stream {
    Token get();
    …
};
extern Token_stream ts;
…
```

token.cpp:

```
#include "token.h"
//definitions:
Token Token_stream::get()
{ /* … */ }
Token_stream ts;
…
```

use.cpp:

```
#include "token.h"
…
Token t = ts.get();
…
```

- A header file (here, **token.h**) defines an interface between user code and implementation code (usually in a library)
- The same **#include** declarations in both **.cpp** files (definitions and uses) ease consistency checking

Scope

- A scope is a region of program text
  - Global scope (outside any language construct)
  - Class scope (within a class)
  - Local scope (between { … } braces)
  - Statement scope (e.g. in a for-statement)
- A name in a scope can be seen from within its scope and within scopes nested within that scope
  - Only after the declaration of the name (“can’t look ahead” rule)
  - Class members can be used within the class before they are declared
- A scope keeps “things” local
  - Prevents my variables, functions, etc., from interfering with yours
  - Remember: real programs have **many** thousands of entities
  - Locality is good!
    - Keep names as local as possible

Scope

```
#include "std_lib_facilities.h"            // get max and abs from here
// no r, i, or v here
class My_vector {
    vector<int> v;                          // v is in class scope
public:
    int largest()                                   // largest is in class scope
    {
            int r = 0;                      // r is local
            for (int i = 0; i<v.size(); ++i)  // i is in statement scope
                    r = max(r,abs(v[i]));
            // no i here
            return r;
    }
    // no r here
};
// no v here
```

Scopes nest

```
int x;      // global variable – avoid those where you can
int y;      // another global variable

int f()
{
    int x;                        // local variable (Note – now there are two x's)
    x = 7;              // local x, not the global x
    {
            int x = y;  // another local x, initialized by the global y
                                  // (Now there are three x's)
            ++x;                  // increment the local x in this scope
    }
}

// avoid such complicated nesting and hiding: keep it simple!
```

Recap: Why functions?

- Chop a program into manageable pieces
  - “divide and conquer”
- Match our understanding of the problem domain
  - Name logical operations
  - A function should do one thing well
- Functions make the program easier to read
- A function can be useful in many places in a program
- Ease testing, distribution of labor, and maintenance
- Keep functions small
  - Easier to understand, specify, and debug

Functions

- General form:
  - **return_type *name* (*formal arguments*);** **//** *a declaration*
  - **return_type *name* (*formal arguments*) *body*** **//** *a definition*
  - For example

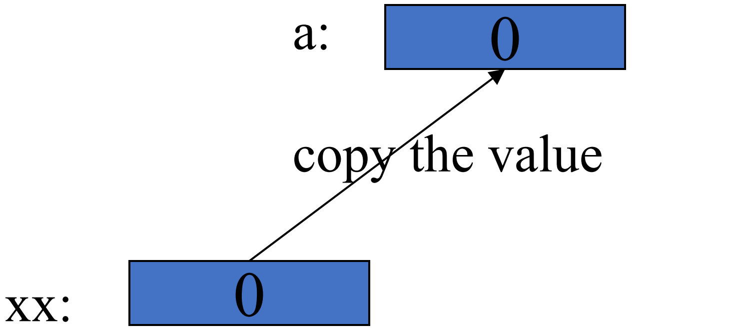
    ```
    double f(int a, double d) { return a*d; }
    ```
- Formal arguments are often called parameters
- If you don't want to return a value give **void** as the return type

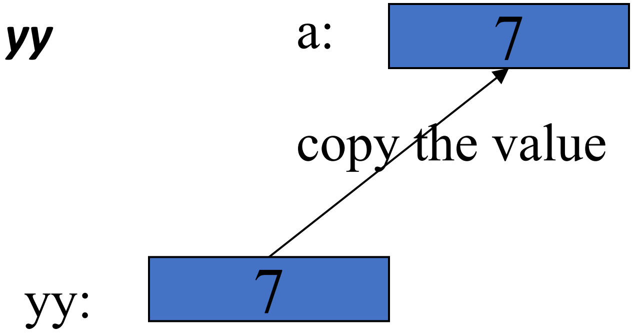
  ```
  void increase_power_to(int level);
  ```
  - Here, **void** means "doesn't return a value"
- A body is a block or a try block
  - For example

    ```
    { /* code */ }        // a block
    try { /* code */ } catch(exception& e) { /* code */ }   // a try block
    ```
- Functions represent/implement computations/calculations

Functions: Call by Value

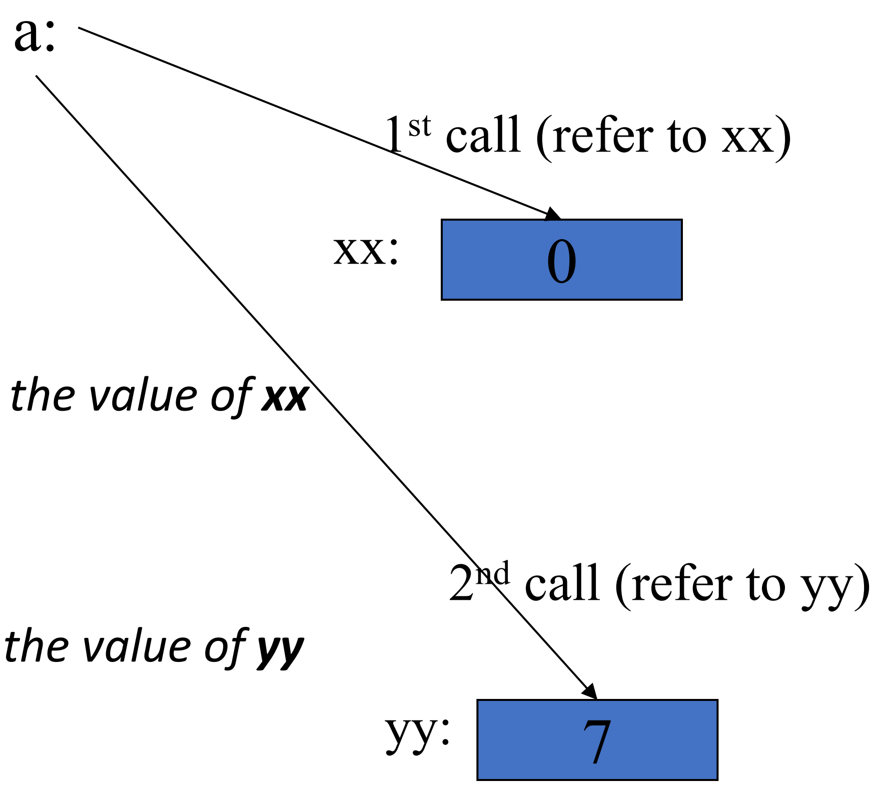
```
// call-by-value (send the function a copy of the argument's value)
int f(int a) { a = a+1; return a; }


int main()
{
    int xx = 0;
    cout << f(xx) << '\n';       // writes 1
    cout << xx << '\n';          // writes 0; f() doesn't change xx
    int yy = 7;
    cout << f(yy) << '\n'; // writes 8;  f() doesn't change yy
    cout << yy << '\n';          // writes 7
}
```





Functions: Call by Reference

```
// call-by-reference (pass a reference to the argument)
int f(int& a) { a = a+1; return a; }


int main()
{
   int xx = 0;
   cout << f(xx) << '\n';       // writes 1
                                // f() changed the value of xx
   cout << xx << '\n';          // writes 1
   int yy = 7;
   cout << f(yy) << '\n'; // writes 8
                                // f() changes the value of yy
   cout << yy << '\n';          // writes 8

}
```



Functions

- Avoid (non-const) reference arguments when you can
  - They can lead to obscure bugs when you forget which arguments can be changed

```
int incr1(int a) { return a+1; }
void incr2(int& a) { ++a; }
int x = 7;
x = incr1(x);     // pretty obvious
incr2(x);         // pretty obscure
```

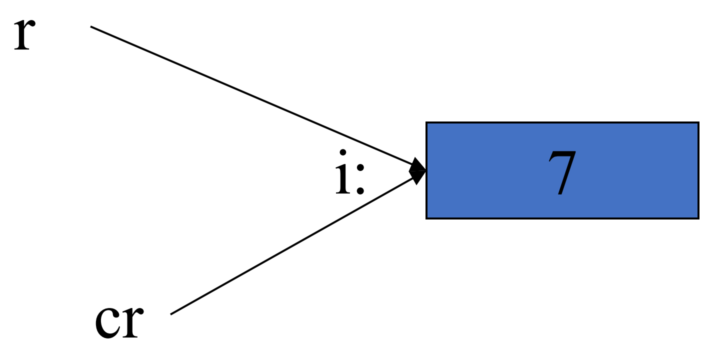
- So why have reference arguments?
  - Occasionally, they are essential
    - *E.g.,* for changing several values
    - For manipulating containers (*e.g.,* vector)
  - **const** reference arguments are very often useful

Call by value/by reference/ by const-reference

```
void f(int a, int& r, const int& cr) { ++a; ++r; ++cr; } // error: cr is const
void g(int a, int& r, const int& cr) { ++a; ++r; int x = cr; ++x; } // ok

int main()
{
    int x = 0;
    int y = 0;
    int z = 0;
    g(x,y,z);          // x==0; y==1; z==0
    g(1,2,3);          // error: reference argument r needs a variable to refer to
    g(1,y,3);          // ok: since cr is const we can pass "a temporary"
}
// const references are very useful for passing large objects
```

References

- "reference" is a general concept
  - Not just for call-by-reference




```
int i = 7;
int& r = i;
r = 9;              // i becomes 9
const int& cr = i;
// cr = 7;          // error: cr refers to const
i = 8;
cout << cr << endl;     // write out the value of i (that's 8)
```

- You can
  - think of a reference as an alternative name for an object
- You can't
  - modify an object through a **const** reference
  - make a reference refer to another object after initialization

For example

- A range-for loop:
  - **for (string s : v) cout << s << "\n";** **//** s is a copy of some v[i]
  - **for (string& s : v) cout << s << "\n";** **//** no copy
  - **for (const string& s : v) cout << s << "\n";** **//** and we don't modify v

Compile-time functions

- You can define functions that *can be* evaluated at compile time: **constexpr** functions

```
constexpr double xscale = 10;               // scaling factors
constexpr double yscale = .8;

constexpr Point scale(Point p) { return {xscale*p.x,yscale*p.y}; };

constexpr Point x = scale({123,456});     // evaluated at compile time

void use(Point p)
{
        constexpr Point x1 = scale(p);      // error: compile-time evaluation
                                            // requested for variable argument
        Point x2 = scale(p);                // OK: run-time evaluation
}
```

Guidance for Passing Variables

- Use call-by-value for very small objects
- Use call-by-const-reference for large objects
- Use call-by-reference only when you have to
- Return a result rather than modify an object through a reference argument

- For example

```
class Image { /* objects are potentially huge */ };
void f(Image i);  … f(my_image);   // oops: this could be s-l-o-o-o-w
void f(Image& i); … f(my_image); // no copy, but f() can modify my_image
void f(const Image&); … f(my_image);  // f() won't mess with my_image
Image make_image(); // most likely fast! ("move semantics" – later)
```

Namespaces

- Consider this code from two programmers Jack and Jill

```
class Glob { /*…*/ };           // in Jack's header file jack.h
class Widget { /*…*/ };         // also in jack.h

class Blob { /*…*/ };           // in Jill's header file  jill.h
class Widget { /*…*/ };         // also in jill.h


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

void my_func(Widget p)          // oops! – error: multiple definitions of
Widget
{
   // …
}
```

Namespaces

- The compiler will not compile multiple definitions; such clashes can occur from multiple headers.
- One way to prevent this problem is with namespaces:

```
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
    // …
}
```

Namespaces

- A namespace is a named scope
- The :: syntax is used to specify which namespace you are using and which (of many possible) objects of the same name you are referring to
- For example, **cout** is in namespace **std**, you could write:

```
std::cout << "Please enter stuff… \n";
```

using Declarations and Directives

- To avoid the tedium of
  - **std::cout << "Please enter stuff… \n";**

  you could write a "using declaration"
  - **using std::cout;** **//** *when I say* ***cout****, I mean* ***std::cout***
  - **cout << "Please enter stuff… \n";** **//** *ok: std::cout*
  - **cin >> x;** **//** *error: cin not in scope*

- or you could write a "using directive"
  - **using namespace std;** **//** *"make all names from namespace* ***std*** *available"*
  - **cout << "Please enter stuff… \n";** **//** *ok: std::cout*
  - **cin >> x;** **//** *ok: std::cin*

- More about header files in chapter 12

Classes

- The idea:
  - A class directly represents a concept in a program
    - If you can think of “it” as a separate entity, it is plausible that it could be a class or an object of a class
    - Examples: vector, matrix, input stream, string, FFT, valve controller, robot arm, device driver, picture on screen, dialog box, graph, window, temperature reading, clock
  - A class is a (user-defined) type that specifies how objects of its type can be created and used
  - In C++ (as in most modern languages), a class is the key building block for large programs
    - And very useful for small ones also
  - The concept was originally introduced in Simula67

Members and member access

- One way of looking at a class;

```
class X {  // this class' name is X
    // data members (they store information)
    // function members (they do things, using the information)
};
```

- Example

```
class X {
public:
    int m;   // data member
    int mf(int v) { int old = m; m=v; return old; }   // function member
};

X var;                          // var is a variable of type X
var.m = 7;                      // access var's data member m
int x = var.mf(9);  // call var's member function mf()
```

Classes

- A class is a user-defined type

```
class X {          // this class' name is X
public:            // public members -- that's the interface to users
                   //          (accessible by all)
   // functions
   // types
   // data (often best kept private)
private:           // private members -- that's the implementation details
                   //          (accessible by members of this class only)
   // functions
   // types
   // data
};
```

Struct and class

- Class members are private by default:

```
class X {
        int mf();
        // …
};
```

- Means

```
class X {
private:
        int mf();
        // …
};
```

- So

```
X x;                // variable x of type X
int y = x.mf();     // error: mf is private (i.e., inaccessible)
```

Struct and class

- A struct is a class where members are public by default:

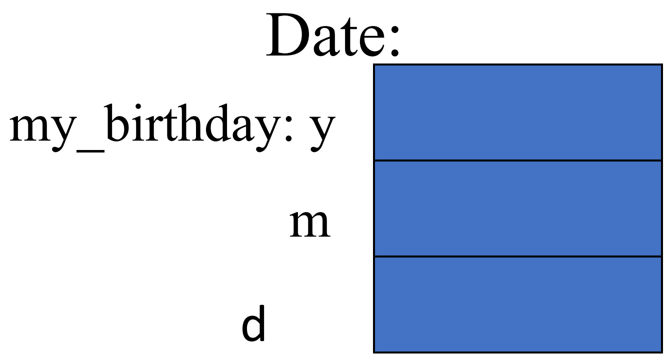
```
struct X {
        int m;
        // …
};
```

- Means

```
class X {
public:
        int m;
        // …
};
```

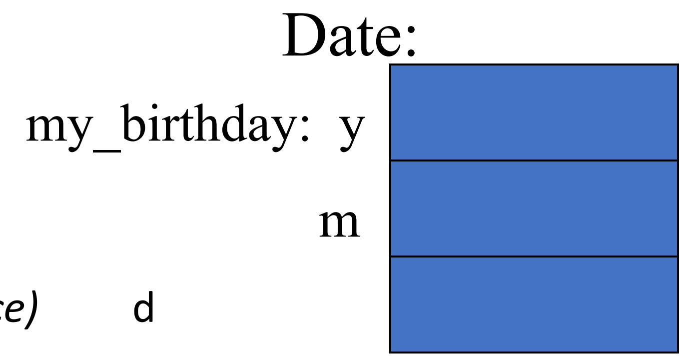
- **struct**s are primarily used for data structures where the members can take any value

Structs



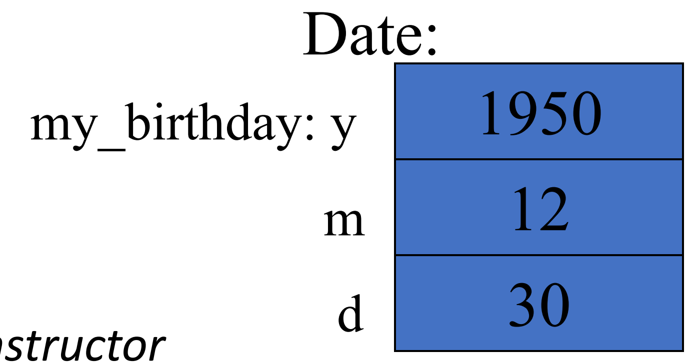

```
// simplest Date (just data)
struct Date {
   int y,m,d;          // year, month, day
};

Date my_birthday;          // a Date variable (object)

my_birthday.y = 12;
my_birthday.m = 30;
my_birthday.d = 1950;      // oops! (no day 1950 in month 30)
                           // later in the program, we'll have a problem
```

Structs



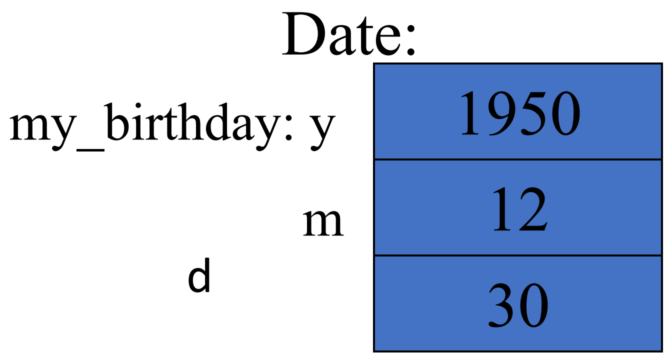

```
// simple Date (with a few helper functions for convenience)
struct Date {
    int y,m,d;            // year, month, day
};

Date my_birthday;   // a Date variable (object)


// helper functions:

void init_day(Date& dd, int y, int m, int d); // check for valid date and initialize
                                              // Note: this y, m, and d are local

void add_day(Date& dd, int n);            // increase the Date by n days
// …

init_day(my_birthday, 12, 30, 1950); // run time error: no day 1950 in month 30
```

Structs




```
// simple Date
//                      guarantee initialization with constructor
//           provide some notational convenience
struct Date {
    int y,m,d;                          // year, month, day
    Date(int y, int m, int d);  // constructor: check for valid date and initialize
    void add_day(int n);            // increase the Date by n days
};

// …
Date my_birthday;                       // error: my_birthday not initialized
Date my_birthday {12, 30, 1950};   // oops! Runtime error
Date my_day {1950, 12, 30};          // ok
my_day.add_day(2);                     // January 1, 1951
my_day.m = 14;                         // ouch! (now my_day is a bad date)
```

Classes

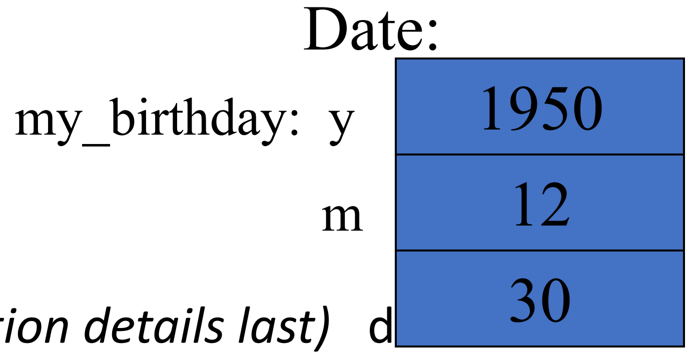



```
// simple Date (control access)
class Date {
    int y,m,d;          // year, month, day
public:
    Date(int y, int m, int d);  // constructor: check for valid date and initialize

    // access functions:
    void add_day(int n);          // increase the Date by n days
    int month() { return m; }
    int day() { return d; }
    int year() { return y; }
};

// …
Date my_birthday {1950, 12, 30};            // ok
cout << my_birthday.month() << endl;        // we can read
my_birthday.m = 14;                         // error: Date::m is private
```

Classes

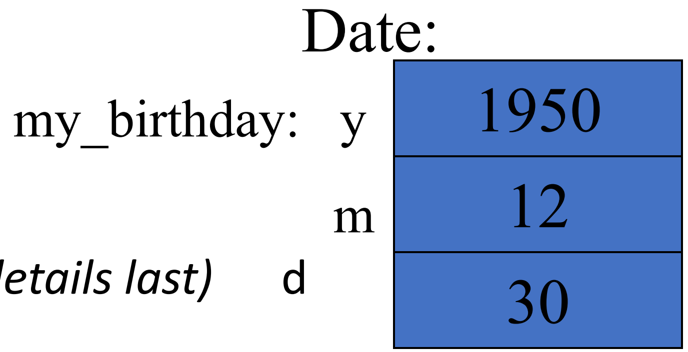
- The notion of a “valid Date” is an important special case of the idea of a valid value
- We try to design our types so that values are guaranteed to be valid
  - Or we have to check for validity all the time
- A rule for what constitutes a valid value is called an “invariant”
  - The invariant for Date (“a Date must represent a date in the past, present, or future”) is unusually hard to state precisely
    - Remember February 28, leap years, etc.
- If we can’t think of a good invariant, we are probably dealing with plain data
  - If so, use a struct
  - Try hard to think of good invariants for your classes
    - that saves you from poor buggy code

Classes




```
// simple Date (some people prefer implementation details last)
class Date {
public:
    Date(int yy, int mm, int dd);    // constructor: check for valid date and
                                     // initialize
    void add_day(int n);             // increase the Date by n days
    int month();
    // …
private:
    int y,m,d;        // year, month, day
};

Date::Date(int yy, int mm, int dd)            // definition; note :: "member of"
    :y{yy}, m{mm}, d{dd} { /* … */ };// note: member initializers

void Date::add_day(int n) { /* … */ };        // definition
```

Classes




```
// simple Date (some people prefer implementation details last)
class Date {
public:
    Date(int yy, int mm, int dd);  // constructor: check for valid date and
                                   // initialize
    void add_day(int n);           // increase the Date by n days
    int month();
    // ...
private:
    int y,m,d;          // year, month, day
};

int month() { return m; }     // error: forgot Date::
                              // this month() will be seen as a global function
                              // not the member function, so can't access members

int Date::season() { /* ... */ }   // error: no member called season
```

Classes

```
// simple Date (what can we do in case of an invalid date?)
class Date {
public:
    class Invalid { };              // to be used as exception
    Date(int y, int m, int d);      // check for valid date and initialize
    // …
private:
    int y,m,d;                                          // year, month, day
    bool is_valid(int y, int m, int d);      // is (y,m,d) a valid date?
};

Date:: Date(int yy, int mm, int dd)
    : y{yy}, m{mm}, d{dd}                    // initialize data members
{
    if (!is_valid (y,m,d)) throw Invalid();             // check for validity
}
```

Classes

- Why bother with the public/private distinction?
- Why not make everything public?
  - To provide a clean interface
    - Data and messy functions can be made private
  - To maintain an invariant
    - Only a fixed set of functions can access the data
  - To ease debugging
    - Only a fixed set of functions can access the data
    - (known as the "round up the usual suspects" technique)
  - To allow a change of representation
    - You need only to change a fixed set of functions
    - You don't really know who is using a public member

Enumerations

- An **enum** (enumeration) is a simple user-defined type, specifying its set of values (its enumerators)

- For example:

```
enum class Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

Month m = feb;
m = 7;                  // error: can't assign int to Month
int n = m;              // error: we can't get the numeric value of a Month
Month mm = Month(7);    // convert int to Month (unchecked)
```

“Plain” Enumerations

- Simple list of constants:

```
enum { red, green };   // a "plain" enum { } doesn't define a scope
int a = red;           // red is available here
enum { red, blue, purple };   // error: red defined twice
```

- Type with a list of named constants

```
enum Color { red, green, blue, /* … */  };
enum Month { jan, feb, mar, /* … */ };

Month m1 = jan;
Month m2 = red;     // error: red isn't a Month
Month m3 = 7;       // error: 7 isn't a Month
int i = m1;         // ok: an enumerator is converted to its value, i==0
```

Class Enumerations

- Type with a list of typed named constants

```
enum class Color { red, green, blue, /* … */  };
enum class Month { jan, feb, mar, /* … */ };
enum class Traffic_light { green, yellow, red }; // OK: scoped enumerators

Month m1 = jan;                    // error: jan not in scope
Month m1 = Month::jan;             // OK
Month m2 = Month::red;             // error: red isn't a Month
Month m3 = 7;                      // error: 7 isn't a Month
Color c1 = Color::red; // OK
Color c2 = Traffic_light::red;     // error
int i = m1;                        // error: an enumerator is not converted to int
```

Enumerations – Values

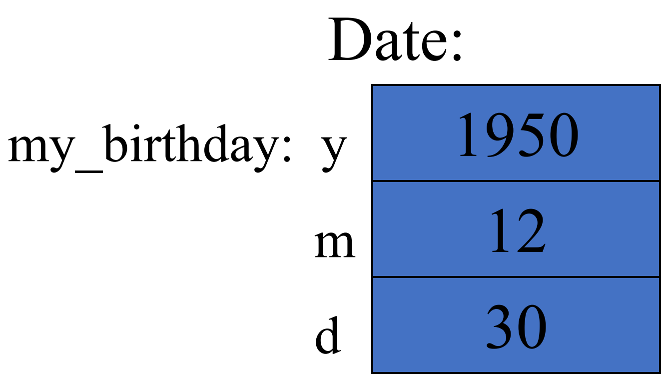
- By default

```
// the first enumerator  has the value  0,
// the next enumerator has the value  "one plus the value of the
// enumerator before it"
enum { horse, pig, chicken };                // horse==0, pig==1, chicken==2
```

- You can control numbering

```
enum { jan=1, feb, march /* … */ };       // feb==2, march==3
enum stream_state { good=1, fail=2, bad=4, eof=8 };
int flags = fail+eof;                     // flags==10
stream_state s = flags;                   // error: can't assign an int to a
   stream_state
stream_state s2 = stream_state(flags);  // explicit conversion (be careful!)
```

Classes




```
// 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
```

Const

```
class Date {
public:
    // …
    int day() const { return d; }          // const member: can't modify
    void add_day(int n);                   // non-const member: can modify
    // …
};

Date d {2000, Month::jan, 20};
const Date cd {2001, Month::feb, 21};

cout << d.day() << " – " << cd.day() << endl;   // ok
d.add_day(1);       // ok
cd.add_day(1);      // error: cd is a const
```

Const

```
Date d {2004, Month::jan, 7};               // a variable
const Date d2 {2004, Month::feb, 28};       // a constant
d2 = d;              // error: d2 is const
d2.add(1);           // error d2 is const
d = d2;              // fine
d.add(1);            // fine

d2.f();    // should work if and only if f() doesn't modify d2
           // how do we achieve that? (say that's what we want, of course)
```

Const member functions

```
// Distinguish between functions that can modify (mutate) objects
// and those that cannot ("const member functions")
class Date {
public:
    // …
    int day() const;  // get (a copy of) the day
    // …
    void add_day(int n);        // move the date n days forward
    // …
};

const Date dx {2008, Month::nov, 4};
int d = dx.day();    // fine
dx.add_day(4);       // error: can't modify constant (immutable) date
```

Classes

- What makes a good interface?
  - Minimal
    - As small as possible
  - Complete
    - And no smaller
  - Type safe
    - Beware of confusing argument orders
    - Beware of over-general types (e.g., int to represent a month)
  - Const correct

Classes

- Essential operations
  - Default constructor (defaults to: nothing)
    - No default if any other constructor is declared
  - Copy constructor (defaults to: copy the member)
  - Copy assignment (defaults to: copy the members)
  - Destructor (defaults to: nothing)
- For example

```
Date d;        // error: no default constructor
Date d2 = d;   // ok: copy initialized (copy the elements)
d = d2;        // ok copy assignment (copy the elements)
```

Interfaces and “helper functions”

- Keep a class interface (the set of public functions) minimal
  - Simplifies understanding
  - Simplifies debugging
  - Simplifies maintenance
- When we keep the class interface simple and minimal, we need extra “helper functions” outside the class (non-member functions)
  - E.g. == (equality) , != (inequality)
  - **next_weekday(), next_Sunday()**

Helper functions

```
Date next_Sunday(const Date& d)
{
    // access d using d.day(), d.month(), and d.year()
    // make new Date to return
}

Date next_weekday(const Date& d) { /* … */ }

bool operator==(const Date& a, const Date& b)
{
    return a.year()==b.year()
            && a.month()==b.month()
            && a.day()==b.day();
}

bool operator!=(const Date& a, const Date& b) { return !(a==b); }
```

Operator overloading

- You can define almost all C++ operators for a class or enumeration operands
  - That's often called "operator overloading"

```
enum class Month {
    jan=1, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
};

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

Month m = Month::nov;
++m;     // m becomes dec
++m;     // m becomes jan
```

Operator overloading

- You can define only existing operators
  - *E.g.*, **+ - * / % [] () ^ ! & < <= > >=**
- You can define operators only with their conventional number of operands
  - *E.g.*, no unary **<=** (less than or equal) and no binary **!** (not)
- An overloaded operator must have at least one user-defined type as operand
  - **int operator+(int,int);** **//** error: you can't overload built-in +
  - **Vector operator+(const Vector&, const Vector &);** **//** ok
- Advice (not language rule):
  - Overload operators only with their conventional meaning
  - **+** should be addition, ***** be multiplication, **[]** be access, **()** be call, etc.
- Advice (not language rule):
  - Don't overload unless you really have to

Ideals

- Our ideal of program design is to represent the concepts of the application domain directly in code.
  - If you understand the application domain, you understand the code, and *vice versa*. For example:
    - **Window** – a window as presented by the operating system
    - **Line** – a line as you see it on the screen
    - **Point** – a coordinate point
    - **Color** – as you see it on the screen
    - **Shape** – what's common for all shapes in our Graph/GUI view of the world
- The last example, **Shape**, is different from the rest in that it is a generalization.
  - You can't make an object that's "just a Shape"

Logically identical operations have the same name

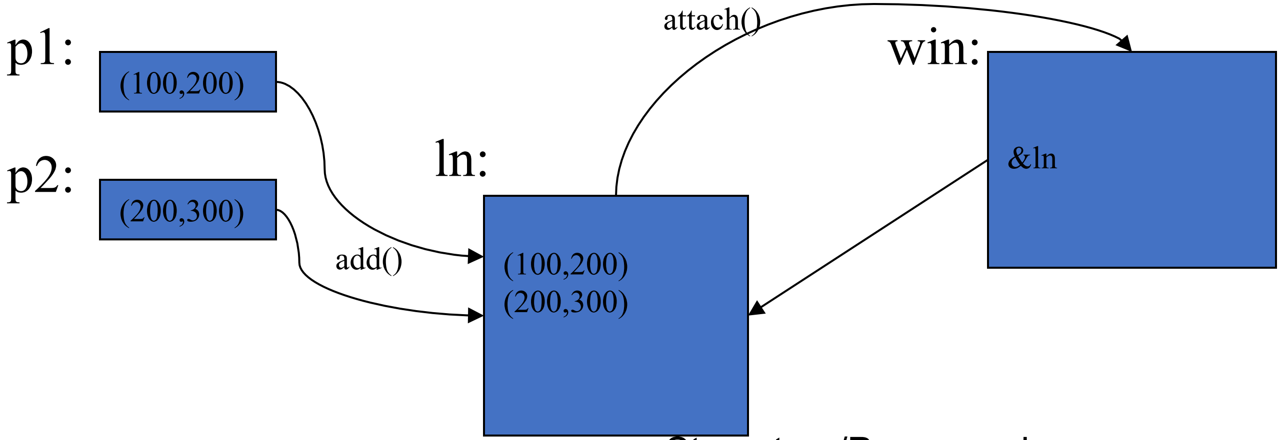
- For every class,
  - **draw_lines()** does the drawing
  - **move(dx,dy)** does the moving
  - **s.add(x)** adds some **x** (*e.g.*, a point) to a shape **s**.
- For every property **x** of a Shape,
  - **x()** gives its current value and
  - **set_x()** gives it a new value
  - e.g.,

```
Color c = s.color();
s.set_color(Color::blue);
```

Logically different operations have different names

```
Lines ln;
Point p1(100,200);
Point p2(200,300);
ln.add(p1,p2);          // add points to ln (make copies)
win.attach(ln);         // attach ln to window
```

- Why not **win.add(ln)**?
  - **add()** copies information; **attach()** just creates a reference
  - we can change a displayed object after attaching it, but not after adding it



Expose uniformly

- Data should be private
  - Data hiding – so it will not be changed inadvertently
  - Use **private** data, and pairs of public access functions to get and set the data

```
c.set_radius(12);              // set radius to 12
c.set_radius(c.radius()*2);    // double the radius (fine)
c.set_radius(-9);              // set_radius() could check for negative,
                               // but doesn't yet
double r = c.radius();         // returns value of radius
c.radius = -9;                 // error: radius is a function (good!)
c.r = -9;                      // error: radius is private (good!)
```

- Our functions can be private or public
  - Public for interface
  - Private for functions used only internally to a class

What does “private” buy us?

- We can change our implementation after release
- We don’t expose FLTK types used in representation to our users
  - We could replace FLTK with another library without affecting user code
- We could provide checking in access functions
  - But we haven’t done so systematically (later?)
- Functional interfaces can be nicer to read and use
  - E.g., **s.add(x)** rather than **s.points.push_back(x)**
- We enforce immutability of shape
  - Only color and style change; not the relative position of points
  - **const** member functions
- The value of this “encapsulation” varies with application domains
  - Is often most valuable
  - Is the ideal
    - i.e., hide representation unless you have a good reason not to

"Regular" interfaces

```
Line ln(Point(100,200),Point(300,400));
Mark m(Point(100,200), 'x');          // display a single point as an 'x'
Circle c(Point(200,200),250);

// Alternative (not supported):
Line ln2(x1, y1, x2, y2);             // from (x1,y1) to (x2,y2)

// How about? (not supported):
Rectangle s1(Point(100,200),200,300);         // width==200 height==300
Rectangle s2(Point(100,200),Point(200,300));  // width==100 height==100

Rectangle s3(100,200,200,300);// is 200,300 a point  or a width plus a height?
```

A library

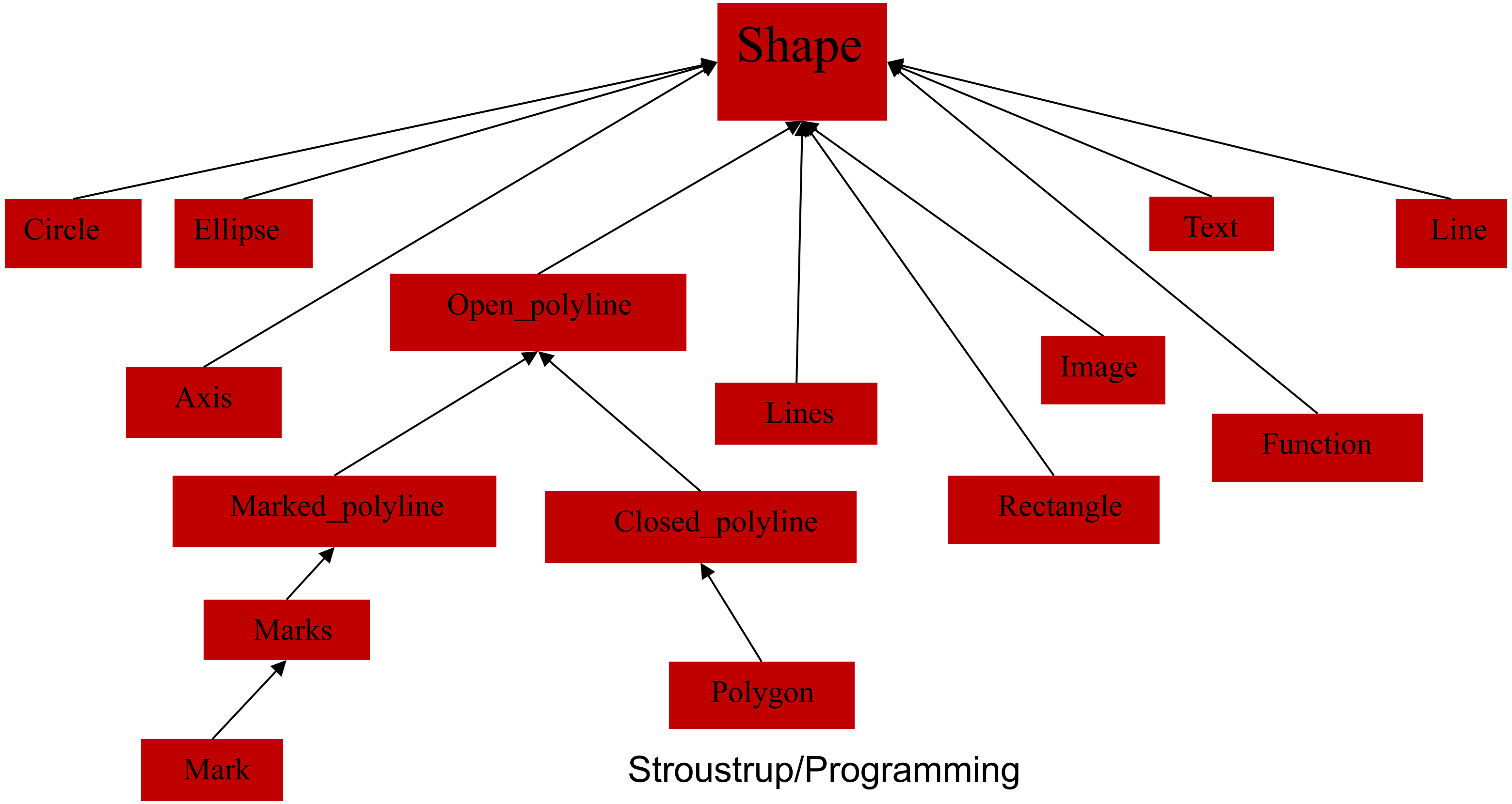
- A collection of classes and functions meant to be used together
  - As building blocks for applications
  - To build more such “building blocks”

- A good library models some aspect of a domain
  - It doesn’t try to do everything
  - Our library aims at simplicity and small size for graphing data and for very simple GUI

- We can’t define each library class and function in isolation
  - A good library exhibits a uniform style (“regularity”)

Class Shape

- All our shapes are "based on" the Shape class
  - E.g., a **Polygon** is a kind of **Shape**



Class Shape – is abstract

- You can't make a "plain" Shape

```
protected:
    Shape();          // protected to make class Shape abstract
```

For example

```
    Shape ss;         // error: cannot construct Shape
```

  - Protected means "can only be used from this class or from a derived class"

- Instead, we use Shape as a base class

```
struct Circle : Shape {      // "a Circle is a Shape"
    // …
};
```

Class Shape

- **Shape** ties our graphics objects to “the screen”
  - **Window** “knows about” **Shape**s
  - All our graphics objects are kinds of **Shape**s
- **Shape** is the class that deals with color and style
  - It has **Color** and **Line_style** members
- **Shape** can hold **Point**s
- **Shape** has a basic notion of how to draw lines
  - It just connects its **Point**s

Class Shape

- Shape deals with color and style
  - It keeps its data private and provides access functions

```
    void set_color(Color col);
    Color color() const;
    void set_style(Line_style sty);
    Line_style style() const;
    // …
private:
    // …
    Color line_color;
    Line_style ls;
```

Class Shape

- **Shape** stores **Point**s
  - It keeps its data private and provides access functions

```
    Point point(int i) const;  // read-only access to points
    int number_of_points() const;
    // …
protected:
    void add(Point p);              // add p to points
    // …
private:
    vector<Point> points;     // not used by all shapes
```

Class Shape

- **Shape** itself can access points directly:

```
void Shape::draw_lines() const          // draw connecting lines
{
   if (color().visible() && 1<points.size())
      for (int i=1; i<points.size(); ++i)
               fl_line(points[i-1].x,points[i-1].y,points[i].x,points[i].y);
}
```

- Others (incl. derived classes) use **point()** and **number_of_points()**
  - why?

```
void Lines::draw_lines() const  // draw a line for each pair of points
{
   for (int i=1; i<number_of_points(); i+=2)
               fl_line(point(i-1).x, point(i-1).y, point(i).x, point(i).y);
}
```

Class Shape (basic idea of drawing)

```
void Shape::draw() const
    // The real heart of class Shape (and of our graphics interface system)
    // called by Window (only)
{
    // … save old color and style …
    // … set color and style for this shape…

    // … draw what is specific for this particular shape …
    // … Note: this varies dramatically depending on the type of shape …
    // … e.g. Text, Circle, Closed_polyline

    // … reset the color and style to their old values …
}
```

Class Shape (implementation of drawing)

```
void Shape::draw() const
    // The real heart of class Shape (and of our graphics interface system)
    // 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());

    draw_lines();          // call the appropriate 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
}
```

Note!

Class shape

- In class **Shape**

  virtual void draw_lines() const; **//** *draw the appropriate lines*

- In class **Circle**

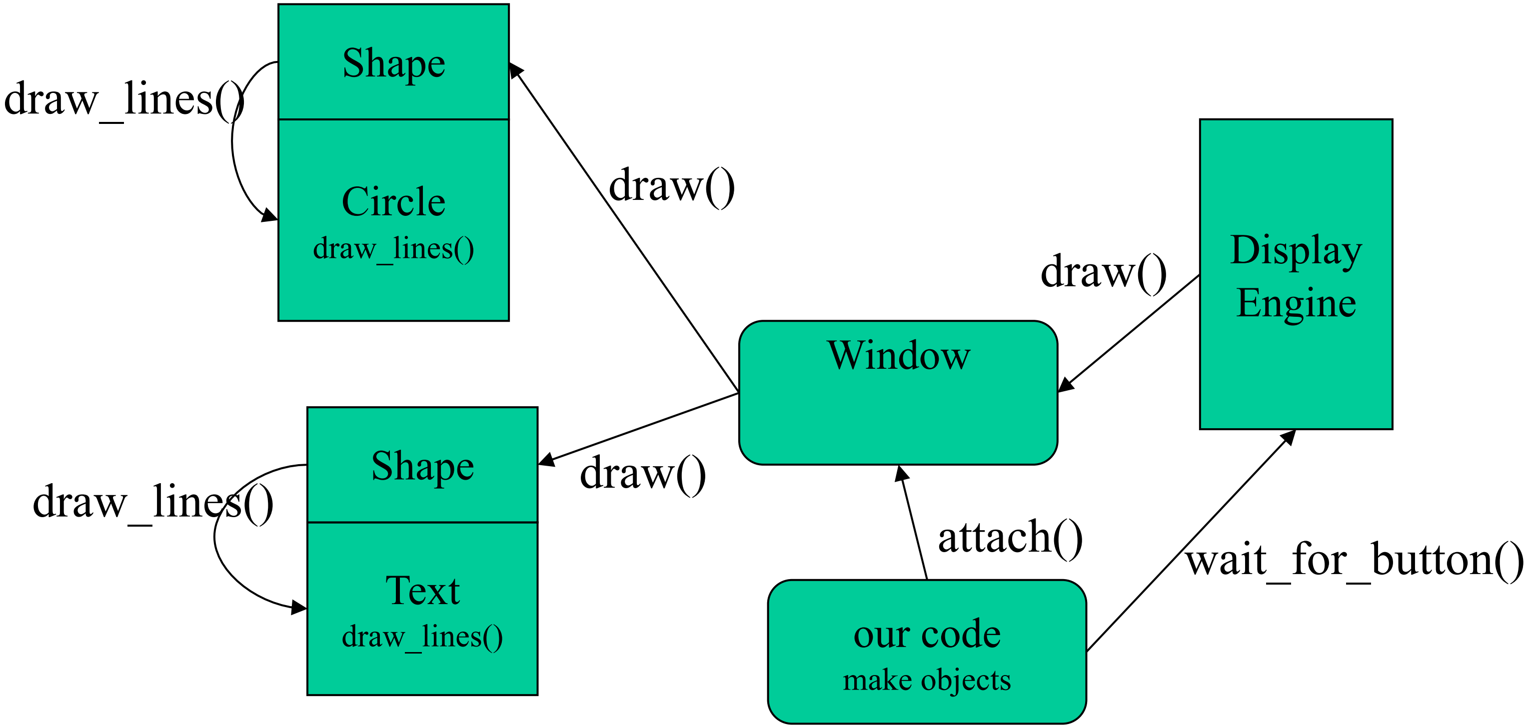
  void draw_lines() const { /* *draw the Circle* ***/ }**

- In class **Text**
  - **void draw_lines() const { /*** *draw the Text* ***/ }**

- **Circle**, **Text**, and other classes
  - "Derive from" **Shape**
  - May "override" **draw_lines()**

```
class Shape {          // deals with color and style, and holds a sequence of lines
public:
    void draw() const;                       // deal with color and call draw_lines()
    virtual void move(int dx, int dy);       // move the shape +=dx and +=dy

    void set_color(Color col);    // color access
    int color() const;
    // … style and fill_color access functions …

    Point point(int i) const;     // (read-only) access to points
    int number_of_points() const;
protected:
    Shape();                                 // protected to make class Shape abstract
    void add(Point p);            // add p to points
    virtual void draw_lines() const;         // simply draw the appropriate lines
private:
    vector<Point> points;                    // not used by all shapes
    Color  lcolor;                           // line color
    Line_style  ls;                          // line style
    Color  fcolor;                           // fill color

    // … prevent copying …
};
```

Display model completed



Language mechanisms

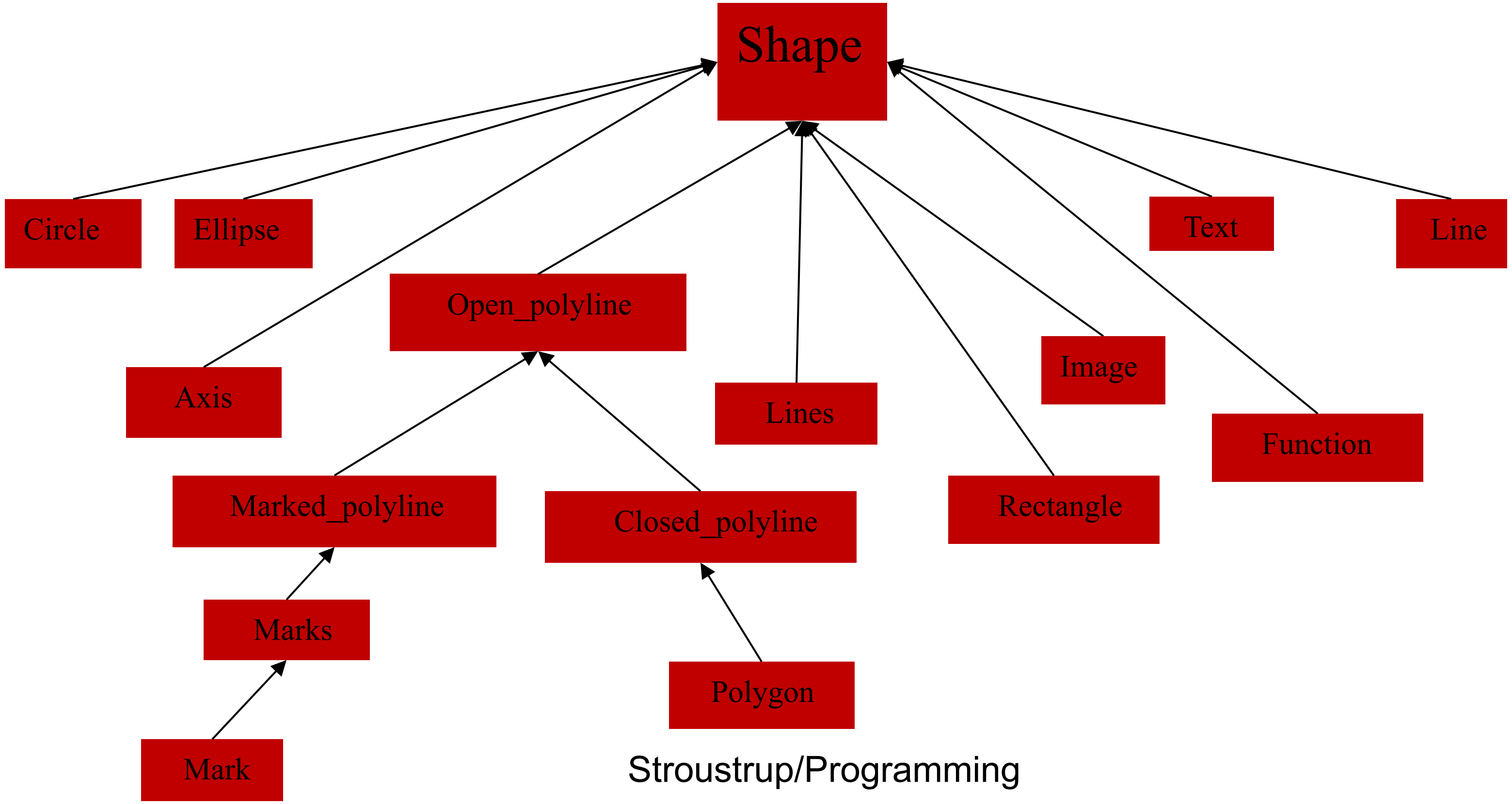
- Most popular definition of object-oriented programming:

  OOP == inheritance + polymorphism + encapsulation

- Base and derived classes // *inheritance*
  - **struct Circle : Shape { … };**
  - Also called "inheritance"
- Virtual functions // *polymorphism*
  - **virtual void draw_lines() const;**
  - Also called "run-time polymorphism" or "dynamic dispatch"
- Private and protected // *encapsulation*
  - **protected: Shape();**
  - **private: vector<Point> points;**

A simple class hierarchy

- We chose to use a simple (and mostly shallow) class hierarchy
  - Based on Shape



Object layout

- The data members of a derived class are simply added at the end of its base class (a Circle is a Shape with a radius)

Shape:

points
line_color
ls

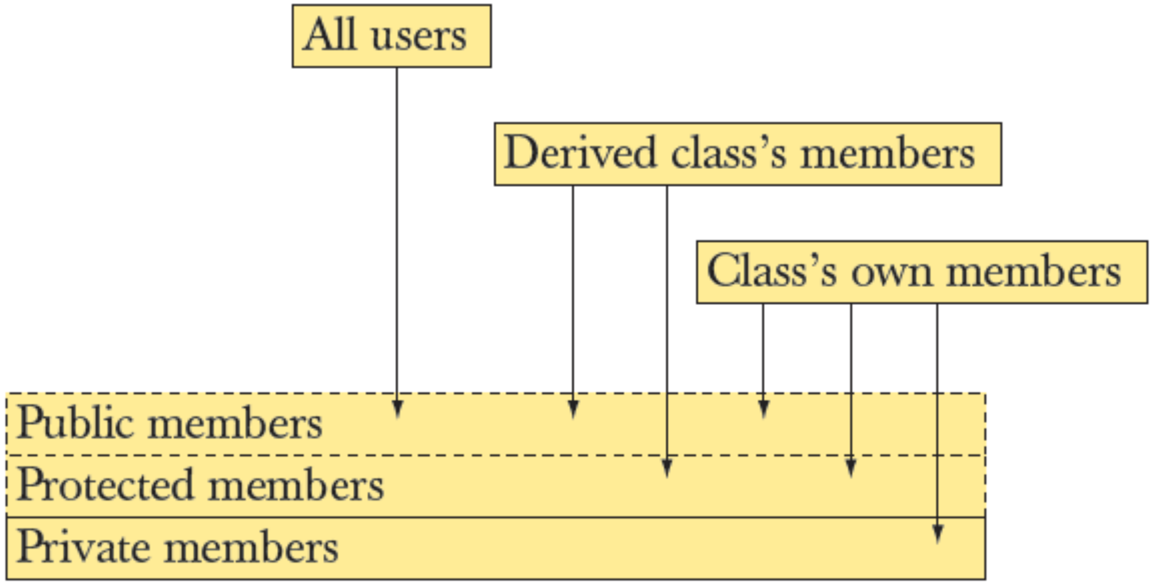
Circle:

points
line_color
ls
----
r

Benefits of inheritance

- Interface inheritance
  - A function expecting a shape (a **Shape&**) can accept any object of a class derived from Shape.
  - Simplifies use
    - sometimes dramatically
  - We can add classes derived from Shape to a program without rewriting user code
    - Adding without touching old code is one of the "holy grails" of programming
- Implementation inheritance
  - Simplifies implementation of derived classes
    - Common functionality can be provided in one place
    - Changes can be done in one place and have universal effect
      - Another "holy grail"

Access model



- A member (data, function, or type member) or a base can be
  - Private, protected, or public

Pure virtual functions

- Often, a function in an interface can't be implemented
  - E.g. the data needed is "hidden" in the derived class
  - We must ensure that a derived class implements that function
  - Make it a "pure virtual function" (**=0**)
- This is how we define truly abstract interfaces ("pure interfaces")

```
struct Engine {            // interface to electric motors
    // no data
    // (usually) no constructor
    virtual double increase(int i) =0;     // must be defined in a derived class
    // …
    virtual ~Engine();    // (usually) a virtual destructor
};
Engine eee;  // error: Collection is an abstract class
```

Pure virtual functions

- A pure interface can then be used as a base class
  - Constructors and destructors will be describe d in detail in chapters 17-19

```
Class M123 : public Engine {    // engine model M123
   // representation
public:
   M123();     // construtor:  initialization, acquire resources
   double increase(int i) { /* … */ }        // overrides Engine ::increase
   // …
   ~M123();   // destructor: cleanup, release resources
};

M123 window3_control;          // OK
```

Technicality: Copying

- If you don't know how to copy an object, prevent copying
  - Abstract classes typically should not be copied

```
class Shape {
    // …
    Shape(const Shape&) = delete;                  // don't "copy construct"
    Shape& operator=(const Shape&) = delete;       // don't "copy assign"
};

void f(Shape& a)
{
    Shape s2 = a;        // error: no Shape "copy constructor" (it's deleted)
    a = s2;              // error: no Shape "copy assignment" (it's deleted)
}
```

Prevent copying C++98 style

- If you don’t know how to copy an object, prevent copying
  - Abstract classes typically should not be copied

```
class Shape {
    // …
private:
    Shape(const Shape&);                    // don't "copy construct"
    Shape& operator=(const Shape&);         // don't "copy assign"
};

void f(Shape& a)
{
    Shape s2 = a;       // error: no Shape "copy constructor" (it's private)
    a = s2;             // error: no Shape "copy assignment" (it's private)
}
```

Technicality: Overriding

- To override a virtual function, you need
  - A virtual function
  - Exactly the same name
  - Exactly the same type

```
struct B {
    void f1();   // not virtual
    virtual void f2(char);
    virtual void f3(char) const;
    virtual void f4(int);
};
```

```
struct D : B {
    void f1();          // doesn't override
    void f2(int);       // doesn't override
    void f3(char);      // doesn't override
    void f4(int);       // overrides
};
```

Technicality: Overriding

- To override a virtual function, you need
  - A virtual function
  - Exactly the same name
  - Exactly the same type

```
struct B {
   void f1();   // not virtual
   virtual void f2(char);
   virtual void f3(char) const;
   virtual void f4(int);
};
```

```
struct D : B {
   void f1() override;           // error
   void f2(int) override;        // error
   void f3(char) override;       // error
   void f4(int) override;        // OK
};
```

Technicality: Overriding

- To invoke a virtual function, you need
  - A reference, or
  - A pointer

```
D d1;
B& bref = d1; // d1 is a D, and a D is a B, so d1 is a B
bref.f4(2);                    // calls D::f4(2) on d1 since bref names a D

// pointers are in chapter 17
B *bptr = &d1;                 // d1 is a D, and a D is a B, so d1 is a B
bptr->f4(2);                   // calls D::f4(2) on d1 since bptr points to a D
```

Overview

- Common tasks and ideals
- Generic programming
- Containers, algorithms, and iterators
- The simplest algorithm: find()
- Parameterization of algorithms
  - find_if() and function objects
- Sequence containers
  - vector and list
- Associative containers
  - map, set
- Standard algorithms
  - copy, sort, …
  - Input iterators and output iterators
- List of useful facilities
  - Headers, algorithms, containers, function objects

Common tasks

- Collect data into containers
- Organize data
  - For printing
  - For fast access
- Retrieve data items
  - By index (e.g., get the **N**th element)
  - By value (e.g., get the first element with the value **"Chocolate"**)
  - By properties (e.g., get the first elements where “**age<64**”)
- Add data
- Remove data
- Sorting and searching
- Simple numeric operations

Ideals

We'd like to write common programming tasks so that we don't have to re-do the work each time we find a new way of storing the data or a slightly different way of interpreting the data

- Finding a value in a **vector** isn't all that different from finding a value in a **list** or an array
- Looking for a **string** ignoring case isn't all that different from looking at a **string** not ignoring case
- Graphing experimental data with exact values isn't all that different from graphing data with rounded values
- Copying a file isn't all that different from copying a vector

Ideals (continued)

- Code that's
  - Easy to read
  - Easy to modify
  - Regular
  - Short
  - Fast
- Uniform access to data
  - Independently of how it is stored
  - Independently of its type
- …

Ideals (continued)

- …
- Type-safe access to data
- Easy traversal of data
- Compact storage of data
- Fast
  - Retrieval of data
  - Addition of data
  - Deletion of data
- Standard versions of the most common algorithms
  - Copy, find, search, sort, sum, …

Generic programming

- Generalize algorithms
  - Sometimes called "lifting an algorithm"
- The aim (for the end user) is
  - Increased correctness
    - Through better specification
  - Greater range of uses
    - Possibilities for re-use
  - Better performance
    - Through wider use of tuned libraries
    - Unnecessarily slow code will eventually be thrown away
- Go from the concrete to the more abstract
  - The other way most often leads to bloat

The STL

- Part of the ISO C++ Standard Library
- Mostly non-numerical
  - Only 4 standard algorithms specifically do computation
    - Accumulate, inner_product, partial_sum, adjacent_difference
  - Handles textual data as well as numeric data
    - E.g. string
  - Deals with organization of code and data
    - Built-in types, user-defined types, and data structures
- Optimizing disk access was among its original uses
  - Performance was always a key concern

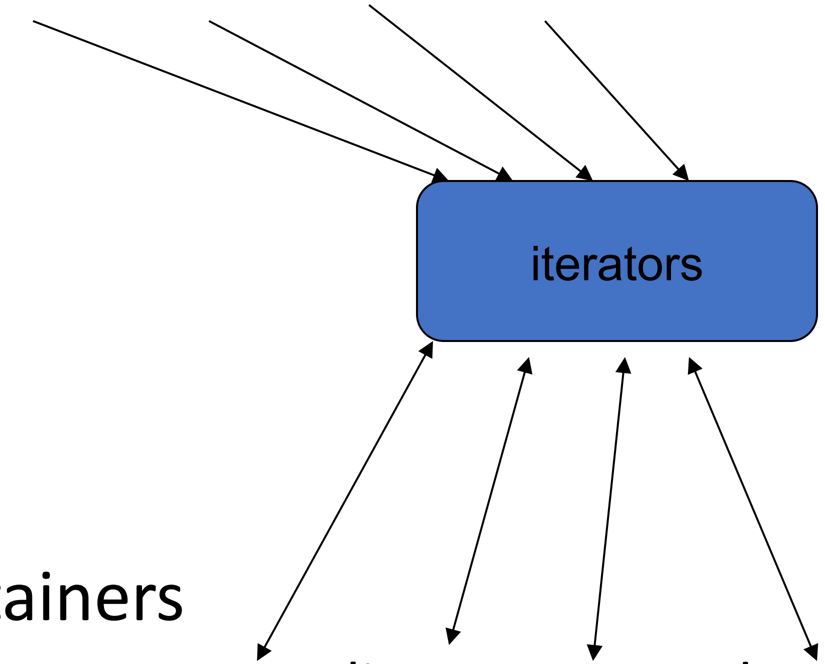
The STL

- Designed by Alex Stepanov
- General aim: The most general, most efficient, most flexible representation of concepts (ideas, algorithms)
  - Represent separate concepts separately in code
  - Combine concepts freely wherever meaningful
- General aim to make programming "like math"
  - or even "Good programming *is* math"
  - works for integers, for floating-point numbers, for polynomials, for …

Basic model

- Algorithms

  sort, find, search, copy, …



- Containers

  vector, list, map, unordered_map, …

- **Separation of concerns**
  - Algorithms manipulate data, but don't know about containers
  - Containers store data, but don't know about algorithms
  - Algorithms and containers interact through iterators
    - Each container has its own iterator types

The STL

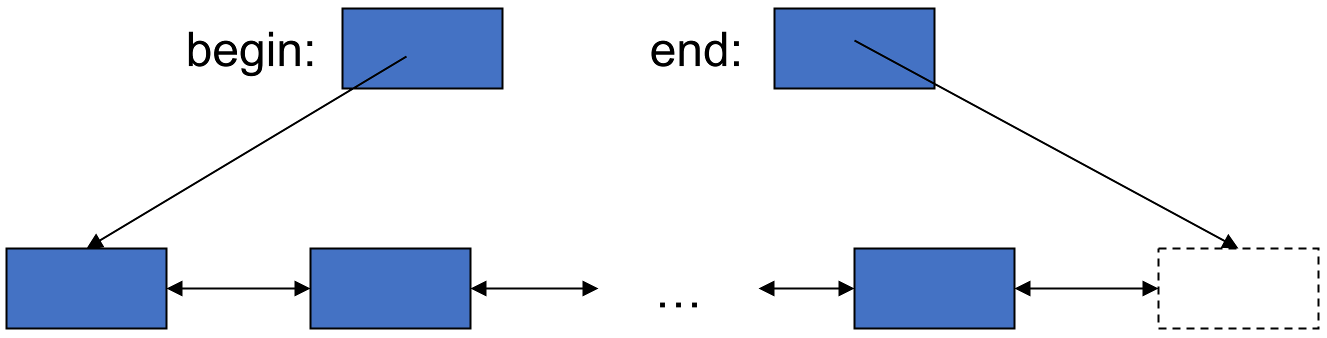
- An ISO C++ standard framework of about 10 containers and about 60 algorithms connected by iterators
  - Other organizations provide more containers and algorithms in the style of the STL
    - Boost.org, Microsoft, … I updated this! -Adriana
- Probably the currently best known and most widely used example of generic programming

The STL

- If you know the basic concepts and a few examples you can use the rest
- Documentation I updated this! -Adriana
  - SGI
    - http://www.martinbroadhurst.com/stl/ (recommended because of clarity)
  - Cpp reference
    - https://en.cppreference.com/w/cpp

Basic model

- A pair of iterators defines a sequence
  - The beginning (points to the first element – if any)
  - The end (points to the one-beyond-the-last element)



- **An iterator is a type that supports the "iterator operations"**
  - **++ Go to next element**
  - *** Get value**
  - **== Does this iterator point to the same element as that iterator?**
- **Some iterators support more operations (e.g. --, +, and [ ])**

Containers

(hold sequences in difference ways)

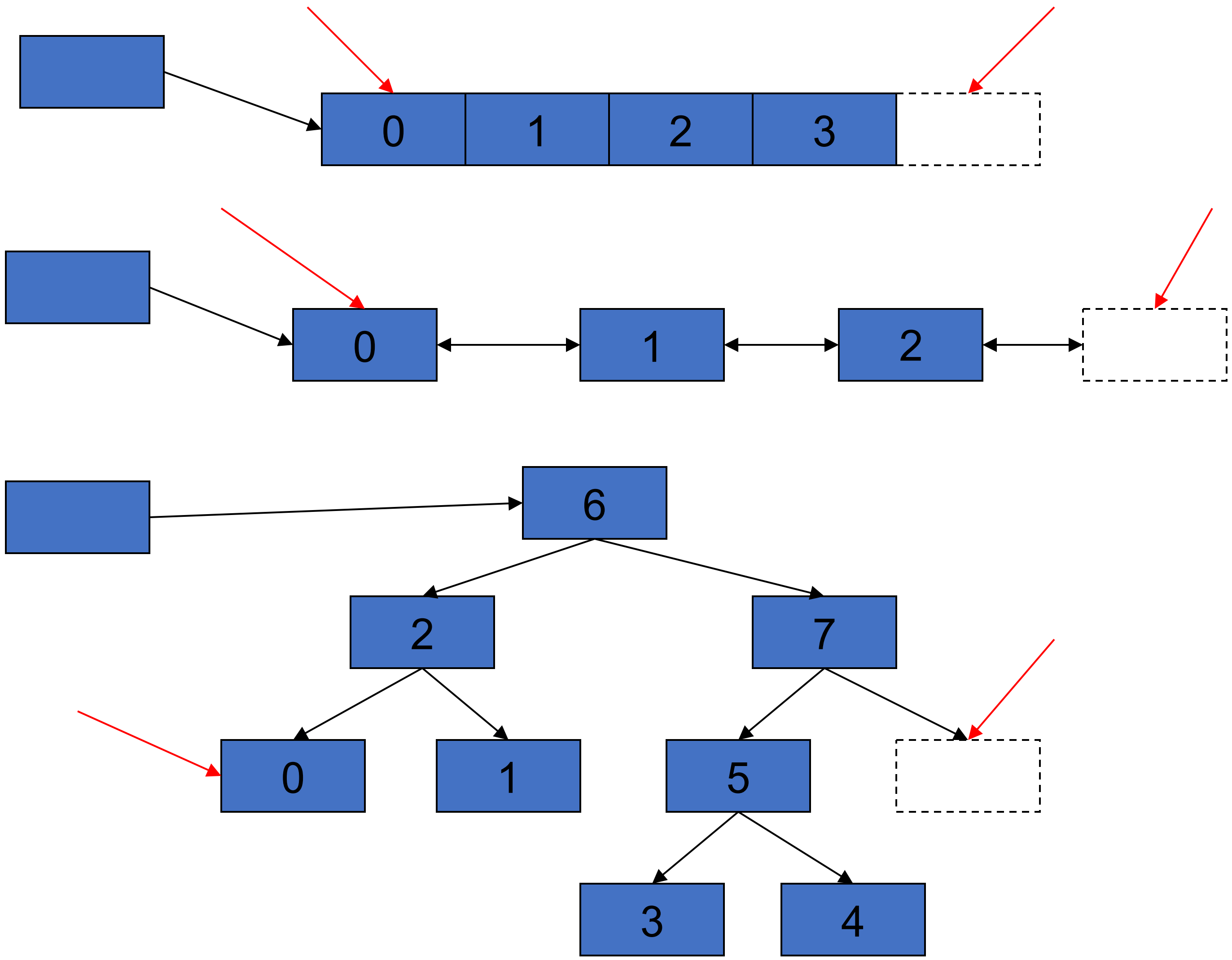
- **vector**

- **list**

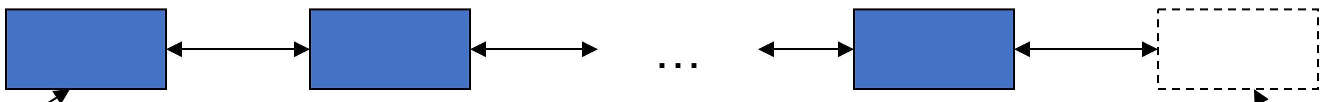
(doubly linked)

- **set**

(a kind of tree)



The simplest algorithm: **find()**




```
// Find the first element that equals a value

template<class In, class T>
In find(In first, In last, const T& val)
{
    while (first!=last && *first != val) ++first;
    return first;
}

void f(vector<int>& v, int x)     // find an int in a vector
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}
```

We can ignore ("abstract away") the differences between containers

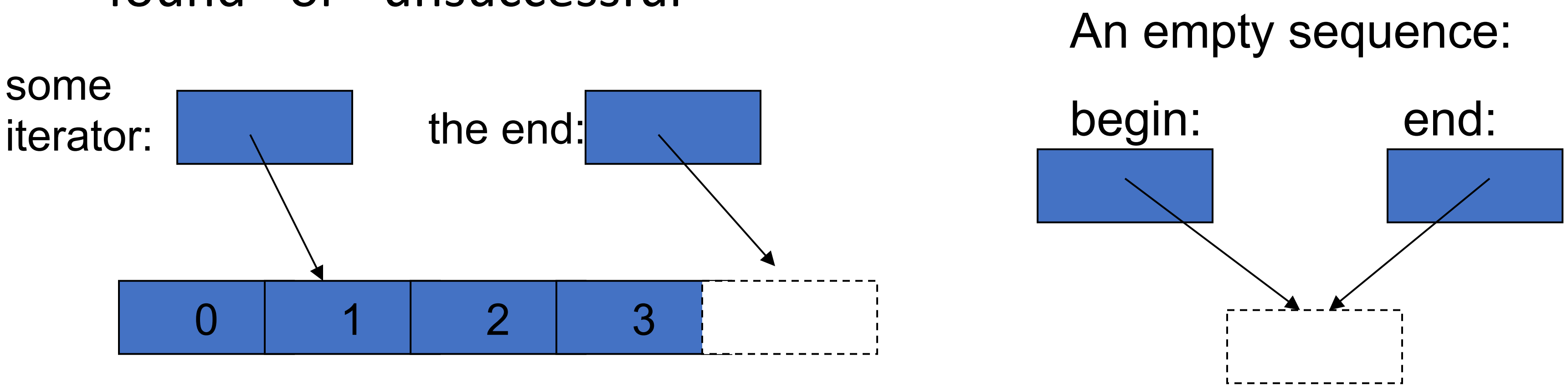
find()

generic for both element type and container type

```
void f(vector<int>& v, int x)              // works for vector of ints
{
    vector<int>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}

void f(list<string>& v, string x)          // works for list of strings
{
    list<string>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}

void f(set<double>& v, double x)           // works for set of doubles
{
    set<double>::iterator p = find(v.begin(),v.end(),x);
    if (p!=v.end()) { /* we found  x */ }
    // …
}
```

Algorithms and iterators

- An iterator points to (refers to, denotes) an element of a sequence
- The end of the sequence is “one past the last element”
  - ***not*** “the last element”
  - That’s necessary to elegantly represent an empty sequence
  - One-past-the-last-element isn’t an element
    - You can compare an iterator pointing to it
    - You can’t dereference it (read its value)
- Returning the end of the sequence is the standard idiom for “not found” or “unsuccessful”



Simple algorithm: **find_if()**

- Find the first element that matches a criterion (predicate)
  - Here, a predicate takes one argument and returns a **bool**

```
template<class In, class Pred>
In find_if(In first, In last, Pred pred)
{
   while (first!=last && !pred(*first)) ++first;
   return first;
}

void f(vector<int>& v)
{
   vector<int>::iterator p = find_if(v.begin(),v.end,Odd());
   if (p!=v.end()) { /* we found an odd number */ }
   // …
}
```

A predicate

Predicates

- A predicate (of one argument) is a function or a function object that takes an argument and returns a **bool**
- For example
  - A function

```
bool odd(int i) { return i%2; } // % is the remainder (modulo) operator
odd(7);                          // call odd: is 7 odd?
```

  - A function object

```
struct Odd {
        bool operator()(int i) const { return i%2; }
};
Odd odd;          // make an object odd of type Odd
odd(7);           // call odd: is 7 odd?
```

Function objects

- A concrete example using state

```
template<class T> struct Less_than {
   T val;        // value to compare with
   Less_than(T& x) :val(x) { }
   bool operator()(const T& x) const { return x < val; }
};

// find x<43 in vector<int> :
p=find_if(v.begin(), v.end(), Less_than(43));

// find x<"perfection" in list<string>:
q=find_if(ls.begin(), ls.end(), Less_than("perfection"));
```

Function objects

- A very efficient technique
  - inlining very easy
    - and effective with current compilers
  - Faster than equivalent function
    - And sometimes you can't write an equivalent function
- The main method of policy parameterization in the STL
- Key to emulating functional programming techniques in C++

Policy parameterization

- Whenever you have a useful algorithm, you eventually want to parameterize it by a "policy".
  - For example, we need to parameterize sort by the comparison criteria

```
struct Record {
    string name;            // standard string for ease of use
    char addr[24];   // old C-style string to match database layout
    // …
};

vector<Record> vr;
// …
sort(vr.begin(), vr.end(), Cmp_by_name());      // sort by name
sort(vr.begin(), vr.end(), Cmp_by_addr());       // sort by addr
```

Comparisons

```
// Different comparisons for Rec objects:

struct  Cmp_by_name {
    bool operator()(const Rec& a, const Rec& b) const
            { return a.name < b.name; }            // look at the name field of Rec
};

struct  Cmp_by_addr {
    bool operator()(const Rec& a, const Rec& b) const
            { return 0 < strncmp(a.addr, b.addr, 24); }      // correct?
};

// note how the comparison function objects are used to hide ugly
// and error-prone code
```

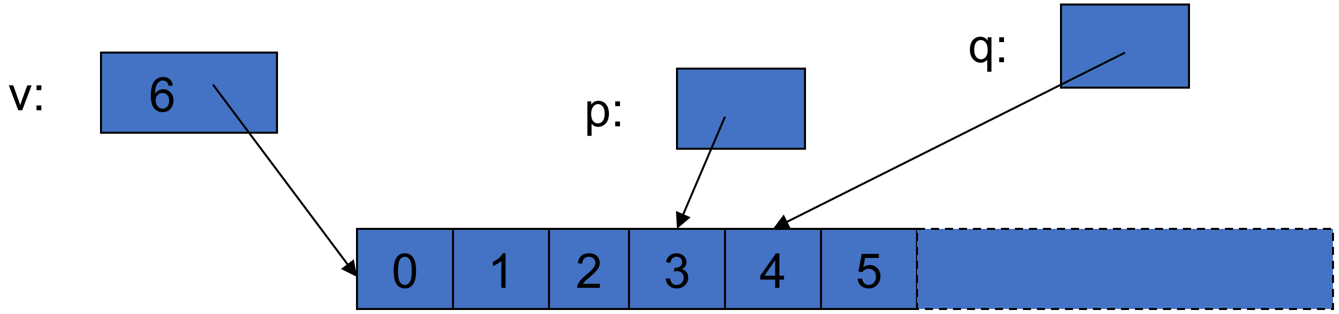
vector

```
template<class T> class vector {
    T* elements;
    // …
    using value_type = T;
    using iterator = ???;          // the type of an iterator is implementation defined
                                   // and it (usefully) varies (e.g. range checked iterators)
                                   // a vector iterator could be a pointer to an element
    using const_iterator = ???;

    iterator begin();              // points to first element
    const_iterator begin() const;
    iterator end();                // points to 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
};
```

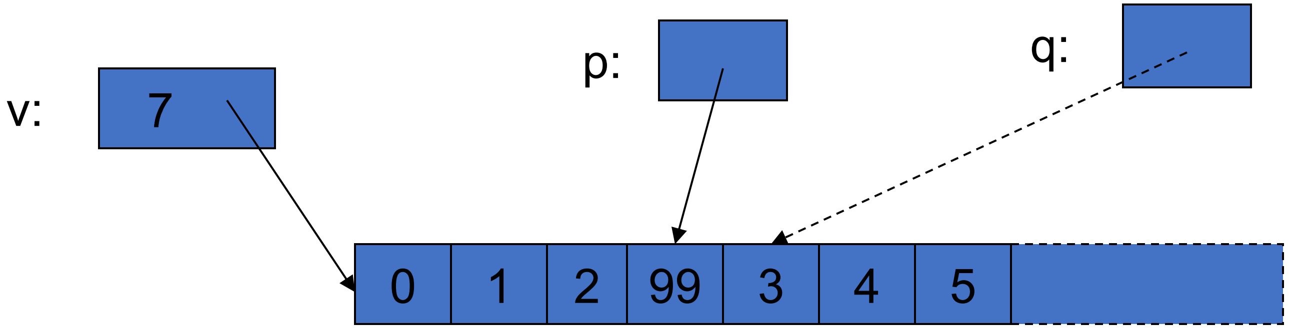
insert() into vector

vector<int>::iterator p = v.begin(); ++p; ++p; ++p;

vector<int>::iterator q = p; ++q;

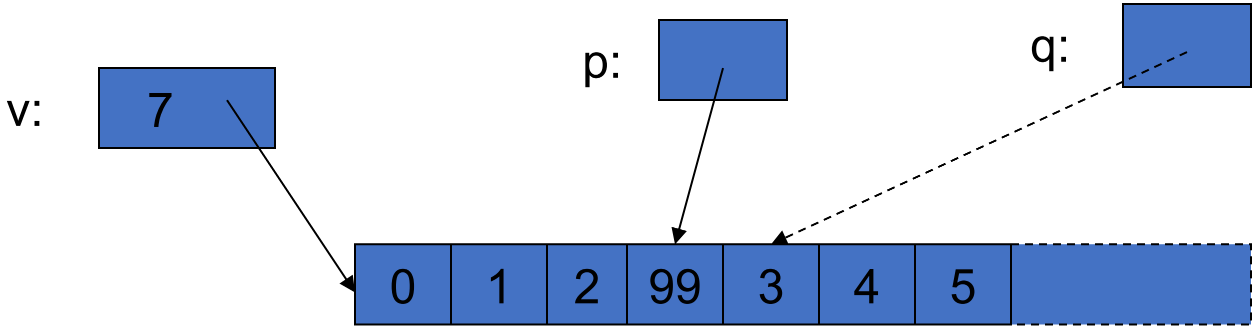


p=v.insert(p,99); // leaves p pointing at the inserted element

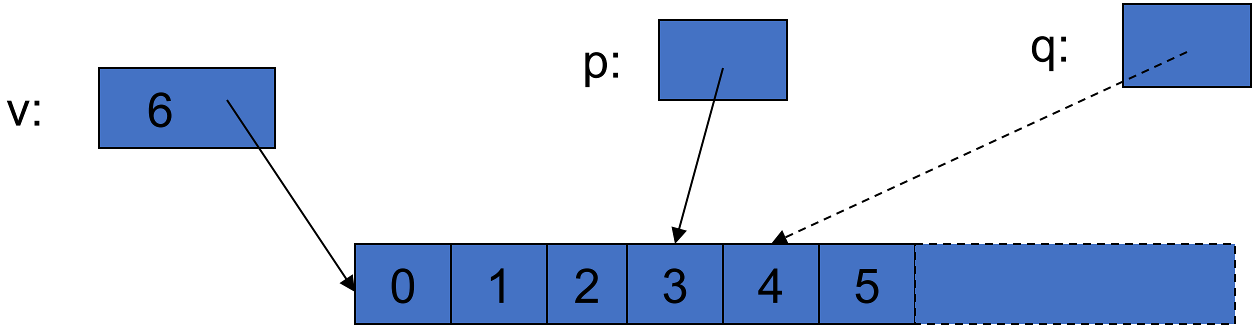


- **Note: q is invalid after the insert()**
- **Note: Some elements moved; all elements could have moved**

erase() from vector



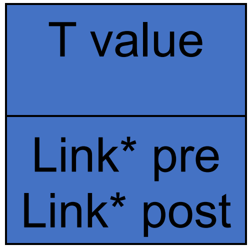
p = v.erase(p);   // leaves p pointing at the element after the erased one



- **vector elements move when you insert() or erase()**
- **Iterators into a vector are invalidated by insert() and erase()**

list

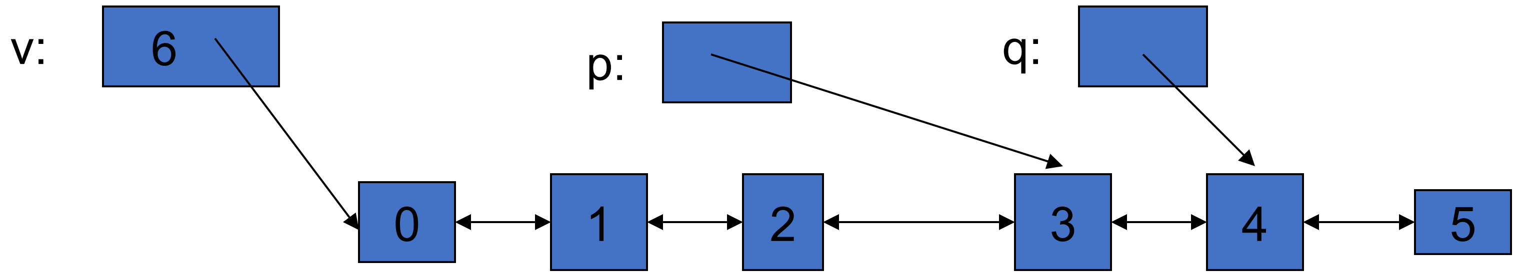
Link:



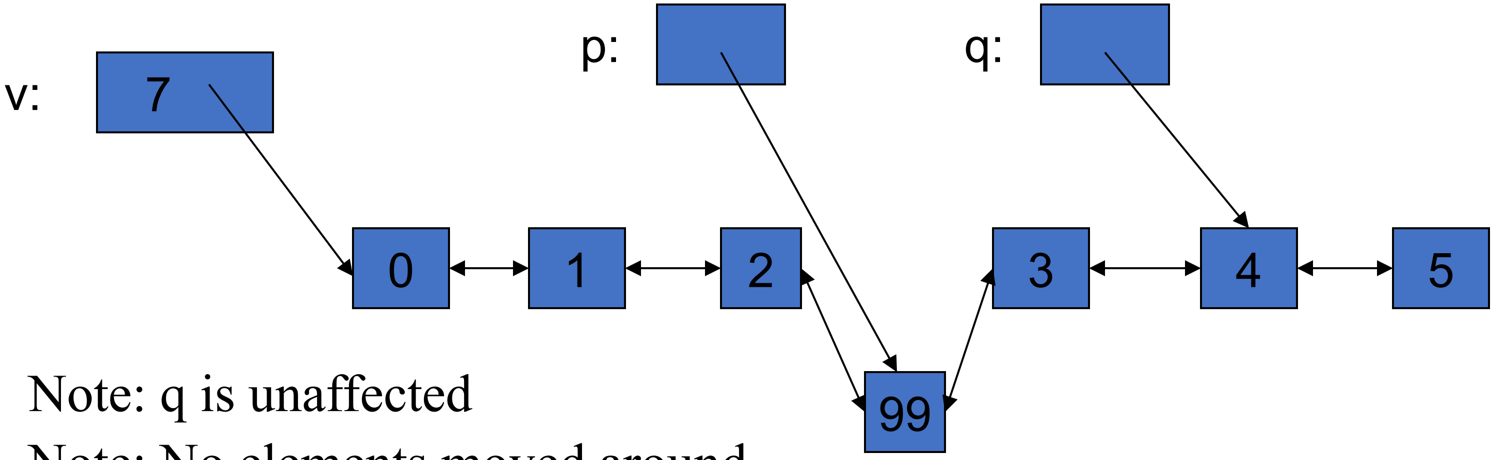

```
template<class T> class list {
    Link* elements;
    // …
    using value_type = T;
    using iterator = ???;          // the type of an iterator is implementation defined
                                   // and it (usefully) varies (e.g. range checked iterators)
                                   // a list iterator could be a pointer to a link node
    using 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
};
```

insert() into list

```
list<int>::iterator p = v.begin(); ++p; ++p; ++p;
list<int>::iterator q = p; ++q;
```

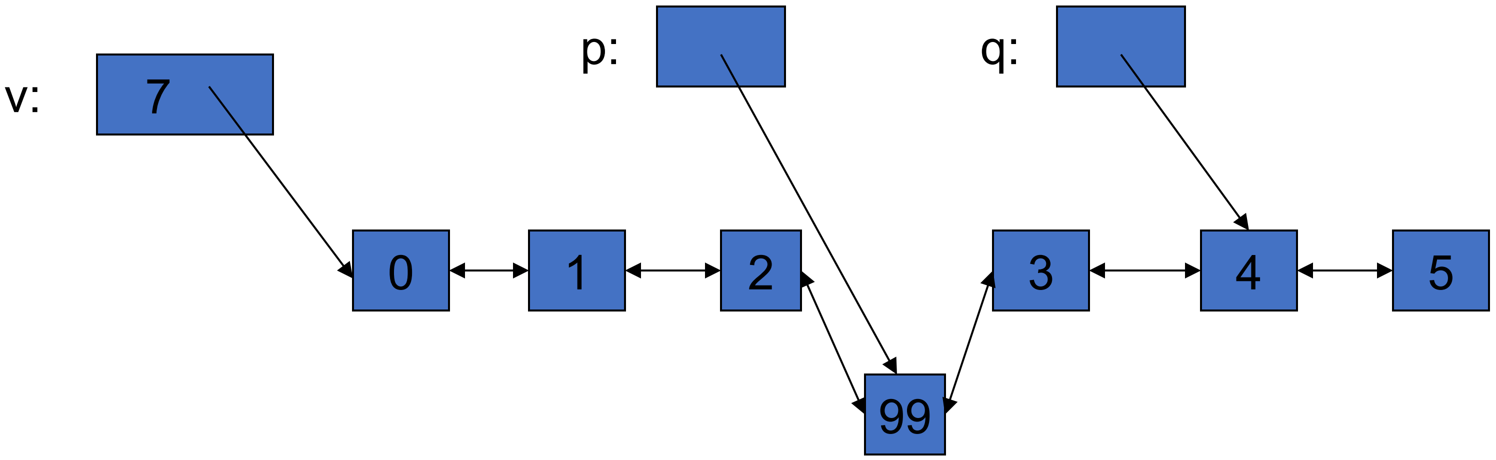


v = v.insert(p,99); // leaves **p** pointing at the inserted element

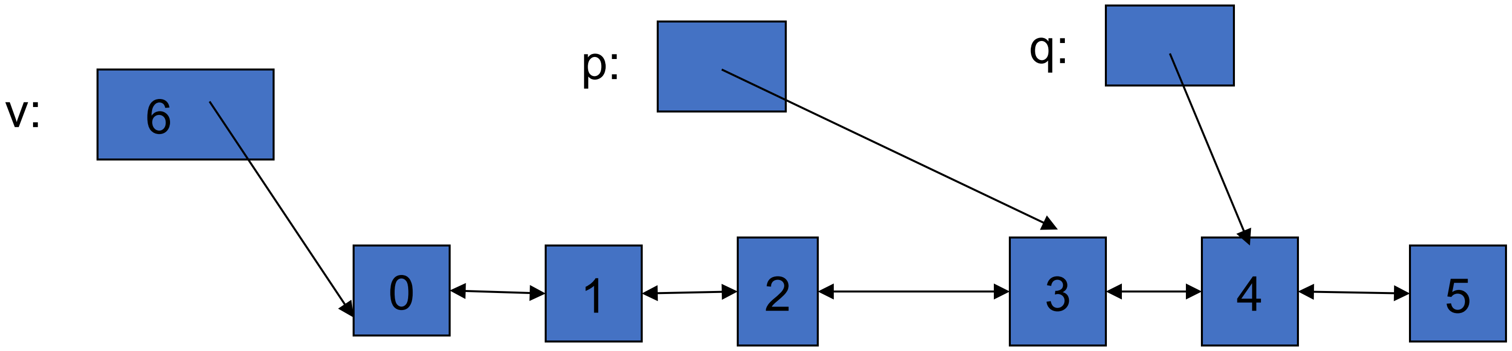


- Note: q is unaffected
- Note: No elements moved around

erase() from list



p = v.erase(p); *// leaves p pointing at the element after the erased one*



- Note: list elements do not move when you insert() or erase()

Ways of traversing a vector

```
for(int i = 0; i<v.size(); ++i)                    // why int?
        … // do something with v[i]

for(vector<T>::size_type i = 0; i<v.size(); ++i)   // longer but always correct
        … // do something with v[i]

for(vector<T>::iterator p = v.begin(); p!=v.end(); ++p)
    …        // do something with *p
```

- Know both ways (iterator and subscript)
  - The subscript style is used in essentially every language
  - The iterator style is used in C (pointers only) and C++
  - The iterator style is used for standard library algorithms
  - The subscript style doesn’t work for lists (in C++ and in most languages)
- Use either way for vectors
  - There are no fundamental advantages of one style over the other
  - But the iterator style works for all sequences
  - Prefer **size_type** over plain **int**
    - pedantic, but quiets compiler and prevents rare errors

Ways of traversing a vector

```
for(vector<T>::iterator p = v.begin(); p!=v.end(); ++p)
    …        // do something with *p

for(vector<T>::value_type x : v)
    …        // do something with x

for(auto& x : v)
    …        // do something with x
```

- “Range **for**”
  - Use for the simplest loops
    - Every element from **begin()** to **end()**
  - Over one sequence
  - When you don’t need to look at more than one element at a time
  - When you don’t need to know the position of an element

Vector vs. List

- By default, use a **vector**
  - You need a reason not to
  - You can “grow” a vector (e.g., using **push_back()**)
  - You can **insert()** and **erase()** in a vector
  - Vector elements are compactly stored and contiguous
  - For small vectors of small elements all operations are fast
    - compared to lists
- If you don’t want elements to move, use a **list**
  - You can “grow” a list (e.g., using **push_back()** and **push_front()**)
  - You can **insert()** and **erase()** in a list
  - List elements are separately allocated
- Note that there are more containers, e.g.,
  - map
  - unordered_map

Some useful standard headers

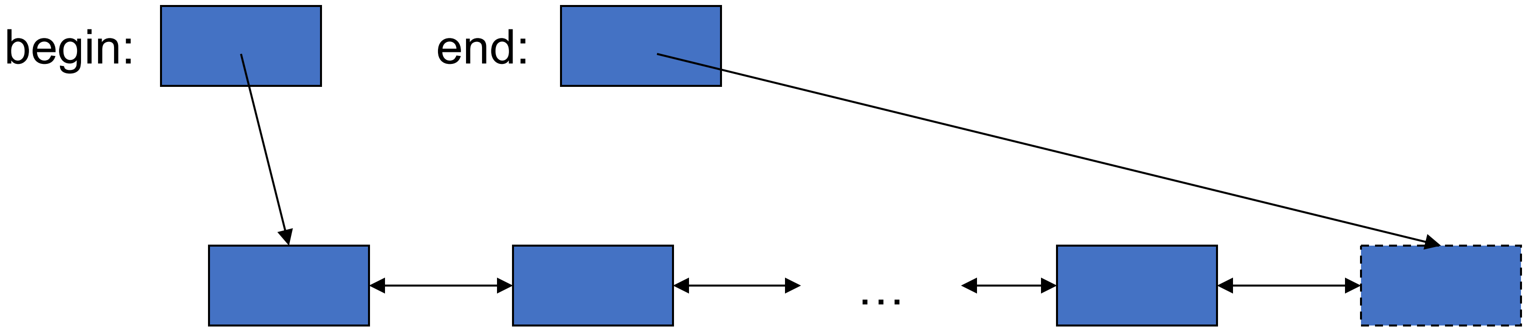
- **<iostream>** I/O streams, cout, cin, …
- **<fstream>** file streams
- **<algorithm>** sort, copy, …
- **<numeric>** accumulate, inner_product, …
- **<functional>** function objects
- **<string>**
- **<vector>**
- **<map>**
- **<unordered_map>** hash table
- **<list>**
- **<set>**

Overview

- Common tasks and ideals
- Containers, algorithms, and iterators
- The simplest algorithm: find()
- Parameterization of algorithms
  - find_if() and function objects
- Sequence containers
  - vector and list
- Algorithms and parameterization revisited
- Associative containers
  - map, set
- Standard algorithms
  - copy, sort, …
  - Input iterators and output iterators
- List of useful facilities
  - Headers, algorithms, containers, function objects

Basic model

- A pair of iterators defines a sequence
  - The beginning (points to the first element – if any)
  - The end (points to the one-beyond-the-last element)



- An iterator is a type that supports the "iterator operations" of
  - ++ Point to the next element
  - *   Get the element value
  - == Does this iterator point to the same element as that iterator?
- Some iterators support more operations *(e.g.*, --, +, and [ ])

Accumulate (sum the elements of a sequence)

```
template<class In, class T>
T accumulate(In first, In last, T init)
{
    while (first!=last) {
            init = init + *first;
            ++first;
    }
    return init;
}
```

v:

1	2	3	4

int sum = accumulate(v.begin(), v.end(), 0); ***// sum becomes 10***

Accumulate (sum the elements of a sequence)

```
void f(vector<double>& vd, int* p, int n)
{
    double sum = accumulate(vd.begin(), vd.end(), 0.0); // add the elements of vd
    // note: the type of the 3rd argument, the initializer, determines the precision used

    int si = accumulate(p, p+n, 0);        // sum the ints in an int (danger of overflow)
                                           // p+n means (roughly) &p[n]

    long sl = accumulate(p, p+n, long(0));        // sum the ints in a long
    double s2 = accumulate(p, p+n, 0.0);          // sum the ints in a double

    // popular idiom, use the variable you want the result in as the initializer:
    double ss = 0;
    ss = accumulate(vd.begin(), vd.end(), ss);  // do remember the assignment
}
```

Accumulate
(generalize: process the elements of a sequence)

```
// we don't need to use only +, we can use any binary operation (e.g., *)
// any function that "updates the init value" can be used:

template<class In, class T, class BinOp>
T accumulate(In first, In last, T init, BinOp op)
{
   while (first!=last) {
            init = op(init, *first);                // means "init op *first"
            ++first;
   }
   return init;
}
```

Accumulate

```
// often, we need multiplication rather than addition:

#include <numeric>
#include <functional>
void f(list<double>& ld)
{
    double product = accumulate(ld.begin(), ld.end(), 1.0, multiplies<double>());
    // …
}

// multiplies is a standard library function object for multiplying
```

Note: multiplies for *

Note: initializer 1.0

Accumulate (what if the data is part of a record?)

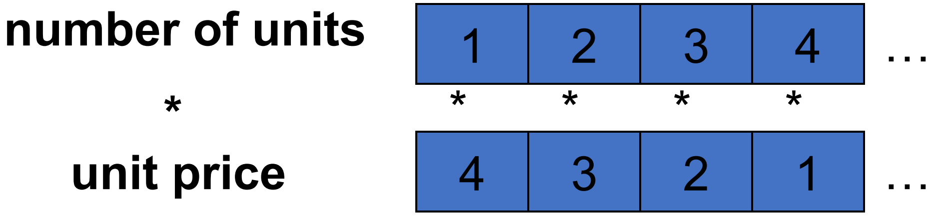
```
struct Record {
    int units;                    // number of units sold
    double unit_price;
    // …
};

// let the "update the init value" function extract data from a Record element:
double price(double v, const Record& r)
{
    return v + r.unit_price * r.units;
}

void f(const vector<Record>& vr) {
    double total = accumulate(vr.begin(), vr.end(), 0.0, price);
    // …
}
```

Accumulate (what if the data is part of a record?)

```
struct Record {
    int units;                          // number of units sold
    double unit_price;
    // …
};

void f(const vector<Record>& vr) {
    double total = accumulate(vr.begin(), vr.end(), 0.0,   // use a lambda
                                                [](double v, const Record& r)
                                                        {  return v + r.unit_price * r.units; }
     );
    // …
}

// Is this clearer or less clear than the price() function?
```

Inner product

```
template<class In, class In2, class T>
T inner_product(In first, In last, In2 first2, T init)
    // This is the way we multiply two vectors (yielding a scalar)
{
    while(first!=last) {
            init  = init + (*first) * (*first2);   // multiply pairs of elements and sum
            ++first;
            ++first2;
    }
    return init;
}
```



Inner product example

```
// calculate the Dow-Jones industrial index:
vector<double> dow_price;            // share price for each company
dow_price.push_back(81.86);
dow_price.push_back(34.69);
dow_price.push_back(54.45);
// …
vector<double> dow_weight;           // weight in index for each company
dow_weight.push_back(5.8549);
dow_weight.push_back(2.4808);
dow_weight.push_back(3.8940);
// …
double dj_index = inner_product(   // multiply (price,weight) pairs and add
            dow_price.begin(), dow_price.end(),
            dow_weight.begin(),
            0.0);
```

Inner product example

```
// calculate the Dow-Jones industrial index:
vector<double> dow_price = {        // share price for each company
    81.86, 34.69, 54.45,
    // …
};
vector<double> dow_weight = {       // weight in index for each company
    5.8549, 2.4808, 3.8940,
    // …
};

double dj_index = inner_product(    // multiply (price,weight) pairs and add
            dow_price.begin(), dow_price.end(),
            dow_weight.begin(),
            0.0);
```

Inner product (generalize!)

```
// we can supply our own operations for combining element values with "init":
template<class In, class In2, class T, class BinOp, class BinOp2 >
T inner_product(In first, In last, In2 first2, T init, BinOp op, BinOp2 op2)
{
   while(first!=last) {
            init  = op(init, op2(*first, *first2));
            ++first;
            ++first2;
   }
   return init;
}
```

Map (an associative array)

- For a **vector**, you subscript using an integer
- For a **map**, you can define the subscript to be (just about) any type

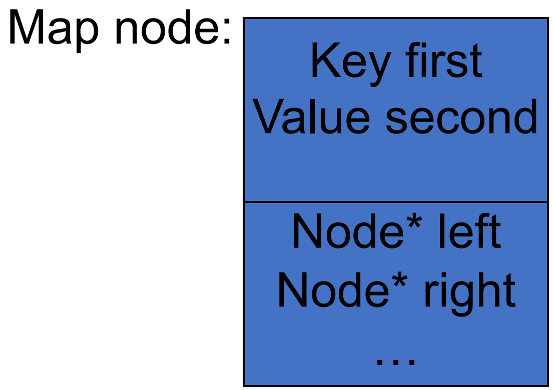
Key type

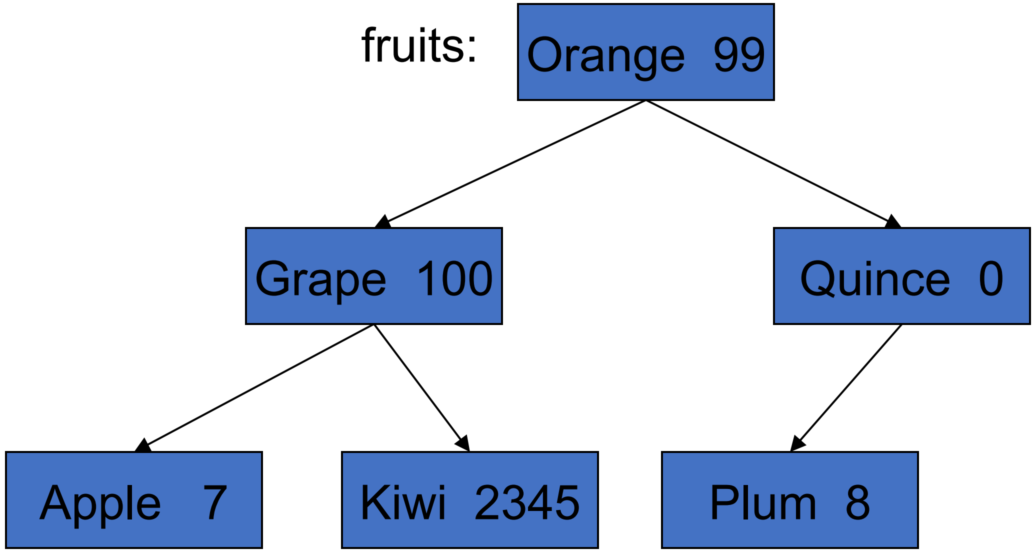
Value type

```
int main()
{
   map<string,int> words;                     // keep (word,frequency) pairs
   for (string s; cin>>s; )
            ++words[s];                        // note: words is subscripted by a
   string
                                               // words[s] returns an int&
                                               // the int values are initialized to 0
   for (const auto&  p : words)
            cout << p.first << ": " << p.second << "\n";
}
```

Map

- After **vector**, **map** is the most useful standard library container
  - Maps (and/or hash tables) are the backbone of scripting languages
- A **map** is really an ordered balanced binary tree
  - By default ordered by **<** (less than)
  - For example, **map<string,int> fruits**;





Some implementation defined type

Map

// note the similarity to ***vector*** *and* ***list***

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

    using iterator = ???;               // probably a pointer to a tree node
    using const_iterator = ???;

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

    Value& operator[ ](const Key&);          // get Value for Key; creates pair if
                                             // necessary, using Value( )
    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 new (Key,Value) pair
    // …                                                 // the bool is false if insert failed
};
```

Map example (build some maps)

```
map<string,double> dow;       // Dow-Jones industrial index (symbol,price) , 03/31/2004
            // http://www.djindexes.com/jsp/industrialAverages.jsp?sideMenu=true.html
dow["MMM"] = 81.86;
dow["AA"] = 34.69;
dow["MO"] = 54.45;
// …
map<string,double> dow_weight;                    // dow (symbol,weight)
dow_weight.insert(make_pair("MMM", 5.8549));  // just to show that a Map
                                        // really does hold pairs
dow_weight.insert(make_pair("AA",2.4808));
dow_weight.insert(make_pair("MO",3.8940));        // and to show that notation matters
// …
map<string,string> dow_name;          // dow (symbol,name)
dow_name["MMM"] = "3M Co.";
dow_name["AA"] = "Alcoa Inc.";
dow_name["MO"] = "Altria Group Inc.";
// …
```

Map example (some uses)

```
double alcoa_price = dow["AA"];      // read values from a map
double boeing_price = dow["BO"];


if (dow.find("INTC") != dow.end())    // look in a map for an entry
    cout << "Intel is in the Dow\n";


// iterate through a map:

for (const auto& p : dow) {
    const string& symbol = p.first;       // the "ticker" symbol
    cout << symbol  << '\t' << p.second << '\t' << dow_name[symbol] << '\n';
}
```

Map example (calculate the DJ index)

```
double value_product(
    const pair<string,double>& a,
    const pair<string,double>& b)          // extract values and multiply
{
    return a.second * b.second;
}

double dj_index =
    inner_product(dow.begin(), dow.end(),          // all companies in index
                  dow_weight.begin(),              // their weights
                  0.0,                             // initial value
                  plus<double>(),                  // add (as usual)
                  value_product                    // extract values and weights
          );                                       // and multiply; then sum
```

Containers and "almost containers"

- Sequence containers
  - **vector, list, deque**
- Associative containers
  - **map, set, multimap, multiset**
- "almost containers"
  - array, **string, stack, queue, priority_queue, bitset**
- New C++11 standard containers
  - **unordered_map** (a hash table), **unordered_set,** …
- For anything non-trivial, consult documentation
  - Online
    - SGI, RogueWave, Dinkumware
  - Other books
    - Stroustrup: The C++ Programming language 4th ed. (Chapters 30-33, 40.6)
    - Austern: Generic Programming and the STL
    - Josuttis: The C++ Standard Library

Algorithms

- An STL-style algorithm
  - Takes one or more sequences
    - Usually as pairs of iterators
  - Takes one or more operations
    - Usually as function objects
    - Ordinary functions also work
  - Usually reports "failure" by returning the end of a sequence

Some useful standard algorithms

- **r=find(b,e,v)** r points to the first occurrence of v in [b,e)
- **r=find_if(b,e,p)** r points to the first element x in [b,e) for which p(x)
- **x=count(b,e,v)** x is the number of occurrences of v in [b,e)
- **x=count_if(b,e,p)** x is the number of elements in [b,e) for which p(x)
- **sort(b,e)** sort [b,e) using <
- **sort(b,e,p)** sort [b,e) using p
- **copy(b,e,b2)** copy [b,e) to [b2,b2+(e-b)) there had better be enough space after b2
- **unique_copy(b,e,b2)** copy [b,e) to [b2,b2+(e-b)) but don't copy adjacent duplicates
- **merge(b,e,b2,e2,r)** merge two sorted sequence [b2,e2) and [b,e) into [r,r+(e-b)+(e2-b2))
- **r=equal_range(b,e,v)** r is the subsequence of [b,e) with the value v (basically a binary search for v)
- **equal(b,e,b2)** do all elements of [b,e) and [b2,b2+(e-b)) compare equal?

Copy example

```
template<class In, class Out> Out copy(In first, In last, Out res)
{
    while (first!=last) *res++ = *first++;
                                        // conventional shorthand  for:
                                        //  *res = *first; ++res; ++first
    return res;
}

void f(vector<double>& vd, list<int>& li)
{
    if (vd.size() < li.size()) error("target container too small");
    copy(li.begin(), li.end(), vd.begin());        // note: different container types
                                                   // and different element types
                                                   // (vd better have enough elements
                                                   // to hold copies of li's elements)
    sort(vd.begin(), vd.end());
    // …
}
```

Input and output iterators

```
// we can provide iterators for output streams

    ostream_iterator<string> oo(cout);        // assigning to *oo is to write to cout

    *oo = "Hello, "; // meaning cout << "Hello, "
    ++oo;            // "get ready for next output operation"
    *oo = "world!\n";          // meaning cout << "world!\n"


// we can provide iterators for input streams:

    istream_iterator<string> ii(cin);   // reading *ii is to read a string from cin

    string s1 = *ii;   // meaning cin>>s1
    ++ii;                          // "get ready for the next input operation"
    string s2 = *ii;   // meaning cin>>s2
```

Make a quick dictionary (using a vector)

```
int main()
{
  string from, to;
  cin >> from >> to;                          // get source and target file names

  ifstream is(from);                     // open input stream
  ofstream os(to);                       // open 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
  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
}
```

Make a quick dictionary (using a vector)

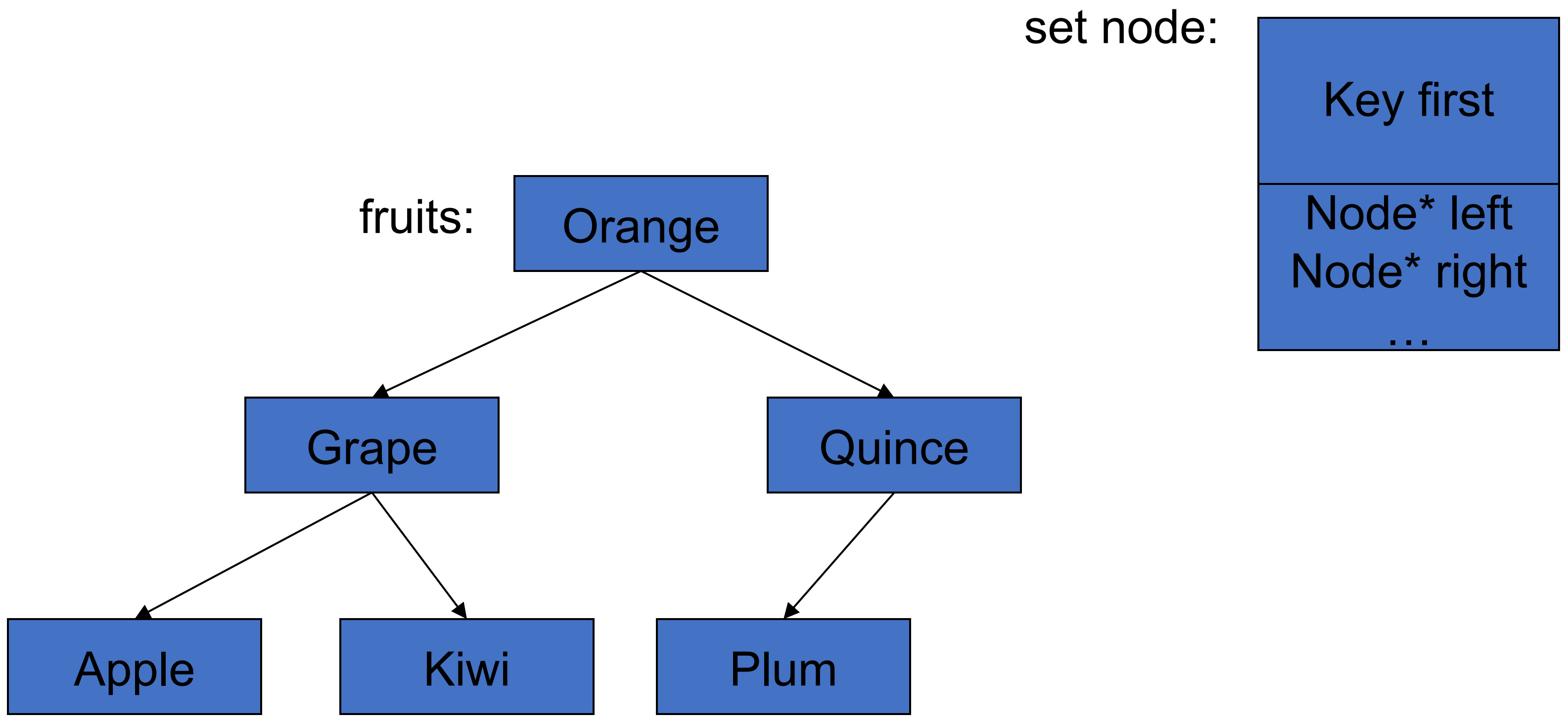
- We are doing a lot of work that we don't really need
  - Why store all the duplicates? (in the vector)
  - Why sort?
  - Why suppress all the duplicates on output?
- Why not just
  - Put each word in the right place in a dictionary as we read it?
  - In other words: use a **set**

Make a quick dictionary (using a set)

```
int main()
{
  string from, to;
  cin >> from >> to;                        // get source and target file names

  ifstream is(from);                   // make input stream
  ofstream os(to);                     // 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
}

// simple definition: a set is a map with no values, just keys
```

Set

- A **set** is really an ordered balanced binary tree
  - By default ordered by <
  - For example, **set<string> fruits**;



copy_if()

```
// a very useful algorithm (missing from the standard library):

template<class In, class Out, class Pred>
Out copy_if(In first, In last, Out res, Pred p)
   // copy elements that fulfill the predicate
{
   while (first!=last) {
            if (p(*first)) *res++ = *first;
            ++first;
   }
   return res;
}
```

copy_if()

```
void f(const vector<int>& v)          // "typical use" of predicate with data
                                      // copy all elements with a value less than 6
{
    vector<int> v2(v.size());
    copy_if(v.begin(), v.end(), v2.begin(),
                     [](int x) { return x<6; } );
    // …
}
```

Some standard function objects

- From <functional>
  - Binary
    - plus, minus, multiplies, divides, modulus
    - equal_to, not_equal_to, greater, less, greater_equal, less_equal, logical_and, logical_or
  - Unary
    - negate
    - logical_not
  - Unary (missing, write them yourself)
    - less_than, greater_than, less_than_or_equal, greater_than_or_equal