C++ : ALBERT'S WEI

Fall 2011

Compiled by Judy Nguyen

A dedication to students that struggle
with listening and taking presentable
notes at the same time.

Table of Contents

Introduction

- multi-paradigm: procedural, OO, generic programming
- creator: Bjarne Stroustrup
- ISO/IEC C++ 1998, 2003, 2011

Some Features of C++

References

```
void swap(int& m, int& n) {
  int tmp = m;
  m = n;
  n = tmp;
}

int a = 1, b = 2;
swap(a,b);
```

Function Templates

```
template<typename T>                  // what follows is a template in header file
T max(T a, T b) {
  return a > b ? a : b;
}

int n = max(2,3);                     // compiler instantiates a version of max with T = int
```

Classes and Objects

eg. Stack Class

```
// Stack.h

~~#define CAPACITY 100~~
template<typename T, size_t CAPACITY = 100>

class Stack {
private:
  T ~~int~~ data_ [CAPACITY];            // end with "_" to indicate private
  size_t size_;
```

```
public:
  // precondition: !full()
  void push(T int n) { data_[size_++] = n; }
  // precondition: !empty()
  void pop() { size_--; }

  bool empty() const { return size_ == 0; }
  bool full()  const { return size_ == CAPACITY; }

  // precondition: !empty()
  T int top()   const { return data_[size_ - 1]; }
  size_t size() const { return size_; }

  // constructor
  Stack(): size_(0){}
};                    \
                       member initializer
```

```
// Stacktest.cpp

...
#include "stack.h"

int main() {
  Stack s;                                  // no need for new !!!

  for(int i = 0; i < 10000; i++) {
    if(s.full())
      break;
    s.push(i);
  }

  while(!s.empty()) {
    cout << s.top() << endl;
    s.pop();
  }
}

Stack<int> s;                               // stack of at most 100 ints
Stack<double, 500> s2;                      // stack of at most 500 doubles
s2.push(3,14);
```

Input/Ouput

eg.

```
#include <iostream>
                    insertion/output operator
int main() { /
  std::cout << "hello, world" << std::endl;
}      |     \
  scope   predefined object associated with standard output
operator
```

- int main() is the same as int main(void)
- main defaults to return 0
- basically, everything in the C++ standard library (except those in C++ part) is in the std namespace
- using declarations: using std::cout;
- using directives: using namespace std; // like "importing" everything from std

eg.

```
int main() {
  using std::cout;
  using std::endl;

  cout << "hello, world" << endl;        // can use cout and endl
  ...
}

void f() {
  std::cout << ...                       // need to prefix with std
}
```

Note: Don't use using declarations/directives in a header file!!

eg.

```
int m = 12, n = 34;

cout << "The sum of " << m << " and " << n << " is " << (m+n) << endl;
         |
fullname is operator <<
two possibilities: a method or an outside function
```

```
int m = 12;

cout << "hello";          ↝ operator <<(cout, "hello");
cout << m;                ↝ cout.operator <<(m);
```

function overloading – multiple functions within the same scope and with same name but different parameter lists

eg.

```
int   m = 12;
float f = 1.23;
cout << m;                ↝ cout.operator <<(m);
cout << f;                ↝ cout.operator <<(f);
```

Note: We cannot add more methods to the ostream class but we can implement non-method operator << to print objects from classes that we implement ourselves.

eg. implement << to print an integer stack object (from last time)

```
ostream&
operator <<(ostream& os, const Stack& s) {
  for(size_t i = 0; i < s.size(); i++)
    os << s.data_[i] << " ";
  return os;
}
```

Note: Need a **friend** declaration!

Some Classes Related to Input/Output

- ios_base // independent of character type

The following are specialization of class templates (where the character type is char). They are defined by using **typedef**:

- ios
- istream/ostream/iostream
- ifstream/ofstream/fstream // file I/O
- istringstream/ostringstream/stringstream // string I/O

eg. **typedef** Stack<**int**> IntegerStack; // IntegerStack is the type Stack<**int**>

Some predefined objects:

- **cout** // ostream object
- **cerr** // ostream object, standard error
- **clog** // ostream object, standard error, buffered
- **cin** // istream object, standard input

Formatted Output

Basically, an ostream object has a data member of type ***ios_base****::fmtflags* that controls the output format. We can retrieve and set the flags using the flags() methods.

ios_base is formatflags

ios_base::fmtflags is a bitmask type

eg. **ios_base**::fmtflags f = **cout**.flags();
...
cout.flags(f);

Two ways to change the output format

1. By invoking a method on the ostream object
2. By using manipulators

Methods

eg. setf/unsetf // set/unset a flag

eg. **cout.setf**(iso_base::showpos); // eg. +123

cout.unsetf(iso_base::showpos); // eg. 123

iso_base::**dec** ┐
oct | (group name) iso_base::basefield
setw ┘

cout.setf(ios_base::**setw**, ios_base::basefield);

Manipulators

eg.   **int** n = 13;

cout << **setw** << n << **endl**;   // d
|
manipulator

Some manipulators:

- dec/oct/**setw**
- fixed/scientific   // for floating-point numbers (default: general)
- showpos/noshowpos   // 123 / +123
- showbase/noshowbase   // for integers; 123 / 0x123 / 0123
- uppercase/nouppercase   // 1a2b / 1A2B; 1.23e+001 / 1.23E+001
- boolalpha/noboolalpha   // for boolean type
  /   \
  false/true   0/1
- left/right/internal — -␣␣␣123
  \   └ ␣␣␣-123
  -123␣␣␣
- setprecision   // number of decimal places
- **setw**   // set width   **#include** <iomanip>
- setfill   // set fill character

eg.   **double** d = 1.23456;
cout << **setprecision**(3) << d << **endl**;   // 1.235

int n = 123;
cout << **setw**(6) << n << **endl**;   // ␣␣␣123
setfill('0') << **setw**(6) << n << **endl**;   // 000123

Note: The width needs to be set every time we perform a formatted output. It then reverts to 0.

How do manipulators work?

```
cout << setw;  ↝ hex(cout)
           \
        name of function (function pointer)
```

The ostream class has method that looks like the following:

```
ostream&
operator <<(ios_base&(*pf)(iso_base&)) {
  pf(*this);
  return *this;
}
```

```
ios_base& setw(ios_base& x) {
  x.setf(ios_base::setw, ios_base::basefield);
  return x;
}
```

Input

eg.

```
int m, n;
cin >> m >> n;
└---------┘     \
returns         extraction/input operator
original
   cin
```

Input may fail. How do we detect failure?

An istream object has a data member of type ios_base::iostate (bitmark type) that indicates the state of the object.

There are predefined values for ios_base::iostate:

- ios_base::goodbit (defined to be 0)
- ios_base::eofbit
- ios_base::failbit
- ios_base::badbit

There are methods that test for these:

- good()
- eof()
- failbit() // test for both failbit and badbit
- bad()

Example:

```
int n;
cin >> n;
```

input	n	stateofcin
123␣456	123	good
abc	unchanged	failbit set (eof bit not set)
123■	123	eofbit set (failbit not set)
■	unchanged	both failbit and eofbit set

■ is end of file

eg. **Summing Integers**

```
int n, sum = 0;                123 456 789■

while(1) {
  cin >> n;
  if(cin.fail())               ← this is complicated
    break;
  sum += n;
}

cout << sum << endl;
```

In general, if an istream object is not in a good state, further input operations will fail. We can use the clear() method to put the object in a good state.

It turns out that an istream object can be used as a truth value.

eg. `if(cin) {...}`

The istream object is true iff fail() returns false

eg. **cin** is true iff **cin**.fail() returns false

eg. **Summing Integers**

```
int n, sum = 0;

while(cin >> n)          // as long as we can read an integer
  sum += n;
cout << sum << endl;
```

How do we throw away invalid input? Use the ignore() method:

eg.
```
ignore()                 // ignore one character
ignore(128)              // ignore 128 characters
ignore(128, '\n')        // ignore 128 characters or up to and including the new line
 |                          character
can use this to throw away a line
```

```
#include <limits>

cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

We'll implement a manipulator to do this:

```
ios_base& ignoreline(ios_base& x) {
  x.clear();
  x.ignore(numeric_limits<streamsize>::max(), '\n');
  return x;
}

int n;
```

eg. **cin** >> ignoreline >> n;

eg. **Summing Integers**

```
int n, sum = 0;

while(1) {
  if(cin >> n)           // as long as we can read an integer
    sum += n;
  if(cin.eof())
    break;
  if(cin.fail())
    cin >> ignoreline;
}
```

For iterative input, it's better to read line-by-line

eg.
```
char line[1024];
cin.getline(line, 1024, '\n');
```

Problem: We don't know how long the line is going to be.

getline can fail for two reasons:

1. It didn't read anything
2. Input line is too long

We'll use the *String* class

Strings

```
#include <string>

string  s,                    // empty string
        s2("hello"),
        s3(5, '*'),           // string of 5 *'s
        s4(s2);               // copy of s2
```

Can compare string objects using ==, !=, <, <=, …

```
if(s == "hello")              // can mix string objects with C style strings
```

Concatenate strings +, +=

eg.

```
string s1("hello"), s2("world");

cout << (s1 + " " + s2 + "!") << endl;
```

eg.

```
int main(int argc, char* argv[])
...

string s1(argv[1]);

if(s1 == "-A")
```

- length() and size() both return the length of a string
- operator[] can be used to index into a string
- max_size() returns the maximum possible number of characters
- string can store (this typically returns a very large value)

Standard Idiom: To Process a String Object Character by Character

s – string

```
for(string::size_type i = 0; i < s.size(); i++)
  /* process s[i] */
```

eg. Converting a string to all uppercase

```
void uppercase(string& s) {
  for(string::size_type i = 0; i < s.size(); i++)
    s[i] = toupper(s[i]);
}
```

eg. Returns a string that is the all uppercase equivalent of a given string

```
string uppercase(const string& s) {
  string t(s);            // calls copy constructor

  for(string::size_type i = 0; i < t.size(); i++)
    t[i] = touppercase(t[i]);

  return t;
}

// Think about reversing a string
```

We can use the getline function to read a string

eg.

```
string word;

if(cin >> word)               // read a word
  ...
```

eg.

```
string line;

if(getline(cin, line))        // read a line
  ...

getline(stream, str, delim)
```

Keeps extracting characters from stream and storing them in str until either

1. End of file is encountered (sets eofbit)
2. A character equal to delim is extracted in which case the character is thrown away
3. So many characters have been extracted that it exceeds the max_size() of the string (set failbit)

   If no characters are extracted, also sets failbit

Since max_size() is very large, we assume #3 does not happen. Then, …

Standard Idiom: To Process Input Line By Line

in – istream

```
string line;

while(getline(in, line))
  /* process line */
```

istringstream

```
#include <sstream>
```

eg.

```
istringstream iss("123hello_world");

int n;
string word;

if(iss >> n)
  cout << n << endl;              // 123
if(iss >> word)
  cout << word << endl;           // hello
if(iss >> word)
  cout << word << endl;           // world
if(iss >> word)
  cout << word << endl;           // fails
iss.clear();                      // important to do this!!
iss.str("goodbye");               // resets the read position to beginning
if(iss >> word)
  cout << word << endl;           // goodbye
```

ostringstream

```
#include <sstream>
ostringstream oss;

int n = 123;

oss << setw << n << " " << oct << n;
cout << oss.str() << endl;
```

Advantage: We don't need to change the state of the **cout**

Example: Summing Integers

```
string  line;
int     sum = 0, n;

while(1) {
  cout << "Enter an integer: ";

  if(!getline(cin, line)) {
    cin.clear();
    break;
  }

  istringstream iss(line);          /* a new istringstream every time,
  if(iss >> n)                         otherwise we may need to call
    sum += n;                          clear() */
}

cout << sum << endl;

// input: 123hello ↩   ↝123
```

More stringent input validation:

Idea:

- read first word from line
- read integer from word (this should succeed)
- a word from that word (this should fail)

File Input/Output

```
#include <fstream>
```

ifstream/ofstream/fstream
(input)   (output) (both input + output)

eg.   Summing Integers Stored in a File as Text

```
ifstream in("integers.txt");

if(!in) {                                         if want to use header from C:
  cerr << "unable to open file" << endl;          #include <cstdlib>
  exit(1);                                        (for exit)
}

int n, sum = 0;

while(in >> n)
  sum += n;

cout << sum << endl;

// destructor automatically closes the file
```

constructors (ctor)   destructors (dtor)
                                        \
                                   called when an object is destroyed

What about binary files??

The constructor can take 2 arguments: filename and open mode. Open mode is specified by values of type ios_base::openmode; the possible values are:

- ios_base::in
- ios_base::out
- ios_base::trunc   (truncate – delete content)
- ios_base::app
- ios_base::binary
- ios_base::ate   (at end)

The valid combinations are:

in	out	trunc	app	stdio equivalent
✓				"r"
	✓			"w"
	✓	✓		"w"
	✓		✓	"a"
✓	✓			"r+"
✓	✓	✓		"w+"

Note: There is no "a+" mode. The above can be combined with binary.

eg. ifstream in("data", ios_base::in | ios_base::binary);

Note: need to specify "in"

File Copy Program (for text files)

```
int main(int argc, char* argv[]) {
  if(argc != 3) {
    cerr << "usage: " << argv[0] << "{source}{destination}\n";
    return 1;
  }

  ifstream in(argv[1])
  if(!in) {
    cerr << "unable to open " << argv[1] << endl;
    return 2;
  }

  ostream out(argv[2]);
  if(!out) {
    cerr << "unable to open " << argv[2] << endl;
    return 3;
  }

  copy(in, out);
}

void copy(istream& is, ostream& os) {
  char c;
  while(is.get(c))
    os.put(c);
}
```

File Copy Program Version 2: Copy Line by Line

```
void copy(istream& is, ostream& os) {
  string line;

  while(getline(is, line)) {            // Note:  getline throws away the newline char
    if(is.eof())
      os << line;
    else
      os << line << endl;
  }
}
```

File Copy Program Version 3: Copy Block by Block

```
char a[512];
cin.read(a, 512);                       // read + store up to 512 characters into a set both eofbit +
                                           failbit bits on end-of-file return input stream

cin.gcount();                           // return no. of characters read by last unformatted input

void copy(istream& is, ostream& os) {
  char a[512];
  streamsize n;

  while(1) {
    is.read(a, 512);
    if((n = is.gcount()) > 0)           // if we have read any characters
      os.write(a, n);
    if(!is)
      break;
  }
}
```

File Copy Program Version 4

```
void copy(istream& is, ostream& os) {
  os << is.rdbuf();                     // or  is >> os.rdbuf();
}
```

Seeking Within a Stream

Two types:

```
ios::pos_type   - for storing a position
ios::off_type   - for storing an offset (a signed integer type)
  \
dependent on the character type
```

Recall that ios_base is independent of the character type

We can use tellp/tellg to get our current position

```
          /      \
       put        get
```

Conceptually the position to write (put) may be different from the position to read (get). But for file streams, there is only one position.

```
                          some stream
                          /
eg.   ios::pos_type pos = s.tellp();        // tellp/tellg returns ios::pos_type (-1) on failure
```

We can use seekp/seekg to change the current position

```
eg.   if(!s.seekp(pos)) {                  // seekp failed
        ...
      }
```

There are versions of seekp/seekg that take an offset

```
seekp(ios::off_type, ios_base::seekdir)
                                 \
                                  3 possible values:
                                  ios_base::beg
                                  ios_base::end
                                  ios_base::cur
```

```
eg.   if(!s.seekp(ios::off_type(-5), ios_base::cur)) {  // seekp failed
        ...
      }
```

Note: It may be possible to seek past the end of file.

Example:

```
#include <sstream>

string word;
stringstream ss ("hello");                 // both put + get are at the beginning

ss << "hi";

if(ss >> word)
  cout << word << endl;                     // hillo
ss.clear();                                 // clear eofbit
ss << "goodbye";                            // higoodbye

if(ss >> word)
  cout << word << endl;                     // dbye
```

Miscellaneous Facts about I/O

eg.

```
int n;
cin >> setw >> n;              // read hexadecimal number eg. 1a, 0x1a
```

eg.

```
char a[10];
cin >> setw(10) >> a;          // read at most 9 characters (pad with null char)
```

eg.

```
skipws/noskipws
  /           \
skip         don't skip
leading
whitespace

int n;
cin >> noskipws >> n;
```

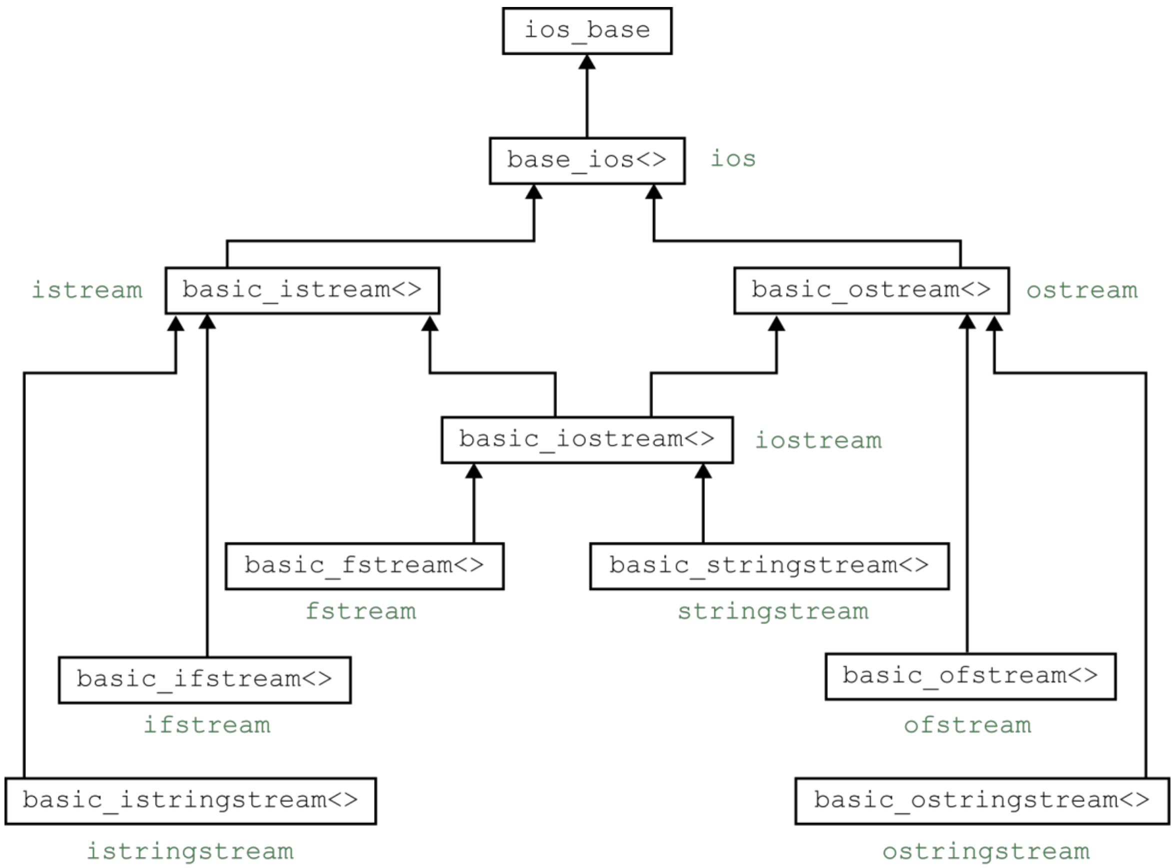
eg. ws – throw away whitespace

```
int n;
string line;

cin >> n;                      // input: 123↩albert ↩
getline(cin, line);

cin >> n;
cin >> ws;
getline(cin, line);
```

Classes Related to I/O



The class templates are parameterized by character type of character traits (+ by the allocator for stringstreams)

eg. ios_base::setw can be referred to as ios::setw

Default Arguments

```
                         default arguments
                      /          |          \
eg.   double volume(double = 1.0, double = 1.0, double = 1.0);

      volume() is the same as       volume(1.0, 1.0, 1.0);
      volume(3.14)                  volume(3.14, 1.0, 1.0);
      volume(3.14, 2.71)            volume(3.14, 2.71, 1.0);
      …
```

Note: Default values are specified in the prototypes which go in header files

If a default value is specified for a parameter, all later parameters in parameter list must have default values.

```
void f(int, int = 1, int);            // not valid
void f(int, int = 1, int = 2);        // OK, can be called with 1, 2, or 3 argv

f(2, 3) → f(2, 3, 2)
       └ rewritten by complier
```

Inline Functions

eg.
```
inline int square (int n) {           // ask compiler to "inline-expand" call this function
  return n * n;
}
```

If the call is "inline-expanded," it doesn't require a jump to a function. inline is just a request; the compiler can choose to ignore it.

eg.
```
int n = 123;
cout << square(n) << endl; ↝ cout << (n * n) << endl;
                           if expanded
```

As the compiler needs to see the code of an inline function in order to expand calls to it, definitions of inline function should be put in header files.

Inline functions can replace macros.

eg.
```
#define SQUARE(x)  ((x)*(x))

cout << SQUARE(x) << endl;
cout << ((x)*(x)) << endl;            // expanded
```

An inline function can typecheck its arguments

A Brief Intro to Classes

class declaration vs. class definition or class implementation

```
class C;                          // class definition – it tells the compiler that C is a class.
                                     Occasionally necessary to break a circular dependency.
```

eg.

```
class C
class D {
private:
  C *p_;
  …
};

class C {
private:
  D *d_;
};

class C {                         // class definition; typically in header files
  …                               // data members and member functions
};
```

eg.

```
//c.h                             // c.cpp (class implementation)
class C {                         #include "C.h"
private:
  int value_;                     c::c(int value): value_(value) {}
public:                           int c::value()const {
  C(int value);                     return value_;
  int value() const;              }
};
```

If the definition of a member function is written the class definition, it is implicitly inline

```
class C {
private:
  int value_;
public:
  C(int value): value_(value){}         // implicitly inline
  int value() const;
};

inline int
C::value() const {
  return value_;
}
```

Macros vs. Inline Functions

```
inline int square(int n) { return n * n; }              // #1

#define SQUARE(x) ((x)*(x))                              // #2
```

#1 only works for **int**, #2 works for all numeric types
#2 does not typecheck its argument. We can use inline function templates

```
template<typename T>
inline T square(T x) { return x * x; }
```

Problem with macros: A macro may evaluate its argument more than once.

```
eg.     int n = 1;
        cout << SQUARE(n++) << endl;              // result is undefined
                └---------------┘
        ((n++)*(n++)) n is incremented twice!!
```

Inline functions do not have this problem

const

- a **const** "variable" has external linkage in C but **int**ernal linkage in C++

  C:
  ```
  /* file1.c */
  const int n = 10;
  ```
  ```
  /* file2.c */
  extern const int n;
  void print_const(void) {
   printf("%d\n", n);
  }
  ```
  Links fine in C.
  But does not link in C++.

- In C++, put **const** in header file

- A method can be declared **const**

  ```
  class C {
  private:
   int value_;                          same as: this → value_ = value;
  public:
   void set(int value) { value_ = value; }
   void get() const { return value_; }
  }                 |                    \
  this method does not          same as this → value_;
  modify the object
  (this pointer has type const C*)

  type: const C*  --- we can't use it to change the object
  ```

Header Files

1. constants eg. `const int n = 10;`

2. function prototypes (with default arguments if applicable)
   eg. `int square(int = 0);`

3. typedefs

4. definitions of inline functions

   eg.
   ```
   inline double cube(double x) {
     return x * x * x;
   }
   ```

5. function and class templates

6. class definitions eg. class C{ … }

References

- a reference is just an alias

- a reference must be initialized when it is created

- anything done to a reference is done to the referent

eg.
```
int n = 10;
int& r = n;
r = 9;              // changes n to 9
int& q = 1;         // doesn't compile
const int& s = 1;   // OK
```

eg.
```
void swap(int& a, int& b) {
  int tmp = a;
  a = b;
  b = tmp;
}
```

int m = 1, n = 2

m $\boxed{1}$ n $\boxed{2}$

a is initialized to m

b is initialized to n

- **Disadvantage:** It is difficult to tell whether a function modifies its argument when we use references

Because a function can return a reference, we can have code like:

```
&f();    f() = 1;
```

eg.
```
int& f(int& n) {
  return n;
}
```

Then, we can have:

```
int m = 1;
f(m) = 2;                  // changes m to 1
```

A function should not return a reference to an object local to the function:

eg.
```
int& f() {                 // BAD: n is destroyed when f returns
  int n = 10;
  return n;
}
```

eg.
```
ostream& operator << (ostream& os, …);
```

- references to **const** a temporary object

```
const int& r = 1;

int n = 1;      long t = n;      // OK
long& s = n;                     // n needs to be converted to a long and that long is a
                                    temporary object + we need to declare the reference as a
                                    reference to a constant object
```

/

this extends the lifetime of the temporary object to the lifetime of s

```
int n = 1;      f(n); calls #2
void f(long&);          // #1
void f(const long&)     // #2
```

Array of references are not allowed

Pointers to references are also not allowed

```
int *p;           int a[10];
int *&r = p;      int(&r)[10] = a;      // reference to array (right-left rule)

// reference to pointer
```

Function Overloading

1. ```
   void f(int&);            // function overloading
   void f(float&);
   ```

2. ```
   void f(int);
   void f(int = 2);         // error
   ```

3. ```
   void f(int);
   void f(const int);       // re-declaration of f
   ```

Function overload resolution (function matching):
process of determining which, if any, among a set of overloaded functions is actually invoked by a particular function call.

Example

```
int max(int, int);
double max(double, double);
int max(const int a[], size_t n);
int min(int, int);

class C {
   …
  static int max(int, int);
};
```

eg. `cout << max(1.3, 2) << endl;`

Three possible outcomes of function matching:

1. There is no match

2. There is a best match

3. There are multiple matches but no best one. We say the call is ambiguous.

Steps of Function Matching

1. Find all candidate functions

2. From candidate functions, find viable functions

3. Rank viable functions and execute the best one
```

Steps to Find Candidate Functions

1. Matching name
2. Viable functions: functions that can actually be called with the arguments provided
3. Rank the viable functions by ranking the conversion needed by each argument

   The rank of a conversion from high to low is as follows:

   i. Exact match
      For pointers and references, there are two sub levels

      a. without qualification conversion

      b. with qualification conversion

   ii. Promotion
      float → double
      smaller integer type → int

   iii. Standard conversions

   iv. User defined conversions: conversions between objects of different classes

A viable candidate is best if:

1. For any other viable candidate, there cannot be a conversion whose rank is higher than the rank of the corresponding conversion of the best candidate.
2. For any other viable candidate, the best candidate must have a conversion with higher rank than the corresponding conversion for the other candidate.

```
int max(int, int);                // #1
double max(double, double);   // #2
```

```
std. conv ┌ exact
max(1.3, 2);    // #1
      /     \
exact     std conv #2 call is ambiguous
```

Examples

1.
```
void f(long);           // #1
void f(float);          // #2
short s = 1;
f(s);
```

ambiguous (both std conversions)

2.
```
void g(int&);           // #1
void g(double);         // #2
short s = 1;
g(s);                   // calls #2 because #1 is not viable (can't create reference to
                           temporary without const)
```

3.
```
void h(const int&);
void h(double);
short s = 1;
h(s);                   // calls #1 (exact match)
```

4.
```
void p(char*);          // #1           char* p = "hello";
void p(const char*);    // #2
p("hello");             // calls #2
```

Dynamic Memory

- new + delete

eg.
```
int *p = new int[100];          int *q = new int(5);
p[0] = 1;                       cout << *q << endl;     // 5
…                               delete q;
delete[] p;
```

Call delete[] if we call new with []

```
int *r = new int[1];
…
delete[] r;
```

- call still use malloc + friends

eg.
```
int *p = (int *) malloc(100 * sizeof(int));
                 |
            need this cast      C++ has stricter type-checking than C
```

- new calls constructors, malloc does not
  delete calls destructors, free does not

eg.
```
class C {
public:
  C();                                   // default constructor
  C(int);                                // constructor
  ~C();                                  // destructor
  …
};

C *p = (C*) malloc(sizeof(c));           // does not call constructor
C *q = new C;                            // calls default constructor
C *r = new C(4);                         // calls constructor that takes an int
C *s = new C[100];                       // calls default constructor 100 times
...

delete q;                                // calls destructor
delete r;                                // calls destructor
delete[] s;                              // calls destructor 100 times

C a[100];                                // calls default constructor 100 times
C *b[100];                               // does not call constructor
b[0] = new C(5);
```

- new throws a bad_alloc exception when it fails

eg.
```
try {
  char *p = new char[2000000000];
  ...
} catch(const bad_alloc& e) {
  cerr << e.what() << endl;
}
```

[The string class – available in SHAREOUT]

Example

- “trimming” leading whitespace from a string

```
string ltrim(const string& s) {
  string::size_type i;

  for(i = 0; i < s.size(); i++)
    if(!isspace(s[i]))
      break;
  return s.substr(i);
}
```

- tokenizing a string

eg. tokenize abd:d;:ef/g: at the delimiters :;/
expected tokens are: abc d ef g

logic: find first non-delimiter from starting position, then find the first delimiter from that point

Note: This version does not allow empty tokens

```
bool next_token(const string& s, const string& delimiters, string& token,
                          string::size_type& start) {
  string::size_type pos;
  pos = s.find_first_not_of(delimiters, start);

  if(pos = string::npos)                                        /* no more tokens */
    return false;

  start = s.find_first_of(delimiters, pos);
  if(start == string::npos)
    token = s.substr(pos);
  else
    token = s.substr(pos, start - pos);
  return true;
}
```

Standard Template Library (STL)

[The Standard Template Library (STL) – available in SHAREOUT]

- Implemented as class templates and function templates
- consist of

1. Containers eg. vectors, lists, maps, ...
2. Algorithms eg. find, find_if, sort, ...
3. Iterators give between containers and algorithms
   - algorithms work on iterators
   - containers provide iterators
4. Function objects – can be used to customize algorithms

   Function objects are objects that act like functions.
   This is achieved by overloading operator()

eg.

```
class C {
private:
  size_t count_;
public:
  C(): count_(0) {}
  void operator()() {
    cout << ++count_ << endl;
  }
}
```

Algorithms

- Look for an integer in an array of integers

```
size_t find(const int a[], size_t n, int x) {
 size_t i;

 for(i = 0; i < n; i++)
  if(a[i] == x)
   return i;
 return -1;
}
```

What if we have a linked list? The above version doesn’t generalize.

- use pointers int a[] = {3, 2, 7, 6, 8};

```
const int *find(const int *first, const int *last, int x) {
 while(first != last) {
  if(*first == x)
   return first;
  ++first;
 }
 return last;
}
```

This version can be generalized to apply to STL container. Iterators are generalization of pointers.

- Make the above into a function template.

```
template<typename T>

T *find(T *first, T *last, T x) {     // this can only be applied to pointers
 // same implementation
}
```

type char

~~const~~ char *p = “hello”; /

~~const~~ char *q = find(p, p+5, ‘&’);

Can make the function more general

```
template<typename T, typename S>

T find(T first, T last, S x) {      // this can be applied to iterators. It can be applied to
  // same implementation               object that support *, ++, !=
}
```

- Iterators are generalization of pointers. We can think of them as objects that point to other objects. Given an iterator it. We can use *it to refer to the object pointed by it.

  By generalizing pointers in different ways, we get 5 categories of iterators:

1. Input Iterator: supported operations are *it, ++it, it++, it1 == it2, it1 != it2
   They are for reading and for single-pass algorithms

2. Output Iterator: supported operations: *it = x, it++, ++it
   Single-pass

3. Forward Iterator: refinement of Input + Output Iterator
   ie. supports all the operations of Input + Output Iterator

4. Bidirectional Iterator: refinement of Forward Iterator
   Additional operations: --it, it--

5. Random Access Iterator: refinement of Bidirectional Iterator
   Additional operations: it ±n, it1 < it2, ...

Containers

- container get a copy of the objects put into them

- begin() and end()

  c.begin() returns an iterator pointing to the first object
  c.end() returns an iterator one past the last element

- the STL uses half-open range [it1, it2) // include it1, exclude it2

Standard Idiom: To Iterate Through the Objects in a Container

```
for(it = c.begin(); it != c.end(); ++it)
  /* process it */
```

bitset

```
#include <bitset>
bitset<8> a,                              // all 0's
          b(string("11001001"))
```

operations: &, 1, ^, &=, 1=, ^=

```
a.set();                                  // set all bits to 1
cout << a[1] << endl;
a.flip();                                 // flip all bit
a.reset();                                // set all bits to 0
```

Prime Numbers

Find all primes less than 1,000,000,000

Sieve of Eratosthenes:

2, 3, 4, 5, 6, 7, 8, 9, ~~10~~, 11, 12, 13, 14, ~~15~~, 16, 17, 18, 19, ~~20~~

```
#define N 100000000
bitset<N> isprime;

int main() {
  isprime.set();                    // i * i

  for(size_t i = 2; i < N; i++)
    if(isprime[i])                  /* if i is a prime */
      for(size_t j = 2i; j*i < N; j++)   // mark its multiple as non-prime
        isprime[j*i] = 0;

      for(size_t i = 2; i < N; i++)
        if(isprime[i])
          cout << i << endl;
}
```

If a positive integer n is composite, then it must have at least 1 factor <= √n

Mergesort

if size <= 1, do nothing

Otherwise, divide the vector into 2 halves, recursively sort that 2 halves + merge.

```
vector<int> mergesort(const vector<int>& v) {
  return v.size() <= 1 ? v : merge(mergesort(vector<int>(v.begin(), v.begin() + n/2)),
                                   mergesort(vector<int>(v.begin() + n/2, v.end())));
}

vector<int> merge(const vector<int>& v1, const vector<int>& v2);
```

```
3 2 7 6 8 5 5 3 6                 size: 9  size/2: 4
|         \          └ end()
begin()   begin() + 4
```

```
vector<int> merge(const vector<int>& v1, const vector<int>& v2) {
  vector<int>::size_type i1 = 0, i2 = 0, n1 = v1.size(), n2 = v2.size();
  vector<int> v;
  v.reserve(n1 + n2);

  while(i1 < n1 && i2 < n2)
    v.push_back(v1[i1] < v2[i2] ? v1[i1++] : v2[i2++]);
  v.insert(v.end(), v1.begin() + i1, v1.end());
  v.insert(v.end(), v2.begin() + i2, v2.end());
  return v;
}
```

Quicksort

[1, 2, 3] – list of 3 numbers

```
qsort([]) → [];
qsort([H|T]) → qsort([x || x ← T, X =< H])++ [H]++
        /   \       qsort([x || x ← T, X > H])
       1   [2, 3]                 |
                          list comprehension
{x | x ∈ T, x <= 3}
```

```
vector<int> operator +(const vector<int>& v1, const vector<int>& v2) {
  vector<int> v(v1);
  v.insert(v.end(), v2.begin(), v2.end());
  return v;
}
```

Pair

```
#include <utility>

template<typename S, typename T>
struct pair {
  S first;                                   /* this is basically the
  T second;                                     definition of pair in
  …                                             <utility> */
};

pair<int, double> p(3, 2.5);
  cout << p.first << endl;                   // 3
  cout << p.second << endl;                  // 2.5

pair<vector<int>, vector<int> >

partition(const vector<int>& v) {            // precondition: !v.empty()
  pair<vector<int>, vector<int> > p;
  vector<int>::size_type i, n = v.size();

  for(i = 1; i < n; i++)
    (v[i] <= v[0] ? p.first : p.second).push_back(v[i]);

  return p;
}

vector<int> qsort(const vector<int>& v) {
  if(v.size() <= 1)
    return v;

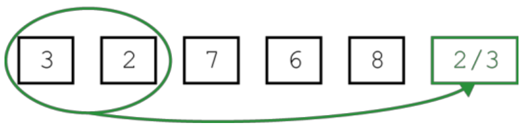
  pair<vector<int>, vector<int> > p = partition(v);

  return qsort(p.first) + vector<int>(1, v[0]) + qsort(p.second);
}
```

calls vector constructor

Non-Recursive Mergesort

3 2 7 6 8




```
vector<int> mergesort(const vector<int>& v) {        // precondition: !v.empty()
  deque<vector<int> > d;

  for(vector<int>::size_type i = 0; i < v.size(); i++)
    d.push_back(vector<int>(1, v[i]));

  while(d.size() > 1) {
    d.push_back(merge(d[0], d[1]));
    d.pop_front();
    d.pop_front();
  }

  return d[0];
}
```

eg. Deleting All Even Integers in a List of Integers

```
lst = list;

list<int>::iterator it;

for(it = lst.begin(); it != lst.end());
  if(*it % 2 == 0)
    it = lst.erase(it);      // erase returns an iterator to the following element
  else
    ++it;
```

Example

```
struct Pair {
  int x, y;
  Pair(int a, int b): x(a), y(b) {}
};

bool operator <(const Pair& p1, const Pair& p2) {
  return p1.x + p1.y < p2.x + p2.y;
}

Pair p1(1, 2), p2(1, 3), p3(-1, 4);
```

p1 < p2 is true
p1 ≮ p3 and p3 ≮ p1 // We regard them as equivalent

```
set<Pair> s;
s.insert(p1);               // OK
s.insert(p2);               // OK
s.insert(p3);               // not inserted, p3 is equivalent to p1
```

[Sets & Multisets, Maps & Multimaps – available in SHAREOUT]

Example: Sorting Words + Eliminating Duplicates

```
set<string> s;
string word;

while(cin >> word)
  s.insert(word);

set<string>::const_iterator it;

for(it = s.begin(); it != s.end(); ++it)
  cout << *it << endl;
```

Example: A Function Template to Print Containers

```
template<typename C>
void print(const C& c) {
  typename C::const_iterator, it;

  for(it = c.begin(); it != c.end(); ++it)
    cout << *it << endl;
}
```

tells compiler that C::const_iterator is the name of a type

```
examples:       ios_base::setw            - a value
                ios_base::fmtflags        - a type

                cout.setf(ios_base::setw, ios_base::basefield);
```

Basically when you have C::x where C is template parameter. We need to prefix it with typename if it refers to a name of a type.

- lower_bound and upper_bound:

  [Recall Pair]

  Define(x1, y1) < (x2, y2) if x1 + y1 < x2 + y2

```
multiset<Pair> s;
 s.insert(Pair(1, 2));            (1, 2), (1, 3), (4, 0), (1, 5)
 s.insert(Pair(1, 3));                    |             |  |
 s.insert(Pair(4, 0));                    p             q  r
 s.insert(Pair(1, 5));

p = lower_bound(Pair(2, 2));      // returns position not less than (2, 2)
q = lower_bound(Pair(2, 2));      // returns position not less than (2, 2)

r = lower_bound(Pair(2, 3));
s = lower_bound(Pair(2, 3));      // Note: r = s and hence there are no
                                     elements equivalent to (2, 3)
```

Example: Sorting Integers in Descending Order

```
   a class template (uses > for comparison by default)
                          |
multiset<int, greater<int> > s;
                           |
                         a type

greater<int> g;              // function object – there is a method operator()(int, int);

cout << g(1, 2) << endl;     // 0
cout << g(3, 2) << endl;     // 1
```

Example: Set of Students

```
struct Student {
 string id, firstname, lastname;

 Student(const string& i = "", const string& f = "", const string& l = "")
 : id(i), firstname(f), lastname(l){}
};
```

Need to implement operator < in order to put Students in a set/multiset

```
bool operator <(const Student& sl, const Student& s2) {
  return s1.id < s2.id;
}
```

```
set<Student> s;                         Student st;
```

```
set<Student> s;
string id, firstname, lastname;
```

```
                        cin >> st
while(cin >> id >> firstname >> lastname)
  s.insert(Student(id, firstname, lastname));

for(set<Student>::const_iterator it = s.begin(); it != s.end(); ++it)
  cout << it->id << " " << it->firstname << " "
      << it->lastname << endl;
                 cout << *it << endl;
```

(*p).x ≡ p-> x   Students sorted in ascending order of 10 and duplicates are eliminated

We need to implement operator << and operator >> for Students

```
ostream&
operator <<(ostream& os, const Student& s) {
  return os << s.id << " " << s.firstname << " " << s.lastname;
}
```

Student
/
cout << st ~~<< **endl**~~; ↝ **operator <<**(**cout**, st)

```
istream&
operator >>(istream& is, Student& s) {
  return is >> s.id >> s.firstname >> s.lastname;
}
```

Note: This version does not perform validation

cin >> st; ↝ **operator >>**(**cin**, st)

We can then do the following:

```
set<Student> s;
Student st;

while(cin >> st)
  s.insert(st);
```

calls default constructor (constructor that can be called without arguments)

```
for(set<Student>::const_iterator it = s.begin();        // we want to think of this as copying
  it != s.end(); ++it)                                    a range to a destination
  cout << *it << endl;
```

What **if** we need a d**iff**erent sorting order?

```
set<Student, Cmp> s;
```

name of class and it specifies the sorting order.
A Cmp object can be used to compare 2 student
(it is a function object)

A function object(functor) is an object that acts like a function. This is achieved by overloading the function call operator, operator ()

```
struct NameCmp {       // a class of function objects
  bool operator()(const Student& s1, const Student& s2) const {
    if(s1.lastname != s2.lastname)
      return s1.lastname < s2.lastname;
    return s1.firstname < s2.firstname;
  }
}
```

```
NameCmp f;    // calls default constructor (automatically generated by the compiler in this case)
```

Assume s1 + s2 are 2 student objects

```
  f(s1, s2) ↝ f.operator()(s1, s2)
```

```
(NameCmp())(s1, s2)
```

this creates a temporary NameCmp object

```
set<Student, NameCmp> s;    // this set sorts by name
```

Maps

A map<Key, Value> stores pairs<**const** Key, Value>

eg. `map<string, int> m;`

An object in m has type pair<**const string**, **int**>

make_pair<U, V> is a function template that returns a pair<U, V>

make_pair<1, 2> return a pair<**int**, **int**>

```
pair<int, int> p = make_pair(1, 2);
pair<double, double> q = make_pair<double, double>(1, 2)
```

Example: Word Frequencies

```
map<string, int> count;
string word;

while(cin >> word)
  count[word]++;

map<string, int>::const_iterator it;
for(it = count.begin(); it != count.end(); ++it)
  cout << it->first << ":" << it->second << endl;
```

For a map

```
                                  key type      value type
                                           \   /
operator[](k)   (*(insert(make_pair<K, T>(K, T()).first).second))
                       |                               |
m[3]            tries to insert a pair with     default constructor
                the default value before
                returning the value
```

Generic Algorithms

Idea: If we have an algorithm that processes arrays, we can make it into a generic algorithms

We have already ‘find’

```
template<typename InputIterator, typename T>
InputIterator
find(InputIterator first, InputIterator last, const T& t) {
  while(first != last) {
    if(f(*first))
      break;
    ++first;
  }
  return first;
}
```

We can generalize this

For Arrays:

```
const int *find_if(const int *first, const int *last, bool(*f)(int)) {
  while(first != last) {
    if(f(*first))
      break;
    ++first;
  }
  return first;
}
```

```
template<typename InputIterator, typename Predicate>
InputIterator
find_if(InputIterator first, InputIterator last, Predicate p) {
  while(first != last) {
    if(p(*first))
      break;
    ++first;
  }
  return first;
}
```

Copying

```
void copy(const int *first, const int *last, int *result) {
  while(first != last)
    *result++ = *first++;
}
```

```
template<typename InputIterator, typename OutputIterator>
OutputIterator
copy(InputIterator first, InputIterator last, OutputIterator result) {
  while(first != last)
    *result++ = *first++;
  return result;
}
```

Note: Copy overwrites the destination. We have to ensure that the destination is large enough to store the source.

```
int a[] = {1, 2, 3, 4, 5};
vector<int> v(a, a + 5);
list<int> list(a, a+ 5);
int b[] = {10, 9, 8, 7, 6, 5, 4, 3, 2, 1};
deque<int> d(b, b + 10);
```

```
it = copy(v.begin(), v.end(), d.begin());        // 1, 2, 3, 4, 5, 5, 4, 3, 2, 1

copy(list.begin(), list.end(), it);              // 1, 2, 3, 4, 5, 1, 2, 3, 4, 5
```

How to use copy to print objects?

associate an Iterator with an ostream object

ostream_iterator   #include<iterator>

```
ostream_iterator<int> it(cout, "\n");
                  |          |
           type of object   separator
```

```
copy(v.begin(), v.end(), it);                    // prints all integers in vector<int> v
```

OR

```
copy(v.begin(), v.end(), ostream_iterator<int>(cout, "\n"));
```

Recall: OutputIterator:

single-pass

it++, ++it, *it = x;

By suitably defining operator++(), operator++(int), operator*()

| |
prefix postfix

operator*(...) we can print to an ostream

The two ++ and * do basically nothing = calls << to print object

Possible Implementation:

```
template<typename T>
class Ostream_Iterator {
private:
  ostream *pos_;
  const char *s_;
public:
  ostream_iterator(ostream& os, const char *s = 0): pos_(&os), s_(s) {}

                                              current object
                                                    /
  ostream_iterator operator ++() { return *this; }

  ostream_iterator operator ++(int) { return *this; }

  ostream_iterator operator *() { return *this; }

  ostream_iterator& operator = (const T& t) {
    *pos_ << t;
    if(s_)
      *pos_ << s_;
    return *this;
  }
}
```

```
eg.     ostream_iterator<int> it(cout, "\n");
         it   = 2;        // print 2
        *it   = 3;        // print 3
        *it++ = 4;        // print 4
```

istream_iterator
#include <iterator>

eg.
```
istream_iterator<int> in(cin), eof;      // corresponds to end-of-file

vector<int> v(in, eof);                  // keeps reading integers from cin and
                                            stores them in v

vector<int> v(istream_iterator<int>(cin), istream_iterator<int>());
```

possible interpretation: an input_iterator with parameter name cin

possible interpretation: function that takes nothing and returns an input_iterator

function prototype:
v is a function that takes 2 arguments + returns a vector<int>

```
voidf(int n);
voidf(int (n));          // OK
voidf((int (n)));        // not OK
voidf((int n));          // not OK

vector<int> v((istream_iterator<int>(cin)), istream_iterator<int>());
```

- reads ints from stdin and store them in the vector
- no error-handling: may get into infinite loop if input is not an integer

eg. Sorting Words

```
multiset<string> s((istream_iterator<string>(cin)),
  istream_iterator<string>());

copy(s.begin(), s.end(), ostream_iterator<string>(cout, "\n"));
```

- So far, copy overwrites the destination. But by using special iterators, we can make it "insert" into a container.

```
copy:  while(first != last)
           *result++ = *first++;
```

By suitably defining *, **, =, we can make this into an insertion:

*, ++ do nothing
= calls push_back

back_insert_iterator – class template parameterized by the type of container

eg.

```
vector<int> v;
// add some ints to v
deque<int> d;
// add ints to d
copy(v.begin(), v.end(), back_insert_iterator<deque<int> >(d));
```

back_inserter – function template that takes a container and returns a back_insert_iterator

```
copy(v.begin(), v.end(), back_inserter(d));
```

We'll implement a version of back_insert_iterator:

```
template<typename C>
class Back_insert_iterator {
private:
  C *pc_;
public:
  Back_insert_iterator(C& c): pc_(&c) {}
  Back_insert_iterator& operator *() { return *this; }
  Back_insert_iterator& operator++() { return *this; }
  Back_insert_iterator& operator++(int) { return *this; }
  Back_insert_iterator& operator=(const typename C::value_type& x) {
    pc->push_back(x);                                   \
    return *this;                        typename C::const_reference
  }
};
```

```
template<typename C>
inline Back_insert_iterator<C>
Back_insert(C& c) {
  return Back_insert_iterator<C>(c);
}
```

- It's fairly standard to provide function template that returns an object from a class template
- A function template can deduce from its argument what the template parameter should be
- There is also a front_insert_iterator

All the standard container classes have standard typedefs (for associated types):

- size_type
- iterator
- const_iterator
- value_type: type of objects stored in the container
- pointer: typically value_type*
- const_pointer: typically const value_type*
- reference: typically value_type&
- const_reference: typically const value_type&
- difference_type: signed integer type for meaning the distance between two iterators

eg. Finding the Maximum in a Container

```
template<typename C>
typename C::value_type

max(const C& c) { // precondition: c.size() > 0
  typename C::value_type largest = *c.begin();

  for(typename C::const_iterator it = c.begin(); it != c.end(); ++it)
    if(*it > largest)
      largest = *it;

  return largest;
}
```

eg.

```
vector<int> v;
   ...
cout << max(v) << endl;
```

eg. vector<int>::value_type is int

map<Key, T>::value_type is pair<const Key, T>
map<Key, T>::key_type is Key
map<Key, T>::mapped_type is T

Function Objects

- objects from class that overloads the function call operator, operator()

Example: Fibonacci Sequence

C:

```
int fib() {                        // 1, 1, 2, 3, 5, 8, 13, ...
  static int a = 1, b = 1;
  int result = a;                  fib(); ↝ 1
  a = b;                           fib(); ↝ 1
  b += result;                     fib(); ↝ 2
  return result;                   ...
}
```

What if we want two independent Fibonacci sequences? Implement another fib() function!!

C++:

```
class Fib {
private:
  int a_, b_;
public:
  Fib(): a_(1), b_(1) {}

  int operator()() {
    int result = a_;

    a_ = b_;
    b_ += result;

    return result;
  }
}
```

eg.

```
Fib f1, f2;

f1() ↝ 1
f2() ↝ 1
f1() ↝ 2
f2() ↝ 1
...
```

More Algorithms

```
#include<algorithm>
```

- for_each: applies a function to a range

possible implementation:

```
template<typename InputIterator, typename UnaryFunction>

UnaryFunction
for_each(InputIterator first, InputIterator last, UnaryFunction f) {
  while(first != last)
    f(*first++);
  return f;
}
```

eg.

```
vector<int> v;
// code to add some ints
for_each(v.begin(), v.end(), print);
```

```
void print(int n) {
  cout << n << endl;
}
```

eg. Collecting Statistics

eg. Find the Average of the Integers in v

```
struct Stats {
  size_t count;
  int total;

  Stats(): count(0), total(0) {}

  void operator()(int n) {
    count++;
    total += n;
  }
};
                                   calls default constructor to
                                   create a temporary stats object
                                     /
Stats s = for_each(v.begin(), v.end(), Stats());

if(s.count != 0)
  cout << (double)s.total / s.count << endl;
```

- find_if

eg. v – vector<int>

find first even number in v:

```
vector<int>::iterator it = find_if(v.begin(), v.end(), iseven);

bool iseven(int n) {  return n % 2 == 0; }
```

What if we need to find the first number divisible by 3? by 7?

Use function objects:

```
struct is_divisible_by {
  int divisor;

  is_divisible_by(int d): divisor(d) {}

  bool operator()(int n) {
    return n % divisor == 0;
  }
}
```

eg.

```
vector<int>::iterator it = find_if(v.begin(), v.end(),is_divisible_by(3));
```

What if we need to deal with long ints?

Make is_divisible_by into a class **template**!!

```
template<typename T>
struct is_divisible_by {
  T divisor;

  is_divisible_by(T d): divisor(d) {}

  bool operator()(T n) {
    return n % divisor == 0;
  }
};
```

lst – list<long>

```
list<long>::iterator it = find_if(lst.begin(), lst.end(),is_divisible_by<long>(3));

template<typename T>
inline is_divisible_by<T>
divisible_by(T d) {
  return is_divisible_by<T>(d);
}
```

eg.

```
list<long>::iterator it = find_if(lst.begin(), lst.end(), divisible_by(3L));
```

What if we want to find the first number NOT divisible by 3??

Idea: wrap a is_divisible_by object in another object that negates its meaning.

function object adapter

The standard library provides the unary_negate class template

eg.

```
unary_negate< is_divisible_by<long> > (is_divisible_by<long>(3))
```

class name — divisible_by(3L)

There is also a function template, not1, that returns a unary_negate object.

eg.

```
not1(divisible_by(3L))
```

Note: We need to modify is_divisible_by to make(*) work.

We can implement a version of unary_negate as follows:

```
template<typename Predicate>
class Unary_negate {
private:
  Predicate f_;
public:
  Unary_negate(const Predicate& f): f_(f) {}

  bool operator()(const typename Predicate::argument_type& x) {
    return !f_(x);
  }
};
```

- Use standard names for the type of the argument and the type of the return value:

  argument_type
  result_type

  We need to modify is_divisible_by (we need to add two typedefs):

```
template<typename T>
class is_divisible_by {
private:
  T divisor_;
public:
  typedef T argument_type;       with these two typedefs, we can use
  typedef bool result_type;      unary_negate with this class
  ... // same as before
};
```

functor class – class of function objects

A functor class that has the standard typedef is called an ADAPTABLE functor class. (Objects from such a class can be used with function object adaptors).

To make it simpler to define the typedefs, the standard library provides a unary_function class template that we can inherit from.

```
                                           argument type    result_type
template<typename T>                                  |        /
class is_divisible_by: public unary_function<T, bool> {
                 |
         inheritence
  // same as before (without the two typedefs)
};
```

unary_function is basically as follows:

```
template<typename Arg, typename Result>
struct unary_function {
  typedef Arg argument_type;
  typedef Result result_type;
};
```

Function that returns a Unary_negate object:

```
template<typename Predicate>
inline Unary_negate<Predicate>
Not1(const Predicate& f) {
  return Unary_negate<Predicate>(f);
}
```

- functor class: class of function objects
- function object adaptor: unary_negate
- adaptable functor class: class of function objects that work with function object adaptors. This means that class has certain standard typedefs:
  - for unary functions – argument_type, result_type
  - for binary functions – first_argument_type, second_argument_type, result_type

  The typedefs can be made by inheriting from unary_function and binary function.

- We've already seen unary_negate/not1. There is also binary_negate/not2.
- Recall: greater<T>(x, y) ↦ x > y   set<int, greater<int> >s;

- binary_negate<greater<int> >(greater<int>())

  |

  calls constructor to create a

  not2(greater<int>()) temporary greater<int> object

Some predefined functor classes:

#include <functional>

arithmetic:

plus<T>	(x, y) ↦ x + y
minus<T>	(x, y) ↦ x - y
multiplies<T>	(x, y) ↦ x * y
divides<T>	(x, y) ↦ x / y
modulus<T>	(x, y) ↦ x % y
negate<T>	(x, y) ↦ -x

comparisons:

less<T>	(x, y) ↦ x < y
greater<T>	(x, y) ↦ x > y
less_equal<T>	(x, y) ↦ x <= y
greater_equal<T>	(x, y) ↦ x >= y
equal_to<T>	(x, y) ↦ x == y
not_equal_to<T>	(x, y) ↦ x != y

cout << plus<int>()(1, 2) << **endl**; // 3

plus<int> p;

p(3, 5) returns 8 function template

/ \

bind1st bind2nd

two other adapters classes: binder1st & binder2nd (for binary functions)

/ \

fix 1st arugment fix 2nd argument

to a specific value to a specific value

adding 10:

binder1st<plus<int> >(plus<int>(); 10) – creates temporary unary function

x ↦ 10 + x

bind1st(plus<int>(), 10)

We’ll implement a version of binder1st:

```
template<typename BinaryFunction>
class Binder1st
: public unary_function<typename BinaryFunction::second_argument_type,
                        typename BinaryFunction::result_type> {
private:
  BinaryFunction f_;
  typename BinaryFunction::first_argument_type x_;
public:
  Binder1st(const BinaryFunction& f,
            const typename BinaryFunction::first_argument_type& x)
       : f_(f), x_(x){}

  typename BinaryFunction::result_type
  Operator()(const typename BinaryFunction::second_argument_type& x) {
    return f_(x_, x);
  }
};

template<typename BinaryFunction>
inline Binder1st<BinaryFunction>
Bind1st<const BinaryFunction& f, const typename::first_argument_type& x> {
  return Binder1st<BinaryFunction>(f, x);
}
```

unary_negate/not1, binary_negate/not2,
binder1st/bind1st and binder2nd/bind2nd are in the standard C++ library

We’ll implement two adapters to compose functions (not in standard C++ library):

f1(f2(x))   (f1 · f2)(x)

```
template<typename Fun1, typename Fun2>              type of argument
class unary_compose                                 /
  : public unary_function<typename Fun2::argument_type,
                          typename Fun1::result_type> {
private:                                            \
  Fun1 f1_;                                         typename of return value
  Fun2 f2_;
public:
  unary_compose(const Fun1& f1, const Fun2& f2)
  : f1_(f1), f2_(f2) {}
  typename Fun1::result_type
  operator()(const typename Fun2::argument_type& x) {
    return f1_(f2_(x));
  }
};
```

```
template<typename Fun1, typename Fun 2>
inline unary_compose<Fun1, Fun2>
compose1(const Fun1& f1, const Fun2& f2) {
  return unary_compose<Fun1, Fun2>(f1, f2);
}
```

eg. find_if

look for first number in a vector that is divisible by 3

```
modulus<int>(): (x, y) ↦ x % y
bind2nd(modulus<int>(), 3): x ↦  x % 3
equal_to<int>(): (x, y) ↦ x == y
bind2nd(equal_to<int>(), 0): x ↦ x == 0;
compose1(bind2nd(equal_to<int>(), 0), bind2nd(modulus<int>(), 3)): x ↦ (x % 3) == 0
vector<int> v;
... // add ints to v
vector<int>::iterator it = find_if(v.begin(), v.end(),
                                   compose1(bind2nd(equal_to<int>(), 0),
                                            bind2nd(modulus<int>(), 3)));
                                        /
            What is the type of the object?? f = ...

unary_compose<binder2nd<equal_to<int> >, binder2nd<modulus<int> > >

cout << f(3) << endl;  // 1
cout << f(4) << endl;  // 0
```

binary_compose:

x ↦ f1(f2(x), f(x))

```
template<typename Fun1, typename Fun2, typename Fun3>
class binary_compose
  : public unary_function<typename Fun2::argument_type, typename Fun1::result_type> {
private:
  Fun1 f1_;
  Fun2 f2_;
  Fun3 f3_;
public:
  binary_compose(const Fun1& f1, const Fun2& f2, const Fun3& f3)
    : f1_(f1), f2_(f2), f3_(f3) {}
    typename Fun1::result_type
    operator()(const typename Fun2::argument_type& x) {
      return f1(f2_(x), f3_(x));
    }
};
```

```
template<typename Fun1, typename Fun2, typename Fun3>
inline binary_compose<Fun1, Fun2, Fun3>
compose2(const Fun1& f1, const Fun2& f2, const Fun3& f3) {
  return binary_compose<Fun1, Fun2, Fun3>(f1, f2, f3);
}
```

Algorithms

- find/find_if/copy

- find_first_of

```
template<typename InputIterator, typename ForwardIterator>
InputIterator
find_first_of(InputIterator, InputIterator, ForwardIterator, ForwardIterator);
```

eg. look for 2.3 or 7 in a vector<int>

```
vector<int> v;
int a[] = {2, 3, 7}

vector<int>::iterator it = find_first_of(v.begin(), v.end(), a, a + 3);
```

eg. look for newline, tab or space in a C-style string

```
char *s = "hello world";
char *t = "␣\t\n";
char *p = find_first_of(s, s + strlen(s), t, t + strlend(t));
```

A string object is also a container

```
string s("hello␣world");
string t("␣\t\n");
string::iterator it = find_first_of(s.begin(), s.end(), t.begin(), t.end());
```

Note: The string class already has a find_first_of method

- transform – 2 versions

1.
```
template<typename InputIterator, typename OutputIterator, typename UnaryFunction f>
{
  OutputIterator
  transform(InputIterator first, InputIterator last, OutputIterator result, UnaryFunction f)
  {
    while(first != last)
      *result++ = f(*first++);
    return result;
  }
}
```

Note: Result is allowed to be the same as first transform typically overwrites the destination

eg.
```
vector<int> v;
...
transform(v.begin(), v.end(), back_inserter(lst), bind1st(multiplies<int>(), 2));

list<int>lst; // lst is empty

transform(v.begin(), v.end(), back_insert(lst), bind1st(multiplies<int>(), 2));
```

2.
```
template<typename InputIterator1, typename InputIterator2,
         typename OutputIterator, typename BinaryFunction>
OutputIterator
transform(InputIterator1 first1, InputIterator1 last1, InputIterator2 first2,
          OutputIterator result, BinaryFunction f) {
  while(first1 != last1)
    *result++ = f(*first1++, *first2++);
  return result;
}
```

eg.
```
vector<int> v;
list<int> lst;
// assume we have stored 100 ints in both v and lst
deque<int> d; // empty
transform (v.begin(), v.end(), lst.begin(), back_inserter(d), plus<int>());
```

- sort – 2 versions

1. takes two random access iterator

eg.
```
vector<int> v;
...
sort(v.begin(), v.end());        // sort in ascending order
```

2. takes two random access iterator and a comparison function

eg.
```
sort(v.begin(), v.end(), greater<int>()); // sort in descending order
```

How to sort a list? The list class already has a sort method

```
list<int> lst;
...
lst.sort();                      // ascending order
lst.sort(cmp);
            |
comparison function
```

Iterators

eg. function to return the maximum value within a non-empty range

```
                                                iterator_trait<InputIterator>::value_types
template<typename InputIterator> /
typename InputIterator::value_type
max_nonempty(InputIterator first, InputIterator last) {
 typename InputIterator::value_type max = *first++;

 while() {                      \
  if(*first > max)              iterator_traits<InputIterator>::value_types
   max = *frist;
  ++first;
 }
 return max;
}
```

- iterators have associated types

```
int a[] = {3, 2, 7, 6, 8};
vector<int> v(a, a + 5);

cout << max_nonempty(v.begin(), v.end()) << endl;  // 8
cout << max_nonempty(a, a + 5) << endl;            // doesn't compile because there is
                                                      no value_type associated with a
                                                      pointer
```

Use iterator_traits

```
template<typename Iterator>
struct iterator_traits {
 typedef typename Iterator::value_type value_type;
 ...
};
```

iterator_traits<Iterator>::value_type ≡ Iterator

C++ supports both full and partial specialization of class template

eg.

```
template<typename T>
class C { … };                    C<int> c;

template<>
class C<int> { … };               (full) specialization of C for type int

template<typename T>
class C<T*> { … };                (partial) specialization of C for pointer types
```

partially specialize iterator_traits

```
template<typename T>
struct iterator_traits<T*> {      const int *p;
  typedef T value_type;
  …
};

template<typename T>
struct iterator_traits<const T*> {
  typedef T value_type;
  …
};
```

Standard associated types of iterators:

value_type	type of object pointed by the iterator
difference_type	distance between two iterators eg. used in the count algorithm
pointer	typically value_type*
reference	typically value_type&
iterator_category	

```
template<typename InputIterator, typename T>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& x) {
  typename iterator_trait<InputIterator>::difference_type n = 0;

  while(first != last)
    if(*first++ = x)
      n++;
  return n++;
}
```

```
eg.     advance(it, 10)
               /   \
         iterator   distance

        template<typename InputIterator, typename Distance> a type
        void                                                  /
        advance(InputIterator it, Distance n, input_iterator_tag) {
          while(n--)
            it++;
        }
        ...     // versions for ForwardIterator, Bidirection Iterator

template<typename RandomAccessIterator, typename Distance>
void
advance(RandomAccessIterator& it, Distance n, random_access_iterator_tag) {
  it + n;                                                \
}                                                        a type
```

The Standard C++ Library 5 Iterator Tags:

```
input_iterator_tag
output_iterator_tag
forward_iterator_tag
bidirectional_iterator_tag
random_access_iterator_tag
```

They are empty classes eg.

```
struct input_iterator_tag {};

template<typename Iterator, typename Distance>
void
advance(Iterator& it, Distance n) {
  advance(it, n, typename iterator_traits<Iterator>::iterator_category());
}
```

Classes

Four standard member functions:

1. default constructor – constructor that can be called without arguments
2. copy constructor – C(const C&);
3. destructor - ~C();
4. assignment operator – C operator = (**const** C&);

- If you don't have a copy constructor or destructor or assignment operator, the compiler declares/generates one for you.
- If you don't have any constructor at all, the compiler declares/generates one for you.
- Disregarding inheritance, the compiler-generated default constructor/copy constructor/assignment operator calls the default constructor/copy constructor/assignment operator for each data member in the order that the data members are listed in the class definition. For data members that are primitive types, the default constructor does nothing, the copy constructor perform bitwise copy and the assignment operator performs regular assignment.

Example:

```
class C { … };
class D { … };
class E {
private:
  C c_;
  D d_;
  int x_;
};
```

compiler-generated copy constructor
class C's copy constructor for c_, then
class D's copy constructor for d_, then
performs bitwise copy for x_

another E object

```
E(const E& src): c_(src.c_), d_(src.d_), x_(src.x_) {}
```

calls copy constructor of C

```
E e1;
E e2(e1);
```

compiler_generated assignment operator

- calls assignment operator for c_ (assignment operator of class C) then
  calls assignment operator for d_ (assignment operator of class D) then
  performs regular assignment of integers for x
  Similarly for default constructor except that x_ has a random value
- The compiler destructor is basically this: ~C(){}

Note: This already automatically calls the destructor for the data members. In general, we don't need to explicitly call destructors.

Note: Order of destructor calls is the reverse of the order of constructor calls.

```
E e;                    // calls default constructor
E *p = new E[2];        // calls default constructor twice
E e2(e);                // calls copy constructor
E e3 = e2;              // also calls copy constructor
e = e2;                 // calls assignment operator
delete[] p;             // calls destructor twice
// destructor called when the other objects goes out of scope
```

Implementing a Simple Class

The Name Class

```
// Name.h
#ifndef NAME_H
#define NAME_H
#include <string>
#include <iostream>

// do not put: using namespace std;

class Name {
private:
  std::string last_, first_;
public:
  explicit Name(const std::string& first = "john", const std::string& last = "doe")
  : first_(first), last_(last) {
    if(!isValidFirstName(first_) || (!isValidLastName(last_))
      throw "Name::Name(const string&, const string&): ...";
  }
}

std::string getFirstName() const { return first_; }
std::string getLastName()  const { return last_; }
bool setFirstName(const std::string& first) {
  if(!isValidFirstName(first))
    return false;
  first_ = first;
  return true;
}
```

```
// setLastName ...

// friend declarations
friend std::ostream& operator <<(std::ostream&, const Name&);
friend std::istream& operator >>(std::istream&, Name&);
friend bool operator == (const Name&, const Name&);
friend bool operator < (const Name&, const Name&);
...

static bool isValidFirstName(const std::string& first) {
  ...
}

// similarly for isValidLastName

inline std::ostream&                        // Note:  inline and no friend keyword
operator <<(std::ostream& os, const Name& n) {
  return os << n.first_ << ' ' << n.last_;  // can directly access first_ and last_ because
}                                              it is a friend


inline std::istream&
operator >>(std::istream& is, Name& n) {
  std::string first, last;                  // need to prefix with Name as operator >> is not
                                               a member of Name

  if(is >> first >> last && Name::isValidFirstName(first) && Name::isValidLastName(last))
    n.first_ = first; n.last_ = last;
  else
    is.setstate(ios_base::faitbit);
  return is;
}

/* If we don't need to do input validation, we can simply do this:

inline std::istream&
operator >>(std::istream& is, Name& n) {
  return is >> n.first_ >> n.last_;
}
*/

inline bool
operator ==(const Name& lhs, const Name& rhs) {
  return lhs.first_ = rhs.first_ && lhs.last_ == rhs.last_;
}
```

```
inline bool
operator < (const Name& lhs, const Name &rhs) {
  if(lhs.last_ != rhs.last_)
    return lhs.last_ < rhs.last_;
  return lhs.first_ < rhs.first_;
}

inline bool
operator != (const Name& lhs, const Name &rhs) {
  return !(lhs == rhs);
}

inline bool
operator <= (const Name& lhs, const Name &rhs) {
  return lhs < rhs || lhs = rhs;
}

inline bool
operator >= (const Name& lhs, const Name &rhs) {
  return !(lhs < rhs);
}

inline bool
operator > (const Name& lhs, const Name &rhs) {
  return !(lhs <= rhs);
}

#endif
```

eg.

```
Set<Name> s;  // needs <
Name n;
while(cin >> n)
  s.insert(n);
copy(s.begin(), s.end(), ostream_iterator<Name>(cout, "\n"));
                                   // needs <<
```

```
sort(v.begin(), v.end());          // needs <
sort(v.begin(), v.end(), (mp));
```

The Employee Class

```
// Employee.h

#ifndef EMPLOYEE_H
#define EMPLOYEE_H
#include <string>
#include <iostream>
#include “Name.h”
#include “Date.h”

class Employee {                    Employee e;   // calls default constructor
private:
  std::string id_;                  cin >> e;
  Name name_;
  Date birthdate_;
public:
  explicit Employee(const std::string& id == “A00000000”, const Name& name == Name(),
                    const Date& birthdate == Date());

  // compiler-generated copy constructor and assignment
  virtual ~Employee(){}   // needs to be virtual because we are going to inherit from this class

  // get and set methods
  set::string getId() const { return id_; }
  ...
  bool setId(const std::string& id);

  // static methods
  static bool isValidId(const std::string& id);

  // friend declarations
  friend std::ostream& operator <<(std::ostream&, const Employee&);
  friend std::istream& operator >>(std::istream&, Employee&);
  friend bool operator ==(const Employee&, const Employee&);
  friend bool operator <(const Employee&, const Employee&);
  // ... <=, >=, >, !=
};

#endif
```

explicit – any constructor that takes one argument can be used for conversion. Unless the constructor is declared “explicit,” the compiler can automatically call such a constructor for conversion.

```
class C {
private:
  int n_;
public:
  C(int n): n_(n) {}
  friend bool operator ==(const C& c1, const C& c2);
};
```

```
eg.     C c1(1), c2(2);

        if(c1 == c2)...                 // OK
        if(c1 == 1)...
          /    \
         a C      an int                // compiler calls constructor to create
        object                             C(1) and compare it to C1
```

```
// Employee.cpp
...

#include "Employee.h"
using namespace std;

// need to prefix with class name; Note: no explicit keyword and  no default arguments

Employee::Employee(const string& id, const Name& name, const Date& birthdate): id_(id),
name_(name), birthdate_(birthdate) {
  if(!isValidId(id_))
    throw "Employee::Employee(const string&, const Name&, const Date&)";

  // additional validation if necessary
}

bool
Employee::setId(const string& id) {
  if(!isValidId(id))
    return false;
  id_ = id;
  return true;
}

bool // Note:  no static keyword
Employee::isValidId(const string& id) {
  if(id.size() != 9)
    return false;
```

```
  if(id[0] != 'A')
    return false;

  for(string::size_type i = 1; i < id.size(); i++)
    if(!isdigit(id[i]))
      return false;
}

ostream&         // Note: No friend keyword; not a member of Employee class
operator <<(ostream& os, const Employee& e) {
  os << e.id_ << e.name_ << e.birthdate_;
  return os;
};

istream&
operator >>(istream& is, Employee& e) {
  string id;
  Name name;
  Date birthdate;

                    need this because operator >> is not a member of Employee
                                                   |
if(is >> id >> name >> birthdate && Employee::isValidId(id)) {
    e.id_ = id;                               ┐
    e.name_ = name;                           |   or e = Employee(id, name, birthdate);
    e.birthdate_ = birthdate;                 ┘
  }
  else
    is.setstate(ios_base::failbit);
}

bool
operator ==(const Employee& lhs, const Employee& rhs) {
          return lhs.id_ == rhs.id_;
}
...
```

Inheritance

C++ has 3 types of inheritance: public, protected, private
C++ has 3 access specifies: public, protected, private

```
class C {
public:
  // interface for everyone
protected:
// interface for derived classes
private:
// interface for current class
};
```

inheritance	public	protected	private
access specifiers			
public	public	protected	private
protected	protected	protected	private
private	not accessible	n/a	n/a

- Everything in the base class is inherited in the derived class but may be hidden or inaccessible.

  ```
  class B {
  public:
    void f();
  protected:
    void g();
  private:
    void h();
  };

  class D: protected B {
  public:
    void f1() { g(); }        // OK, g is protected in D
    void g1() { h(); }        // invalid, h is private in B
  };

  D d;            B b;
  d.f1();                   // OK
  d.g();                    // invalid, g is protected in D
  d.f();                    // invalid, f is protected in D
  b.f();                    // OK, f is public in B
  ```

- public inheritance models the “is-a-kind-of” relationship
  private inheritance is for implementation inheritance, eg. stack inherits privately from vector

  ```
  class stack: private vector {...}
  ```

- Alternative: Use composition instead of private inheritance

  ```
  class Stack {
  private:
    vector<int> v_;
    ...
  };
  ```

Public Inheritance

- a base reference can refer to a derived object

- a base pointer can point to a derived object

  ```
  class B { ... };
  class D { ... };

  B b; D d;
  B& r  = b;              // OK
  B& r2 = d;              // OK under public inheritance
  B *p  = &b;             // OK
  B *p2 = &d;             // OK

  class B {
  public:
    void f();
    virtual void g();
  };

  class D: public B {
  public:
    void f();
    virtual void g();
  };

  D d;
  B& r = d;
  r.f();                  // which f is called B::f() (early binding)
  r.g();                  // virtual is needed for late binding
  B *p = &d;              // a->b ≡ (*a).b
  p->g();                 // calls D::g() late binding
  ```

- early binding – the compiler determines which method is called when it is compiling the program

- late binding – the compiler generates code that determines at run time which method is actually called

- In C++, late binding is only used when virtual method is invoked through a pointer or a reference

```
                                static type: B*
                                dynamic type: D*
D d;                                 /
B& r = d;                       B *p = &d;                  static type – declared type
    |
static type: B&(same as B)
```

```
// FulltimeEmployee.h ---- include guards!
#include <string>
#include <iostream>
#include "Name.h"
#include "Date.h"
#include "Employee.h"

class FulltimeEmployee: public Employee {
private:
  float salary_;
  static size_t count_;            // declaration only; need to define it in .cpp
public:
  explicit FulltimeEmployee(const std::string& id = "A00000000",
                            const Name& name = Name(),
                            const Date& birthdate = Date(),
                            float salary = 0);

FulltimeEmployee(const FulltimeEmployee& src);
virtual ~FulltimeEmployee() { count_--; }

// get and set methods
float getSalary() const { return salary_; }
bool setSalary(float salary) {
  if(salary < 0)
    return false;
  salary_ = salary;
  return true;
}

static size_t getCount() { return count_; }      // Note: can't declare as const
```

```cpp
friend std::ostream& operator <<(std::ostream& os, const FulltimeEmployee& e);
friend std::istream& operator >>(std::istream& is, FulltimeEmployee& e);
...

virtual void print(std::ostream& os) const;
};
```

```cpp
// FulltimeEmployee.cpp
...
#include "FulltimeEmployee.h"
using namespace std;
size_t FulltimeEmployee::count_ = 0;          // definition of count_;

FulltimeEmployee::FulltimeEmployee(const string& id, const Name& name, const Date&
birthdate, float salary): Employee(id, name, birthdate), salary_(salary) {
  if(salary_ < 0)
    throw "FulltimeEmployee::FulltimeEmployee(...): ...";

  count_++;
}

FulltimeEmployee::FulltimeEmployee(const FulltimeEmployee& src): Employee(src),
salary_(src.salary_) {

  // no need for validation
  count_++;
}
```

FulltimeEmployee; can we pass a FulltimeEmployee to an Employee constructor? Yes, calls Employee copy constructor (a base reference can refer to a derived object under public inheritance)

```cpp
void     // Note:  A virtual keyword
FulltimeEmployee::print(ostream& os) const {  // Assume there is a print in Employee class
  Employee::print(os);                        // call base print method
  os << "salary: " << fixed << setprecision(2) << salary_ << endl;
}
```

```
// Note:  A better way to print to an ostringstream

ostream&
operator <<(ostream& os, const FulltimeEmployee& e) {
  os << (const Employee&)e << ' ' << e.salary_;          // note cast!
  return os;
}

istream&
operator >>(istream& is, FulltimeEmployee& e) {
  float salary;

  if(is >> (Employee&)e >> salary && salary >= 0)       // note cast!
    e.salary_ = salary;
  else
    is.setstate(ios_base::failbit);
  return is;
}
```

- derived class constructor should call the corresponding base class constructor
- Alternative to implementing operators for each derived class:

  Implement operator << only for the base class and have it call a virtual function

  ```
  ostream& operator << (ostream& os, const Employee& e) {
    e.print(os);
    return os;
  }
  ```

- Override print in each derived class

Abstract Base Class (ABC)

An abstract class is a class with at least 1 pure virtual function. An abstract class cannot be instantiated. The implementation of a pure virtual function is optional.

The Shape Class

```
class Shape {
private:
  int colour_;

public:
  explicit Shape(int colour = 0): colour_(colour) {}
  virtual void draw(GC& gc) const = 0;                // pure virtual
                          └ graphics context

  virtual void save(ostream& os) const = 0;           // pure virtual

  explicit Shape(istream& is) { is >> colour_; }
};

// The methods in the abstract base class contain common code
// (common to all derived classes)

inline void
Shape::draw(GC& gc) const {
  // code to set the colour
}

inline void
Shape::save(ostream& os) const {
  os << colour_ << endl;
}
```

- commonality/variability analysis

```cpp
// Circle.h
...                                          typedef pair<int, int> Point;
#include "Shape.h"

class Circle: public Shape {                 // a concrete class
private:
  Point centre_;
  int radius_;

public:
  explicit Circle(int colour, const Point& centre = Point(), int radius = 1)
    : Shape(colour), centre_(centre), radius_(radius) { ... }
                                           validation code ┘

  virtual void draw(GC& gc) const {          // call base draw method
    // code to draw a circle
  }

  virtual void save(ostream& os) const {
    os << "Circle" << endl;
    Shape::save(os);
  }

  explicit Circle(istream& is): Shape(is) {
    is >> centre_ >> radius_;
  }
};

// Triangle.h
...
#include "Shape.h"

class Triangle: public Shape {
private:
  Point v1_, v2_, v3_;

public:
  // constructor, draw, save
  virtual void save(ostream& os) const {
    os << "Triangle" << endl;
    Shape::save(os);
    os << v1_ << ' ' << v2_ << ' ' << v3_ << endl;
  }

  explicit Triangle(istream& is): Shape(is) {
    is >> v1_ >> v2_ >> v3_;
  }
};
```

```
int main() {
  vector<Shape*> v;
  v.push_back(new Circle(25, Point(2, 3), 6));
  v.push_back(new Triangle(3, Point(0, 0), Point(1, 2), Point(3, 1));
  ...

  for(vector<Shape*>::size_type i = 0; i < v.size(); i++)
    v[i]->save(cout);                          └ // save all shapes

  for(vector<Shape*>::size_type i = 0; i < v.size(); i++)
    delete v[i];
}

void draw_all(const vector<Shape*>& v, GC& gc) {
  for(vector<Shape*>::size_type i = 0; i < v.size(); i++)
    v[i]->draw(gc);
}
```

└ does not need to be recompiled even if we add more kinds of shapes and if we cange the implementation of draw in the derived classes (Compiler only looks in the shape class when compiling this)

How can we read back the shapes we have saved?? Use a factory.

Shape data:
25
(2, 3) 6
3
(0, 0), (1, 2), (3, 1)

The save method needs to save the type together with the data.

Circle
25
92, 3) 6
Triangle
3
(0, 0)(1, 2)(3, 1)

We'll implement a constructor that takes an istream to read back the data for each derived class.

The Shape Factory

```
class ShapeFactory {
private:
  istream *in_;

public:
  explicit ShapeFactory(istream& is): in_(&is) {}
  Shape *create(){
    string type;

    if(!(*in_ >> type))
      return 0;

    if(type == "Circle")
      return new Circle(*in_);

    if(type == "Triangle")
      return new Triangle(*in_);

    return 0;
  }
};

int main() {
  vector<Shape*> v;
  ShapeFactory sf(cin);
  Shape *p;

  while((p = sf.create()) != 0)
    v.push_back(p);

  /* code to process the shapes omitted */

  for(vector<Shape*>::size_type i = 0; i < v.size(); i++)
    delete v[i];
}
```

Runtime Type Information (RTTI)

```
#include <typeinfo>
```

This defines a class named type_info. All constructor of this class private. The typeid operator can be used to get type_info object.

```
eg.     type_info  t = typeid(obj);     /* doesn't compile; copy constructor is private */
        type_info &t = typeid(obj);     /* doesn't compile; can't create such a reference to a
                                           temporary object */

const type_info &t = typeid(obj);       /* OK */
```

C – class, then typeid(C) is same as typeid(C&)

To see if an object has a specifiec type, we need to compare its typeid to the typeid of that type.

```
eg.     if(typeid(x) == typeid(C)) ...
                  └ object
```

In order for typeid to be able to retrieve the derived type:

It must be applied to a reference or an object

The classes must be polymorphic

A polymorphic class is one that has at least one virtual function

```
eg.     class B {
        public:
         virtual ~B(){}
        };

        class D: public B {
        };

        B *p = new D;
        D  d;
        B &r = d;

        typeid(p)  == typeid(D*)        // false
        typeid(r)  == typeid(D)         // false
        typeid(*p) == typeid(D)         // true
        typeid(p)  == typeid(B*)        // true
```

The type_info has a name() method that returns a C-style string describing the type. Unfortunately, this string is not standardized – different compilers return different strings for the same type.

Change Shape::save to:

```
virtual void save(ostream& os) const {
  os << typeid(*this).name() << endl;
  os << colour_ << endl;
}
```

We don't need to print the type Circle::save and Triangle::Save.

Change ShapeFactory::create to:

```
Shape* create() {
  string type;

  if(!(*in_ >> type))
    return 0;

  if(type == typeid(Circle.name()))
    return new Circle(*in_);
    ...

  return 0;
}
```

Virtual Functions

- constructor can't be virtual: object is not full-constructed when a constructor is invoked and hence we cannot have late binding
- the assignment operator is not virtual

  B – base class   D – derived class

  ```
                                ┌ different parameters
  B& operator =(const B&);
  D& operator =(const D&);
                                └ D's version does not override B's version
  ```

- destructor are typically virtual

  ```
  B* p = new D;
  delete p;                // if destructor is virtual, D's destructor is called
  ```

Prototype Pattern

- simulates a virtual copy constructor

```
class B {
public:
 virtual void f();
 virtual B* g();
 virtual B& h(int);
};

class D: public B {
public:
 virtual void f();              // overrides B::f
 virtual D* g();                // overrides B::g
 virtual D& h(int);             // overrides B::h
};
```

Note: Overriding version must have the same parameter list as the overridden version but the return type may be slightly different.

```
class B {
public:
 virtual B* clone() const { return new B(*this); }
 …
};

class D: public B {
public:
 virtual D* clone() const { return new D(*this); }     // overrides B's clone
 …
};

B *p = new D(…);
B *q = p->clone();                              // returns a new D object

Circle c;       ┐
Triangle t;     ┘ prototypes

Shape* prototypes[] = {&c, &t};

Shape* p = prototype[choice]->clone();          // create new Shape of the
                                                   appropriate type
```

Suppose we have a vector of Shape*

```
void save_all(const vector<Shape*>& v, ostream& os) {
  vector<Shape*>::iterator it;

  for(it = v.begin(); it != v.end() ++it)
    (*it)->save(os);
}
```

- this is similar to the for_each algorithm which applies a function to each object in a range. But here, we want to invoke a method on the objects in a range.

Pointers to Members

eg.

```
struct C {
  int n;

  explicit C(int m = 0): n(m) {}

  void add(int x) {
    n += x;
  }

  void add() {
    add(1);
  }
};

int C::*p;        // p is a pointer to a member of C that has type int

p = &C::n;
C = x;
cout << x.*p << endl;

void(C::*f)() = &C::add;

(x.*f)();
(x.*g)(2);
```

- There are function templates mem_fcn and mem_fun_ref that return a function object of the appropriate type from a pointer to member function.

- There are 8 types of function objects:

  3 criteria:

  - number of arguments of method: 0 or 1
  - const vs. non_const method
  - are we invoking the method on a reference or a printer?

number of arguments

	0	1	
pointers	mem_fun_t const_mem_fun_t	mem_fun1_t const_mem_fun1_t	non_const const
references	mem_fun_ref_t const_mem_fun_ref_t	mem_fun1_ref_t const_mem_fun1_ref_t	

eg. mem_fun1_ref_t<**void**, **C**, **int**> h(&C::add);

h(x, 3); ↝ x.add(3);

- mem_fun and mem_fun_ref provide a link between OOP and generic programming

We'll look at how mem_fun1_ref_t can be implemented:

```
template<typename Ret, typename C, typename Arg>

class Mem_fun1_ref_t: public binary_function<C, Arg, Ret> {
private:
  Ret(C::*f_)(Arg);

public:
  explicit Mem_fun1_ref_t(Ret(C::*f)(Arg)): f_(f) {}
  Ret operator()(C& x, Arg a) {
    return (x.*f_)(a);
  }
};

template<typename Ret, typename C, typename Arg>
inline Mem_fun1_ref_t<Ret, C, Arg>
Mem_fun_ref(Ret(C::*f)) {
  return Mem_fun1_ref_t<Ret, C, Arg>(f);
}

void save_all(const vector<Shape*>& v, ostream& os) {
  for_each(v.begin(), v.end(), bind2nd(mem_fun(&Shape::save), os));
}
```

A Fraction Class

operations: +, -, *, /
+=, -=, *=, /=
==, !=, <, <=, >, >=
++, -- (2 versions each)
<<, >>

- + can be a member function or a non-member function
- member: `Fraction Fraction::operator+(const Fraction& other);`

  Fraction f1, f2;   f + 1 $\rightsquigarrow$ f.operator+(1);
  1 + f doesn't compile

  f1 + f2 $\rightsquigarrow$ operator + (f2);

- non-member: `Fraction operator+(const Fraction& lhs, const Fraction& rhs);`

  f1 + f2 $\rightsquigarrow$ operator+(f1, f2);   1 + f $\rightsquigarrow$ operator+(1, f) convert to fraction

- Choose the non-member version if we want to take advantage of automatic conversions for both arguments

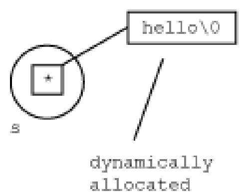
```
class Fraction {
private:
  long n_, d_;
  void reduce();          // reduce to standard form

public:
  Fraction(long num = 0, long den = 1): n_(num), d_(den) {
    if(d == 0)
      throw "...";
  }

  // compiler-generated copy constructor, destructor, and assignment

  Fraction& operator +=(const Fraction& src) {  // Note:  same prototype as operator =
    n_ = n_ * src.d_ + d_ * src.n_;
    d_ *= src.d_;
    reduce();
    return *this;
  }

// Similarly for -=, *=
```

```
Fraction& operator /=(const Fraction& src) {
  if(src.n_ == 0)
    throw "Fraction& operator /=(const Fraction& src): division by 0";

  n_ *= src.d_;
  d_ *= src.n_;
  reduce();
  return *this;
}

Fraction& operator++() {              // prefix version
  *this += 1;
  return *this;
}

Fraction operator++(int) {            // postfix version
  Fraction copy(*this);
  ++*this;                            // call prefix version
  return copy;
}

// similarly with operator--() and operator--(int)

friend std::ostream& operator <<(std::ostream&, const Fraction&);
...
friend bool operator==(const Fraction&, const Fraction&);
...

};

inline Fraction
operator +(const Fraction& lhs, const Fraction& rhs) {
  Fraction copy(lhs);
  return copy += rhs;
}

// similarly for -, *, /
inline bool
operator ==(const Fraction& lhs, const Fraction& rhs) {
  return lhs.n_ * rhs.d_ == lhs.d_ * rhs.n_;
}

inline bool
operator <(const Fraction& lhs, const Fraction& rhs) {
  return lhs.n_ * rhs.d_ < lhs.d_ * rhs.n_;
}

// !=, <=, >, >= can call == and <
```

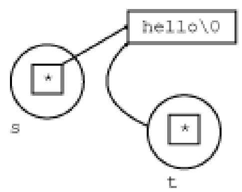
```cpp
/*
eg. inline bool operator >=(const Fraction& lhs, const Fraction& rhs) {
  return !(lhs < rhs);
}
*/

inline std::ostream&
operator <<(std::ostream& os, const Fraction& f) {
  std::ostringstream oss;

  if(f.d_ == 1)
    oss << f.n_;
  else
    oss << f.n_ << '/' << f.d_;

  return os << oss.str();
}

inline std::istream&
operator >>(std::istream& is, Fraction& f) {
  std::string word;

  if(!(is >> word))
    return is;

  std::istringstream iss(word);
  int   n, d;
  char c, extra;

  if(iss >> n >> c >> d && !(iss >> extra) && c == '/' && d != 0) {
    f = Fraction(n, d);
    return is;
  }

  // need to clear iss and seek back
  std::istringstream iss2(word);

  if(iss2 >> n && !(iss >> extra)) {
    f = Fraction(n, 1);
    return is;
  }

  is.setstate(std::ios_base::failbit);
  return is;
}
```

```
void Fraction::reduce() {
  if(d == 0)
    throw "...";

  if(n_ == 0) {
    d_ == 1;
    return;
  }

  if(d < 0) {
    n_ *= -1;
    d_ *= -1;
  }

  long g = gcd(labs(n_), labs(d_));
  n_ /= g;
  d_ /= g;
}

long gcd(long a, long b) {                  // precondition: a > 0 && b > 0
  long c;

  while((c = b % a) != 0) {
    b = a;
    a = c;
  }

  return a;
}
```

A String Class



We'll need a destructor to de-allocate the memory. We also need to implement our own copy constructor and assignment.



Problematic – if one of them goes out of the scope, the other will be pointing to invalid memory.

Note: We need to implement our own copy constructor, assignment, and destructor since we'll be using dynamic memory.

```
class String {
private:
  char* s_;
  enum {bufsiz_ = 1024};

public:
  typedef size_t size_type;
  typedef char& reference;
  typedef const char& const_reference;
  typedef char* iterator;
  typedef const char* const_iterator;

  String(const String& src): s_(new char[strlen(src.s_) + 1]) {
    strcpy(s_, src.s_);
  }

  ~String(){ delete[] s_; }

  string& operator =(const string& src) {
    if(this != &src) {      // make sure we are not assigning to ourself
      char* tmp = new char[strlen(src.s_) + 1];      ie. not s = s;
      strcpy(tmp, src.s_);
      delete[] s_;
      s_ = tmp;
    }
    return *this;
  }
```

```
  size_type length() const { return strlen(s_); }
  size_type size() const { return length(); }

  reference operator[](size_type n){ return s_[n]; }
  const_reference operator[](size_type n) const { return s_[n]; }
  const char* c_str() const { return s_; }
  iterator begin() { return s_; }
  const_iterator begin() const { return s_; }
  iterator end() { return s_ + strlen(s_); }
  const_iterator end() const { return s_ + strlen(s_); }
  string& operator +=(const string& other) {
    char* tmp = new char[strlen(s_) + strlen(other.s_) + 1];
    strcpy(tmp, s_);
    strcat(tmp, other.s_);
    delete[] s_;
    s_ = tmp;
    return *this;
  }

  // friend declarations...
};

inline string operator +(const string& lhs, const string& rhs) {
  string cpy(lhs);
  return copy += rhs;
}

inline std::ostream&
operator <<(std::ostream& os, const string& s) {
  return os << s.s_;
}

inline std::istream&
operator >>(std::istream& is, string& s) {
  char buffer[String::bufsize];
    >> std::setw(String::bufsize_] >>
  if(!is >> buffer)
    return is;
  s = String(buffer);
  return is;
}

inline bool
operator ==(const string& lhs, const string& rhs) {
  return strcmp(lhs.s_, rhs.s_) == 0;
}
```

```
inline bool
operator <(const string& lhs, const string& rhs) {
  return strcmp(lhs.s_, rhs.s_) < 0;
}
...
```

Alternative implementation of operator =:

```
inline void
String::swap(string& other) {
  std::swap(s_, other.s_);
}

inline string&
String::operator=(const string& src) {
  string copy(src);
  swap(copy);
  return *this;
}

string s("hello");
copy(s.begin(), s.end(), ostream_iterator<char>(cout, "\n"));
transform(s.begin(), s.end(), s.begin(), ::toupper);
                                              \
                                              toupper in the global scope
                                             (there are two versions: one in
                                             "using namespace," other in C library)
```

A Date Class

- three data members: year_, month_, day_
- operations:

  comparisons <, ==, ...
  ++, --
  <<, >>

```
// Date.h
...
class Date {
private:
  int year_, month_, day_;
  static const int daysInLeapYear[];           // declaration only
  static const int daysInRegularYear[];

public:
  explicit Date(int year = 2000, int month = 1, int day = 1);
  // compiler-generated copy constructor, destructor, and assignment
  static bool isLeapYear(int year);
  bool isLeapYear() const;
  static Date today();
  static isValidDate(int year, int month, int day);
  Date& operator++();                          // prefix
  Date operator++(int);                        // postfix
  Date& operator--();
  Date operator--(int);

  bool operator==(const Date&)const;
  ...

  friend std::ostream& operator<<(std::ostream&, const Date&);
  ...
};
```

```
// Date.cpp
#include <iostream>
#include “Date.h”
#include <ctime>
using namespace std;

                                    dummy value
                                         |
const int Date::daysInLeapYear[] = {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
const int Date::daysInRegularYear[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

Date::Date(int year, int month, int day): year_(year), month_(month), day_(day) {
  const int* days = isLeapYear(year_) ? daysInLeapYear : daysInRegularLeapYear;

  if(!isValidDate(year, month, day))
    throw “...”;

  return 1 <= month_ && month_ <= 12 && 1 <= day_ && day_ <= days[month_];
}

bool Date::isLeapYear(int year) {
  if(year % 400 == 0)
    return true;
  if(year % 100 == 0)
    return false;
  if(year % 4 == 0)  ┐
    return true;     |  return year % 4 == 0;
  return false;      ┘
}

bool Date::isLeapYear() const {
  return isLeapYear(year_);
}

Date Date::today() {
  time_t t = time(0);
  tm *tt = localtime(&t);
  return Date(tt->year + 1900, tt->month + 1, tt->mday);
}
```

```
Date& Date::operator++() {
  const int* days = isLeapYear(year_) ? daysInLeapYear : daysInRegularYear;

  if(++day_ > days[month_]) {
    day_ = 1;

    if(++month_ > 12) {
      month_ = 1;
      ++year_;
    }
  }
  return *this;
}

Date Date::operator++(int) {
  Date copy(*this);
  ++*this;
  return copy;
}

Date& Date::operator--() {
  if(--day_ < 1) {
    if(--month_ < 1) {
      month_ = 12;
      --year_;
    }

    day_ = isLeapYear(year_) ? daysInLeapYear[month_] : daysInRegularYear[month_];
  }
  return *this;
}

Date Date::operator--(int) {
  Date copy(*this);
  --*this;
  return copy;
}

bool Date::operator==(const Date& other) const {
  return year_ = other.year_ && month_ = other.month_ && day_ == other.day_;
}

bool Date::operator !=(const Date& other) const {
  return !(*this == other);
}
```

```
bool Date::operator <(const Date& other) const {
  if(year_ != other.year_)
    return year_ < other.year_;
  if(month_ != other.month_)
    return month_ < other.month_;
  return day_ < other.day_;
}

ostream& operator <<(ostream& os, const Date& d) {
  ostringstream oss;
  oss << setfill('0') << d.year_ << '/' << setw(2) << d.month_ << '/' << setw(2) << d.day_;
  return os << oss.str();
}

istream& operator >>(istream& is, Date& d) {
  string word;

  if(!(is >> word))
    return is;
  istringstream iss(word);
  int year, month, day;
  char c1, c2, extra;

  if(iss >> year >> c1 >> month >> c2 >> day && !(iss >> extra) && c1 == '/' && c2 == '/' &&
     Date::isValidDate(year, month, day)) {
    d.year_ = year, d.month_ = month, d.day_ = day;
    return is;
  }
  is.setstate(ios_base::failbit);
  return is;
}
```

If we don’t need to do validation:

```
char c1, c2;
return is >> d.year_ >> c1 >> d.month_ >> c2 >> d.day_;
```

Constructors and Destructors

- The compile-generated copy constructor calls the copy constructors of the base classes and then calls the copy constructor of each data member.

```
class C: public A, public B {     // class C inherits from both A and B
private:
 int x;
 D d;
 E e;
};
```

compiler-generated copy constructor

1. calls copy constructor of A
2. calls copy constructor of B
3. performs bitwise copy of x
4. calls copy constructor for d
5. calls copy constructor for e

Similarly for compiler-generated assignment operator and default constructor

Constructor are for initalization

RAIT – Resource Acquisition Is Initialization

eg.
```
ifstream in("data");          // need to acquire a file handle;
                                 file handle is released by the destructor
```

- thread synchronization
  - mutex – for mutual exclusion
  - two operations: lock and unlock   // ie. bathroom stall
  - protect a section of code by a mutex so that only one thread can execute within that section at any time

    ```
    mutex m;
    lock(&m);

    /* critical section */
    unlock(&m);
    ```

- If the thread forget to unlock the mutex, no other thread can go into the critical section
- “automate” the locking and unlocking of the mutex

```
class Lock {
private:
  mutex *pm_;

public:
  Lock(mutex *pm): pm_(pm) {
    lock(pm_);                          // lock mutex
  }

  ~Lock(){ unlock(pm_); }

private:
  Lock(const Lock&);                    // prevents copying of locks
  Lock& operator = (const Lock&);       // prevents assignment of locks
};

mutex m;

{
  Lock l(&m);
  // critical section
}
```

Exceptions

- throw/catch (no finally)

```
void f() {
  int n = 1;
  throw n;                        // a copy of n is thrown
}

int main() {
  try {
    f();
  } catch(long l) {
    ...
  } catch(short s) {
    ...
  } catch(int n) {
    ...                           // exception caught here
  } catch(...) {                  // catches all
    ...                           // exceptions
  }
}
```

Matching of exception objects with catch clauses is strictly than that in function overloading.

```
A          try {
↑            g();          ┌----------- catch exceptions by reference to reduce copying
B          } catch(const C& c) {
↑            ...
C          } catch(const B& b) {
             ...
           } catch(const A& a) {
             ...           └----------- this can also catch exceptions of type B and C
           }
```

Rethrowing an exception:

```
try {
  g();
} catch(const &C c) {
  // partically handle exception
  throw;                          // rethrow c if we had throw c; then we throw a copy of c
}
```

- exceptions can be used for flow control

eg. breaking up an amount into smaller demoninations

amount 1159 coins[200, 100, 25, 10, 5, 1]

1159 – 200 = 959 159

- if amount is 0, return empty list
- if amount < 0 or no more denominations, no solution

```
breakup(1159, [200, 100, 25, 10, 5, 1]);
breakup(959,  [......................]);
breakup(159,  [200, 100, 25, 10, 5, 1]);
breakup(159,  [100, 25, 10, 5, 1]);
breakup(59,   [100, 25, 10, 5, 1]);
breakup(59,   [25, 10, 5, 1]);
```

- breakup(amount – first denomination, ...)

```
vector<int>breakup(int amount, const vector<int>& denom, vector<int>::size_type first) {
 if(amount == 0)
   return vector<int>();
 if(amount < 0 || first == denom.size())
   throw "no solution";
 try {
   vector<int> v = breakup(amount – denom[first], denom, first);
   return v.push_back(demon[first]);
 } catch(const char*) {
   return breakup(amount, denom, first + 1);
 }
}

int main() {
 int a[] = {200, 100, 25, 10, 5, 1};

 vector<int>denom = (a, a + 6);

 try {
   vector<int> v = breakup(1159, denom, 0);
   // print element of v
 } catch(const char* s) {
   cerr << s << endl;
 }
}
```

Three Possible Guarantees for an Operation

1. no-fail eg. delete

2. basic guarantee – system is still in a consistent state

eg. operator << to read a Name object

Assume a valid Name has one word for its firstName
Assume a valid Name has one word for its lastName

```
istream& operator >>(istream& is, Name& n) {
  return is >> n.firstName >> n.lastName;
}
```

If we succeed in the first name but fail to read the last name, the object is still in a consistent state.

```
istream& operator >>(istream& is, Name& n) {
  string first, last;

  if(is >> first >> last) {
    n.first_ = first;
    n.last_  = last;
  }
  else
    is.setstate(ios_base::failbit);

  return is;
}
```

Example: Strong Guarantee

```
istream& operator >>(istream& is, FulltimeEmployee& e) {
  Employee tmp;
  float salary;

  if(is >> tmp >> salary && salary >= 0) {
    (Employee&)e = tmp;
    e.salary_ = salary;
  }
  else
    is.setstate(ios_base::failbit);

  return is;
}
```

When an exception is caught, the stack is unwined(and destructors are called)

```
void f() {        void g() {        void h() {
  a a;              c c;              e e;
  b b;              d d;              throw 1;
  g();              h();            }
}                 }
```

- order of destructor calls: ~E(), ~D(), ~C(), ~B(), ~A()

Multiple Inheritance

- a class can inherit from two or more classes

```
class D: public B1, public B2 {
  ...
};
```

- problems associated with multiple ineritance

```
void f();                void f();
          B1        B2
           \        /
              D
void g();           void g(int);
```

```
D d;
d.f();
d.B1::f();
d.B2::f();
d.g();
d.B1::g();
d.B2::g(3);
```

```
   C
  /  \
B1   B2
  \  /
   D
```

Does D have two copies of x?? Yes to have one copy of x, we need to use virtual inhertiance. But we can't just use virtual inheritance for D.

```
class B1: virtual public C {
  B1(const B1& src)::C(src), ... {}
};

class B2: virtual public C {
  B2
};
```

```
class D: public B1, public B2 {
  D(const D& src): C(src), B1(src), B2(src), ... {}
  // C(src) needs to explicity call the copy constructor of C
};

D& D::operator =(const D& src) {
  if(this != &src) {
    B1::operator=(src);
    B2::operator=(src);
    ...
  }
  return *this;
}
```

Casts

Besides the C-style casts, there are four new casts in C++ static_cast, const_cast, reinterpret_cast, dynamic_cast

static_cast

```
int *p = static_cast<int*>(malloc(100 * sizeof(int)));
```

C++ has stricter type-checking than C

```
double d = 1.234;
int n = static_cast<int>(d);
```

reinterpret_cast

for nonportable operations

```
int n = 0x01020304;
char *p = reinterpret_cast<char*>(&n);
```

const_cast

- typically use to case away const-ness
- target type must be a pointer or a reference

  C-class

```
void f(C& c);    void g(const C& c);

const C c1;
C c2;
g(c2);                      // OK, cast not needed
f(c1);                      // NOT ok
f(const_cast<C&>(c1));      // compiles fine, but may cause a runtime error if f tries to
                               change c1
```

dynamic_cast

- for casting down an inhertiance hierarchy
- class must be ploymorphic (ie. must have virtual function)
- target type must be a ointer or a reference

```
A         A *p = new ...;
↑         B *q = dynamic_cast<B*>(p);         // this returns the null pointer on failure
B         if(q != 0) {
↑           // use q
C           // p point to a B or a C
          }

A& r = ...;       // can refer to an A, a B, or a C

try {
  C& s = dynamic_cast<C&>(r);
} catch(const bad_cast& e) {
  cerr << e.what() << endl;
}
```

Virtual Functions

How does the system determine which virtual function to execute?

static vs. dynamic type

- static type – declared type
- dynamic type – only applies to pointer and references actual type

```
A        A *p = new B;        static_type of p: A*
↑                             dynamic type of p: B*
B
↑        p = new C;           dynamic type of p changed to C*
C

         Bb;
         A& r = b;            static type of r: A
                              dynamic type of r: B
```

When the compiler compiles your program, it only uses static type

```
A *p = new B;

p->f();     // compiler looks for f in A and found it
p->g();     // compiler looks for g in A; no g() in A therefore doesn't compile

class A {
public:
  virtual void f();
  // no g();
  virtual void h();
  virtual void k();

private:
  virtual void m();
};

class B: public A {
public:
  virtual void f();
  virtual void g();
  // no h();
  virtual void k(int);        // hides all inherited k's
};
```

1. Compiler determines whether to use early or late binding. Late binding is only used when a ***virtual*** method is invoked through a ***reference of a pointer***.

2. If late binding is used, the compiler generates code that determines at runtime (by looking at the dynamic type) which version of the virtual method is actually invoked

3. Basically, everything in the base class is inherited by the derived class but some may not be accessible or may be ***hidden***.

```
B b;
A& r = b;

r.g();   // does not compile (compiler looks in a for g())
r.h();   // looks in A for h(), OK, latebinding, no overriding version, calls A::h()

B& s = b;

s.g();   // looks in b for g(), OK, latebinding, calls B::g()
s.h();   // compiles OK (h() is inherited)
s.k();   // doesn't compile, inherited k() hidden by k(int)
s.m();   // doesn't compile, not accessible in B
```

4. Default arguments and access control are determined at compile time; they have no effect on runtime.

```
class A {
public:
  virtual void p(int = 1);
  virtual void q(int = 1);
  virtual void x();
};

class B {
public:
  virtual void p(int = 2);
  virtual void q();                // does not override A::q(int)

private:
  virutal void x();
};

B b;
A& r = b;
r.p();                             // compiler rewrites this to r.p(1); late binding, calls B::p(1)
r.q();                             // compiler rewrites this to r.q(1); late binding, calls A::q(1)
r.x();                             // looks in A for x(), compiles fine; calls B::x()

A:                                 B:

virutal void y();                  void y();  // automatically virtual
```