C++ Code Examples

Contents

1 Introduction

1.1 Java

Motivation

- Developed for consumer products (e.g. VCRs).
- Created at a time when the cost of memory was decreasing and
- Processor speeds were increasing.

Mission

- **Reduce development time and increase portability**.
- Simplicity – Fewer options, fewer mistakes.
- Security – Run time checks, garbage collection.

Platform independent

- Any machine.
- How? – JVM must correct hardware discrepancies with additional software.

Why does it appear to run slower?

- It is doing more work.

1.2 C++

Motivation

- Designed for system programming.
- Created at a time when memory was expensive and
- Processor speeds were slow.

Mission

- **A program should not have to pay for the features it is not using**.
- Get close to the machine.
- Limit features.
- Concise representation.

C with class (classes) a.k.a C++

- Added object-oriented technology to C.

Legacy issues

- Programmers should understand the past and know the current best practices.

- Problems with legacy code:
  - `#define` vs. const
  - macros vs. inline and templates
  - char[] vs. string class
  - c style I/O vs. I/O streams.

2 Basics

2.1 Preprocessor

Consider the following

```
#include <iostream>

int main(){

    // DO SOMETHING

return 1;
}
```

What is going on?

- Before the compiler, Preprocessor.
- Scans programs for directives that start with # marks.
- # include means include the file textually at the point indicated.
- e.g. the file is combined with the current file.
- Then, the combined file is passed to the compiler.
- < > means search system library first.

2.2 Namespaces

- A way of avoiding name collisions in large programs.
- The same name may be used for different purposes.
- Declare all functions and definitions under one namespace;

```
using std::cout;

using namespace std;
```

2.3 Integers

Java

- short = 16-bits
- integer = 32-bits
- long = 64-bits
- Additional software corrects for different machines.

C++

- Does not specify number of bits – WHY?
- short int $\leq$ int $\leq$ long int
- long and short are modifiers

Modifiers

- unsigned are numbers > 0
- 16-bits – $2^{16} = 65536$
- signed $-32768, 32768$
- unsigned $0, 65536$

Issues

- No warning when casting an unsigned int.

```
int x = -5;
unsigned int y = x;  // y is a large number!
```

- Integer division with negative numbers is platform dependent.
- Never use remainder operator with negative values.

2.4 Characters

- minimum length 8-bits

2.5 Boolean

- C++ keyword *bool*
- Java keyword *boolean*
- A nonzero *int* is FALSE
- NULL pointer is FALSE
- Legacy code – bool is recent!

```
int continue = 0;
if(continue){
   // do something
   continue++;
}
```

2.6 Floating point values

- float, double, long double
- Standard library uses double, so it is best not to use float.
- You can assign a double to int without a cast, but this will sometimes generate a warning.
- Standard routines will not throw an exception.

2.7 Type conversions

C++ converts from one type to another when....

- Assignment
- Expressions
- Pass to function (discuss soon)

Assignment

- Converting long double to double – precision loss
- double to int – loss of fraction, may be out of range
- long int to short int – out of range

Expression

- Floating point numbers – converts all values to the type with greatest precision.
- if a long double is in an expression, all values are converted to a long double.
- if only integers are used, then only integers are used.
- Be careful using *signed* and *unsigned* integers.

```
int num1=1;
int num2=2;
double results = num1/num2; // What is the result ?
```

2.8 Comparing data type size on different machines

```
#include <iostream>

using std::cout;
using std::endl;

int main(){

   cout << "bool         = " << 8*sizeof(bool) << endl;
   cout << "char         = " << 8*sizeof(char) << endl;
   cout << "short int    = " << 8*sizeof(short int) << endl;
   cout << "int          = " << 8*sizeof(int) << endl;
   cout << "long int     = " << 8*sizeof(long int) << endl;
   cout << "double       = " << 8*sizeof(double) << endl;
   cout << "long double  = " << 8*sizeof(long double) << endl;

}
```

```
32-bit machine
------------------------------------------------------------

bool         = 8
char         = 8
short int    = 16
int          = 32
long int     = 32
double       = 64
long double  = 96

64-bit machine
------------------------------------------------------------

bool         = 8
char         = 8
short int    = 16
int          = 32
long int     = 64
double       = 64
long double  = 128
```

2.9 Understanding numeric limits

```
#include <iostream>
#include <numeric>

using std::cout;
using std::endl;

int main(){

   cout << "short int  = ";
   cout << std::numeric_limits<short int>::min() <<  " ";
   cout << std::numeric_limits<short int>::max() <<  endl;

   cout << "      int  = ";
   cout << std::numeric_limits<int>::min() <<  " ";
   cout << std::numeric_limits<int>::max() <<  endl;

   cout << "long  int  = ";
   cout << std::numeric_limits<long int>::min() <<  " ";
   cout << std::numeric_limits<long int>::max() <<  endl;

   int a = std::numeric_limits<int>::max();  // max int

   short int b = a;
   double c = 1/2;
   double d = 100/27;

   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
   cout << "c = " << c << endl;
   cout << "d = " << d << endl;

   return 0;
}
```

```
short int  = -32768 32767
      int  = -2147483648 2147483647
long  int  = -2147483648 2147483647

a = 2147483647
b = -1
c = 0
c = 3
```

2.10 Arrays and Strings

An array holds several values of the *same* type.

- Static - remain the same size.
- Declare type and number of elements.
- No *new* keyword.

```
int main(){

    // Declare size and fill with for loop.
    int data[100];

    for(int i=0; i<100; i++)
     data[i] = 0;

    // Declare with initialization list
    int limits[] = {10,20,30,40,50};
    int list[2]  = {10,20,30};       // Compilation error
    int n[10]    = {10,20,30,40,50}; // logic error

    // Array of characters?
    char text = "an array of characters";
}
```

- Does not know its extent.
- How big is an array?

```
int main(){
    const int arraySize = 100;
    int data[arraySize];
    for(int i=0; i<arraySize; i++)
        data[i] = 0;
}
```

- For functions, it is best to pass the size.

A string is a series of consecutive bytes of memory.

- Array of char.
- Last character is the null character.
- String literal.

```
char name1[] = "bob";  // adds the null char to the end
```

- List of chars

```
char name1[4] = {'M','a','t','t'}; // Not a string... array of chars
char name2[5] = {'M','a','t','t','\0'};
```

- strlen() is useful.

2.11 Structures

- Arrays must hold the same *type*.
- A structure is like a class with only data fields and *public* access.
- No inheritance.
- "Collection of data types".
- Declaration does not create a global variable.

```
struct myData{
    int    data1;
    double data2;
};
```

```
int main(){
    myData varName = {1,20.3};
    varName.data1  = 10;
    varName.data2  = 1.1;
}
```

2.12 Unions

- Similar to *struct*, but different data fields share the same location in memory (laid on top).
- Only use one field at a time.

```
union myData{
    int    data1;
    double data2;
}
int main(){
    myData number;
    number.data1 = 10;   // data2 is meaningless
    number.data2 = 5.0;  // data1 is meaningless
}
```

- holds and *int* OR a *double*, not both.

2.13 Enumerated values

- Distinct integer values with named constants.
- Way of creating symbolic constants.

```
enum color {red, orange, yellow};
```

- Names must be distinct.

```
enum fruit {apple, orange}; // orange is defined.
```

- Values start at 0 unless specified, and increment by 1.

```
enum status {CON=1, WON, LOST};  // values 1,2,3
```

- Access

```
int main(){
   status gameStatus;  // can contain any three values.
   gameStatus = WON;
   if( gameStatus == WON ){
      std::cout << "Congrats!" << std::endl;
   }
   return 0;
}
```

- Used in lots of code.
- Cannot access the address of an *enum* – “the enum hack”.

3 Input and Output

3.1 Standard IO Library

- Need to be aquainted with this for legacy reasons.
- Consider *printf*

```
int    N = 10;
int    M = 5;
double D = 0.5*(10 + 5);
printf("The average of %d and %d is: %g", N,M,D);
```

- Painful... we have to format each number.
- Solutions:
  - Overload the print() function... clumsy, but
  - Allows programmer to define output easily for new data types
  - Even better: overload an expression like << or >>.

3.2 Steam IO Facility

Header files

```
#include <iostream.h>  // I/O
#include <fstream.h>   // file I/O
```

What is a stream?

- Three streams just exist (four actually):
  - cout - standard output (terminal, redirection to file)
  - cin - standard input (terminal, redirection from file)
  - cerr - standard error
- File streams: input and output.
- Input

```
ifstream inFile;
inFile.open("myInputFile");
or
ifstream inFile("myInputFile");
```

- output

```
ofstream outFile("myOutputFile");
```

- Closing a stream.

```
inFile.close();
outFile.close();
```

3.3 Standard Output

- Stream Objects, not functions.
- *std::cout* writes to standard output.
- << writes data to the stream object.
- This can be overloaded for any type of object.

```
#include<iostream>

int main(){

    int    N = 10;
    int    M = 5;
    double D = 0.5*(10 + 5);
    std::cout << "The average of " << N << " and " << M << " is " << D << "\n";

// Which can be coded as...
    cout << "The average of";
    cout << N;
    cout << " and ";
    cout << M ;  // ect. ....

    return 0;
}
```

- “using std::cout;” or “using namespace std”;

3.4 Standard Input

- *std::cin* reads from the terminal.
- >> writes input to data object.

```
#include <iostream>

using namespace std;

int main(){
    char name[20];
    cout << "Enter your name:\n";
    cin >> name;
    cout << "Thanks, " << name << endl;
    return 0;
}
```

- `strlen(name)` must be large enough.
- `cin` uses white space to delineate a string
- `cin` offers no protection against placeing a 30 character string into a 20 character array – this will lead to a *segmentation fault*.

3.5 GetLine as a solution

- What about entire phrases or lines?
- *getline()* reads the entire line until it hits a newline character.
- Two arguments: name of array and limit of characters.

```
std::cin.getline(name,5);
std::cout << strlen(name) << std::endl;
```

- Will only read the first 5 characters.
- What if I want to read an integer and then a string?

```
char name[5];
int  year;
cout << "What year were you born and what is your name?" << endl;
std::cin >> year >> name;

char name[20];
int  year;
cout << "What year were you born and what is your full name?" << endl;
(std::cin >> year).getline(name,20);
```

3.6 File Input and Output

- Like cout/cin
- Can append a file also.

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){

    // Write to a file
    ofstream outfile("myOutput.txt");
    if(!outfile)
cerr << "Problem opening file" << endl;
    else{
outfile << "Hello" << endl;
outfile << "This is line 2" << endl;
    }
    outfile.close();

    // Append a file
    outfile.open("myOutput.txt", ios_base::app);
    if(!outfile)
cerr << "Problem opening file" << endl;
    else{
outfile << "Should be line 3" << endl;
    }
    outfile.close();

    // Read text from a file
    const int lineSize = 200;
    char buffer[lineSize];
    ifstream infile("myOutput.txt");
    if(infile){
while(infile >> buffer){
    cout << buffer << endl;
}
    }
    infile.close();

    // Read line from a file
    infile.open("myOutput.txt");
    if(infile){
infile.getline(buffer,lineSize);
while(!infile.eof()){
    cout << buffer << endl;
    infile.getline(buffer,lineSize);
}
    }
    infile.close();

    return 1;
}

Hello
This
is
line
2
Should
be
line
3
Hello
```

```
This is line 2
Should be line 3
```

3.7 Standard I/O code examples

code-standard-io.cc

```
#include <iostream>

//----------------------------------------------------------------------------
int main(){

    // Standard output example
    char   name[]  = "Holds a name of length 36 characters";
    int    account = 657;
    double balance = 10.5;

    std::cout << account << " ";
    std::cout << name    << " ";
    std::cout << balance << std::endl;

    // Standard output example
    std::cin >> account;
    std::cin >> name;
    std::cin >> balance;


    return 0;
}
```

3.8 File I/O

code-file-io.cc

```
#include <iostream>
#include <fstream>

using namespace std;

//----------------------------------------------------------------------------
int main(){

    char   name[100] = "Brad";
    int    account   = 645;
    double balance   = 103.8;

    // Create an output file stream
    ofstream outFile("data");
    outFile << account << " " << name << " " << balance << endl;
    outFile.close();

    // Read from a file
    ifstream inFile("data");
    inFile >> account >> name >> balance;
    inFile.close();

    cout << account << " " << name << " " << balance << endl;

    return 0;
}
```

3.9 Using the EOF()

code-file-eof.cc

```
#include <iostream>
#include <fstream>

using namespace std;

//--------------------------------------------------------------------------------
int main(){

// Create an output file stream
ofstream outFile("data");
for(int i=0; i<10; i++)
    outFile << i+1 << endl;
        outFile.close();

// Create file stream
ifstream inFile("data");
int number = 0;
inFile >> number;
while( !inFile.eof() ){
    cout << number << endl;
            inFile >> number;
}
inFile.close();

// Or, reset and use an istream reference
inFile.open("data",ios::in);
while( inFile >> number )
    cout << number << endl;
inFile.close();

    return 0;
}
```

3.10 Getline() revisited

code-getline.cc

```
#include <iostream>
#include <fstream>

using namespace std;

//--------------------------------------------------------------------------------
int main(){

    char name[80];

    // Getline example
    fstream inFile("celebNames",ios::in);

    inFile.getline(name,80);
    while( !inFile.eof() ){
        cout << name << endl;
        inFile.getline(name,80);
    }
    inFile.close();

    return 0;
```

```
}
```

3.11 Get()

code-get.cc

```
#include <iostream>
#include <fstream>

using namespace std;

//--------------------------------------------------------------------------------
int main(){

    const int MAXCHAR = 80;
    char name[MAXCHAR];

    fstream inFile("celebNames",ios::in);

    inFile.get(name,MAXCHAR);
    cout << name << endl;
    inFile.get(name,MAXCHAR);
    cout << name << endl;

    inFile.close();

    return 0;
}
```

celebNames

```
Brad Pitt
Britney Spears
Lindsay
Mr. Trump
```

celebAccounts

```
12343  Jennifer 10239.0
83738  Brad     837.5
39883  Britney  89378.5
38733  Trump    8838.5
82733  Lindsay   38.5
```

celebNumbers

```
1033 Brad Pitt
8384 Britney Spears
9383 Lindsay
9373 Mr. Trump
```

3.12 Solutions to your questions

code-questions.cc

```
#include <iostream>
#include <limits>
#include <cmath>
#include <cstring>
#include <fstream>

using namespace std;

int main(){

    // signed vs. unsigned int v.s int
    cout << numeric_limits<unsigned int>::digits10 << " ";
    cout << numeric_limits<unsigned int>::max() << endl;
    cout << numeric_limits<signed int>::digits10 << " ";
    cout << numeric_limits<signed int>::max() << endl;
    cout << numeric_limits<int>::digits10 << " ";
    cout << numeric_limits<int>::max() << endl;

// initilization list after declaration?
int limits[5] = {1,2,3,4,5};
//limits = {1,2,3,4,5}; // did no compile

// Number of items in an array using sizeof().
int numItems = sizeof(limits)/sizeof(int);
cout << "Number of items = " << numItems << endl;  // returns 5

// Null character
char name[3] = {'m','e','\0'};  // this is a slash zero
cout << "String length = " << strlen(name) << endl; // returns 2

    // File for both input and output?
    fstream myFile;
    myFile.open("temp",ios::out);
    myFile << "Hello" << endl; // Add line
    myFile << "Line two " << endl;
    myFile.close();

    myFile.open("temp",ios::in);
    char buffer[100];
    myFile.getline(buffer,100);
    cout << buffer << endl;
    myFile.close();

    return 0;
}
```

output

```
9 4294967295
9 2147483647
9 2147483647
Number of items = 5
String length = 2
Hello
```

4 Pointers and References

4.1 Variable address

code-pointers-1-address.cc

```
#include <iostream>
#include <iomanip>

using namespace std;

//------------------------------------------------------------------------
int main(){

    // Declaration
    int  num;
    int *numPtr = 0;  // Null Pointer

    // Assignment
    num = 10;
    numPtr = &num;    // Gets the address of num.

    // Ouput
    cout << setw(20) << "variable";
    cout << setw(20) << "value";
    cout << setw(20) <<  "address" << endl;
    cout << setw(60) << setfill('-') << "-" << setfill(' ') << endl;

    cout << setw(20) << "num";
    cout << setw(20) << num;
    cout << setw(20) << &num << endl;

    cout << setw(20) << "numPtr";
    cout << setw(20) << numPtr;
    cout << setw(20) << &numPtr << endl;

    cout << "\n" << "    *numPtr = " << *numPtr << endl;
    cout <<         " &(*numPtr) = " << &*numPtr << endl;
    cout <<         "       &num = " << &num << endl;

    return 0;
}
```

output

```
            variable               value             address
------------------------------------------------------------
                 num                  10          0xbffd2538
              numPtr          0xbffd2538          0xbffd2534

    *numPtr = 10
 &(*numPtr) = 0xbffd2538
       &num = 0xbffd2538
```

4.2 Creating objects on the free store

code-pointers-2-new.cc

```
#include <iostream>
#include <iomanip>

using namespace std;

//----------------------------------------------------------------------------
int main(){

    // Declaration
    int *numPtr = 0;

    // Allocate memory
    numPtr  = new int;

    // Assignment
    *numPtr = 5;

    // Ouput
    cout << setw(30) << "numPtr value (address): " << numPtr << endl;
    cout << setw(30) << "Dereferenced value (*numPtr): " << *numPtr << endl;
    cout << setw(30) << "numPtr address (&numPtr) : " << &numPtr << endl;

    // delete numPtr;  // Memory leak without!

    return 0;
}
```

output

```
      numPtr value (address): 0x99c7008
Dereferenced value (*numPtr): 5
   numPtr address (&numPtr) : 0xbff9dd0c
```

4.3 Operations with pointers

code-pointers-3-operate.cc

```
#include <iostream>
#include <iomanip>

using namespace std;

//----------------------------------------------------------------------------
int main(){

    // Declaration
    int  num    = 10;
    int *numPtr = &num;

    cout << "*numPtr = " << *numPtr << endl;
    *numPtr += 5;
    cout << "*numPtr = " << *numPtr << endl;
    cout << "    num = " << num     << endl;
    num = 300;
    cout << "*numPtr = " << *numPtr << endl;

    return 0;
}
```

4.4 Deleting free store memory

code-pointers-4-delete.cc

```
#include <iostream>
#include <iomanip>

using namespace std;

//------------------------------------------------------------------------
int main(){

    double  *realPtr = new double;
    delete realPtr;             // Great
    //delete realPtr;           // Run time error

    int     num = 5;
    int *numPtr = &num;
    //delete numPtr;            // Did not call new

    int *p1 = new int;
    int *p2 = p1;
    delete p2;                  // OK, delete with second pointer
    //delete p1;                // Not OK if previous call

    return 0;
}
```

4.5 Static and Dynamic arrays

code-pointers-5-arrays.cc

```
#include <iostream>

using namespace std;

//------------------------------------------------------------------------
int main(){

    const int num = 100;

    // Dynamic array....
    double *data = new double[num];     // num determined at run time
    for(int i=0; i<num; i++)            // Initialize
        data[i] = i;

    cout << "            data[0] = " << data[0] << endl;
    cout << "              *data = " << *data   << endl;
    cout << "           data[10] = " << data[10] << endl;
    cout << "         *(data+10) = " << *(data+10) << endl;
    cout << endl;

    delete [] data;

    // Static array...
    double staticData [100];            // Declare
    for(int i=0; i<100; i++)            // Initialize
        staticData[i] = i;

    cout << "        *staticData = " << *staticData << endl;
    cout << " *(staticData + 10) = " << *(staticData+10) << endl;
    cout << endl;
```

```
    cout << " sizeof(staticData) = " << sizeof(staticData) << endl;
    cout << "      sizeof(*data) = " << sizeof(*data) << endl;
    cout << "       sizeof(data) = " << sizeof(data) << endl;

    return 0;
}
```

4.6 Const pointers and references

code-pointers-7-const.cc

```
#include <iostream>

using namespace std;

//------------------------------------------------------------------------
int main(){

int num1 = 10;
int num2 = 20;

// Pointer
// --------------------------------------
int *p1 = &num1;

*p1  = 25;
num1 = 15;

// const data, non-const pointer
// --------------------------------------
const int *p2 = &num2;
num2  = 25;
// *p2   = 35; // No

// const pointer, non-const data
// --------------------------------------
int * const p3 = &num1;
*p3 = 10;
//p3  = &num2; // No

// Const pointer and data
// --------------------------------------
const int * const p4 = &num1;
//*p4 = 10;
//p4 = &num2;

return 0;
}
```

5 Functions

5.1 Scope and Extent

code-functions-0-memory.cc

```
#include <iostream>

double a = 0.0;        // Global variable.
const double b = 10;   // Const variable.

int main(){

// Variable c exists on the free-store.
int* c = new int(15);

// Variable i has local extent and is on the stack.
for(int i=0; i<*c; i++){
std::cout << i << std::endl;
}

// Variables a and b have static extent.
std::cout << a << " " << b << std::endl;

delete c;

// c is no longer visible.

return 0;
}
```

5.2 Simple Function Example

code-functions-1-greeting.cc

```
#include <iostream>

// Function prototype
void greeting();

//------------------------------------------------------------------------------
int main(){

    greeting();

    return 0;
}

//------------------------------------------------------------------------------
void greeting(){
    std::cout << "Hello" << std::endl;
}
```

5.3 Pass by Value

code-functions-2-passbyvalue.cc

```
#include <iostream>

// Function prototype
```

```
int factorial(int);

//-------------------------------------------------------------------------
int main(){

    int num = 5;
    int fac = factorial(num);

    std::cout << "!" << num << " = " << fac << std::endl;

    return 0;
}

//-------------------------------------------------------------------------
int factorial(int num){

    int result = num;
    while(num>1){
        num = num-1;
        result *= num;
    }
    return result;
}
```

5.4 Pass by Reference

code-functions-3-passbyreference.cc

```
#include <iostream>

// Function prototype
int factorial(int &);

//-------------------------------------------------------------------------
int main(){

    int num = 5;
    int fac = factorial(num);

    std::cout << "!" << num << " = " << fac << std::endl;

    return 0;
}

//-------------------------------------------------------------------------
int factorial(int &num){

    int result = num;
    while(num>1){
        num = num-1;
        result *= num;
    }
    return result;
}
```

code-functions-4-passbyreference.cc

```
#include <iostream>

struct item{
```

```
    char   name[20];
    double price;
    int    quantity;
};

// Function prototype
void sellItem(item &, int);

//------------------------------------------------------------------------------
int main(){

    double temp  = 0.0;

    item burrito = {"Burrito", 1.95 , 30};

    // Sell 2 boxes of nails
    sellItem(burrito,2);
    std::cout << "Burritos available = " << burrito.quantity << std::endl;

    return 0;
}

//------------------------------------------------------------------------------
void sellItem(item &myItem, int quantity){
    myItem.quantity -= quantity;
}
```

5.5 Passing Pointers

code-functions-12-pointerExample.cc

```
#include <iostream>

using namespace std;

typedef int* intPtr;


void setPointer(intPtr a){
    intPtr c = new int(10);
    cout << "       Address of c = " << c << endl;
    a = c;
    cout << "       Address of a = " << a << endl;
}

void setPointerRef(intPtr &a){
    intPtr c = new int(10);
    cout << "       Address of c = " << c << endl;
    a = c;
    cout << "       Address of a = " << a << endl;
}

void setPointerPtr(intPtr *a){
    intPtr c = new int(10);
    cout << "       Address of c  = " << c << endl;
    *a = c;
    cout << "       Address of *a = " << *a << endl;
}
```

```
int main(){

    intPtr d = NULL;
    cout << "Address of d = " << d << endl;

    setPointer(d);
    cout << "Address of d = " << d << endl;

    setPointerRef(d);
    cout << "Address of d = " << d << endl;

    setPointerPtr(&d);
    cout << "Address of d = " << d << endl;

return 0;
}
```

Details

- A pointer is a variable that holds an address.
- A copy of this (pass by value) is made and sent to a function.
- The copy can reference the object or type that is associated with that address.
- We are not acting on the actual pointer we passed.

5.6 Passing Arrays

code-functions-8-pointerarrays.cc

```
#include <iostream>

// Function prototype
void copyByReference   (char [], const char []      );
void copyByPointer     (char *,  const char *       );
void copyByConstPointer(char *,  const char * const );

//-----------------------------------------------------------------------------
int main(){

    char  string1 [10];
    char *string2   = "Hello";
    char  string3[] = "Good bye";

    copyByPointer(string1,string2);
    std::cout << string1 << std::endl;

    copyByReference(string1,string3);
    std::cout << string1 << std::endl;

    copyByReference(string1,string2);
    std::cout << string1 << std::endl;

    copyByConstPointer(string1,string3);
    std::cout << string1 << std::endl;

    return 0;
}

//-----------------------------------------------------------------------------
void copyByReference(char destination[], const char source[]){
    for(int i=0; (destination[i]=source[i]) != '\0'; i++);
}

//-----------------------------------------------------------------------------
void copyByPointer(char *destination, const char * source){
    while( *destination++ = *source++);
}

//-----------------------------------------------------------------------------
void copyByConstPointer(char *destination, const char * const source){
    for(int i=0; (destination[i]=source[i]) != '\0'; i++);
}
```

Details

- Arrays are passed by reference.
- copyByReference
  - assignment.
  - value of assignment = left operand.
- copyByPointer
  - Dereference the pointer.
  - assignment.
  - increment.

  - left hand side.
- copyByConstPointer
- What is the safest way to copy the string and why?

5.7 Returning Objects

What is the best way?

- Create (stack) an object and return a reference.
- Create (new) an object and return a pointer.
  - Who deletes this object?
- Create a static object and return a reference.
- Create the object and pass it in.

code-functions-9-return.cc

```
#include <iostream>

// Function prototype
int & getReference(void);
int & getStaticReference(void);
void  setReference(int &);
int * getPointer(void);

using namespace std;

//--------------------------------------------------------------------------
int main(){

    // Return a reference to a local variable
    int  a = getReference();

    // Return a reference to a static local variable
    a = getStaticReference();

    // Pass the parameter by reference
    setReference(a);

    // Return a pointer to a variable with new
    int* b = getPointer();

    return 0;
}

//--------------------------------------------------------------------------
int & getReference(void){
    int number = 10;
    return number;  // gives a compiler warning!  Bad idea
}

//--------------------------------------------------------------------------
int & getStaticReference(void){
    static int number = 10;
    return number;
}

//--------------------------------------------------------------------------
int * getPointer(void){
    int * number = new int(10);
    return number;
}

//--------------------------------------------------------------------------
void setReference(int &number){
    number = 10;
```

```
}
```

5.8 Inline

- Use keyword `inline`
- Compiler may replace a call with actual code (making the program larger).
- Use with small functions.
- Not appropriate for large (in size) function calls, or recursive calls.
- Prototype not necessary.

code-functions-11-inline.cc

```
#include <iostream>

//-----------------------------------------------------------------------------
inline double square(const double x){
return x*x;
}

//-----------------------------------------------------------------------------
int main(){

std::cout << square(10.0)  << std::endl;
std::cout << square(100.0) << std::endl;

return 0;
}
```

5.9 Separate Files

- Functions take the work out of main().
- Multiple files extract the work from the main file.
- Why is this good?
  - Portability, organization
  - Compiled separately, then linked.
  - Modify one, don't need to necessarily recompile all.
  - Makefile
- Put declaration in header file *.h
- Keep implementation separate *.cc

code-functions-10-separateMain.cc

```
//-----------------------------------------------------------------------------
// Main()
//-----------------------------------------------------------------------------
#include <iostream>
#include "code-functions-10-separateFunc.h"

int main(){

printGreeting();

return 0;
}
```

code-functions-10-separateFunc.h

```
//---------------------------------------------------------------------------
// Header file for code-functions-10-separateFunc
//---------------------------------------------------------------------------

// Print greeting prototype
// ---------------------------------------------------------------------------
void printGreeting(void);
```

code-functions-10-separateFunc.cc

```
//---------------------------------------------------------------------------
// Implementation for code-functions-10-separateFunc
//---------------------------------------------------------------------------
#include <iostream>
#include "code-functions-10-separateFunc.h"

// Print greeting implementation
//---------------------------------------------------------------------------
void printGreeting(void){
std::cout << "Hello" << std::endl;
}
```

What goes in a header file?

- Function prototypes.
- Constants
- Structure definitions.
- Class definitions.
- Template functions.
- inline functions.

5.10 Default Arguments

- Simplify your code: often you use the same value for a parameter.
- Compiler will insert the default for you.

code-functions-20-default.cc

```
#include <iostream>

int boxVolume(int length=1, int height=1, int width=1);

int main(){

    std::cout << boxVolume()      << std::endl;
    std::cout << boxVolume(10)    << std::endl;
    std::cout << boxVolume(10,10) << std::endl;

    return 0;
}

int boxVolume(int length, int height, int width){
    return length*height*width;
}
```

- Defaults appear at the first occurrence of the name.
- Defined once.
- Where should this go? (header or declaration).
- Resolve positionally beginning with the right most parameter.

How else is this useful?

- Default output?
- You specify the file if you don't want cout.

code-functions-21-default.cc

```
#include <iostream>
#include <fstream>

using namespace std;

void display(int value, ostream &os = cout);

int main(){

    display(10);
    display(20,cout);

    fstream outFile;
    outFile.open("out", ios::out);

    display(30,outFile);

    outFile.close();

    return 0;
}

void display(int value, ostream &os){
    os << value << endl;
}
```

code-functions-22-default.cc

```
#include <iostream>
#include <fstream>

using namespace std;

void display(int value, ostream *os = NULL);

int main(){

    display(10);
    display(20,&cout);

    fstream outFile;
    outFile.open("out", ios::out);

    display(30,&outFile);

    outFile.close();

    return 0;
}

void display(int value, ostream *os){
    if(os != 0)
     (*os) << value << endl;
}
```

5.11 Overloaded functions

Why overload?

- Similar operations involving different logic.
- Book uses *min* and *max*. Is this appropriate for overloading?
- How about print?

code-functions-23-overload.cc

```
#include <iostream>

void greeting();
void greeting(const char []);
void greeting(const char [], const int);

//------------------------------------------------------------------------
int main(){

    greeting();
    greeting("John");
    greeting("John",10);

    return 0;
}

void greeting(){
    std::cout << "Hello" << std::endl;
}

void greeting(const char name[]){
    std::cout << "Hello " << name << std::endl;
}

void greeting(const char name[], const int count){
    std::cout << "Hello " << name;
    std::cout << ". You have logged in " << count << " times." << std::endl;
}
```

5.12 Templates

Why Templates?

- What if they have similar logic, different types?
- A single function can define an entire family of functions.

code-functions-24-template.cc

```
#include <iostream>

template < class T >
T maximum(T value1, T value2){
    if(value1 < value2)
        return value2;
    else
        return value1;
}

//------------------------------------------------------------------------
int main(){
```

```
    std::cout << maximum(10,5) << std::endl;

    std::cout << maximum(9.6,5.3) << std::endl;

    std::cout << maximum('a','A') << std::endl;

    return 0;
}
```

Details

- Begin with keyword `template < template parameter list >`
- Placeholder for different types.
- Compiler creates a function when it sees a type being used.
- “code generation”

5.13 Function Pointers

code-functions-25-fptr.cc

```
#include <iostream>

int max(const int i=1, const int j=1);
int min(const int i=1, const int j=1);

//-----------------------------------------------------------------------------
int main(){

    int (*fptr) (const int, const int);

    fptr = max;
    std::cout << fptr(10,5) << std::endl;

    fptr = min;
    std::cout << fptr(10,5) << std::endl;

    return 0;
}

int max(const int i, const int j){
    if(i < j) return j;
    else      return i;
}

int min(const int i, const int j){
    if(i > j) return j;
    else      return i;
}
```

6 Objects-Based Programming

Attributes

- Data members – class representation.
- Member functions – class interface.
- Levels of access – Public, private, protected.
- Type specifier – tag name.

Abstract data type

- Private representation and public operations

6.1 Definition

code-classes-1-definition.cc

```
class Rational{};
```

or

```
class Rational{} numberOne, numberTwo;  // Ok also
```

6.2 Data members

code-classes-2-datamembers.cc

```
class Rational{
    int numerator;
    int denominator;
    // int numerator,denominator;   // This is ok also
};
```

or

```
class Rational{
  private:
    int numerator;
    int denominator;
};
```

Style

- Private unless explicitly defined.
- References and pointers to its own type.

6.3 Initializing Data Members

code-classes-3-initializing.cc

```
class Rational{
  public:
    int numerator;
    int denominator;
};

int main(){

    // Cannot do this if data members are private
    Rational num  = {5,1};
```

```
    return 0;
}
```

- If not public, cannot directly access and initialize.
- Initialize when the object is created.
- Constructor is a function that is automatically invoked when an object is created.
- **General Rule**: Make sure that all constructors initialize everything in the object.

6.4 Constructors

code-classes-4-constructors.cc

```
class Rational{
  public:
    Rational();
    Rational(const int);
    Rational(const int, const int);

  private:
    int numerator;
    int denominator;
};

Rational::Rational(){
    numerator   = 0;
    denominator = 1;
}

Rational::Rational(const int n){
    numerator   = n;
    denominator = 1;
}

Rational::Rational(const int n, const int d){
    numerator   = n;
    denominator = d;
}

int main(){

    Rational n1;       // n1 = 0
    Rational n2(10);   // n2 = 10
    Rational n3(1,5);  // n3 = 1/5

    return 0;
}
```

Details

- Constructor name is the same as the class name.
- No return value and is not declared `void`.

Two methods for multiple constructors.

- **Overloaded** - you can have multiple constructors.
- **Default values** - do this with defaults....

  code-classes-5-defaultvalues.cc

```
class Rational{
  public:
    Rational(const int n=0, const int d=1);

  private:
    int numerator;
    int denominator;
};

Rational::Rational(const int n, const int d){
    numerator   = n;
    denominator = d;
}

int main(){

    Rational n1;       // n1 = 0
    Rational n2(10);   // n2 = 10
    Rational n3(1,5);  // n3 = 1/5

    return 0;
}
```

Default constructor

- Automatically provided if you don't create one.
- It looks like: `Rational::Rational(){};`
- If you provide *any* constructor, the compiler will not give you one.

6.5 Initialization list

code-classes-6-initialization.cc

```
//------------------------------------------------------------------------------
class Rational{
  public:
    Rational(const int n=0, const int d=1);

  private:
    int numerator;
    int denominator;
};

Rational::Rational(const int n, const int d) : numerator(n), denominator(d){}

//------------------------------------------------------------------------------
class Simple{
  public:
    Simple(const Rational&, const int);

  private:
    Rational  num;
    const int constInt;
};

Simple::Simple(const Rational &n, const int A): num(n), constInt(A)  {}

//------------------------------------------------------------------------------
int main(){

    Rational n1(1,5);  // n3 = 1/5
```

```
    Simple   s1(n1,10);

    return 0;
}
```

Details

- Follows signature of constructor.
- Cannot be specified in declaration.
- **Only** way to initialize const and reference data members.

Efficiency

- Assignment: first calls the default constructor to initialize members and then assigns new values.
- Initialization avoids this problem.
- Does not apply for built in types.

code-classes-6-segment.cc

```
Rational::Rational(const int n, const int d) : numerator(0), denominator(1){
    numerator   = n;
    denominator = d;
}
```

6.6 Default Destructors

Details

- Invoked when an object is destroyed.
- Decision of when the destructor is called is left up to the compiler.
- **Goal**: Clean up the debris created by constructor.

code-classes-6-destructors.cc

```
#include <iostream>

//-----------------------------------------------------------------------------
class Rational{
  public:
    Rational(const int n=1, const int d=1);
    ~Rational();

  private:
    int numerator;
    int denominator;
};

Rational::Rational(const int n, const int d){
    numerator   = n;
    denominator = d;
    std::cout << "Constructor called: " << n << "/" << d << std::endl;
}

Rational::~Rational(){
    std::cout << "Destructor called" << std::endl;
}

//-----------------------------------------------------------------------------
int main(){

    Rational n1;       // n1 = 1
    Rational n2(10);   // n2 = 10
    Rational n3(1,5);  // n3 = 1/5

    return 0;
}
```

6.7 Member functions

code-classes-7-memberfunctions.cc

```
#include <iostream>

//-----------------------------------------------------------------------------
class Rational{
  public:
    Rational(const int n=1, const int d=1);
    ~Rational(){};

    void print();

  private:
    int numerator;
    int denominator;
};
```

```
Rational::Rational(const int n, const int d){
    numerator   = n;
    denominator = d;
}

void Rational::print(){
    std::cout << numerator << "/" << denominator << std::endl;
}

//------------------------------------------------------------------------------
int main(){

    Rational n1;       // n1 = 1
    Rational n2(10);   // n2 = 10
    Rational n3(1,5);  // n3 = 1/5

    n1.print();
    n2.print();
    n3.print();

    return 0;
}
```

Details

- Declared in class body.
- Consists of the prototype.
- Definition can be inside the class – *inline*.
- Most functions defined outside the class.
- Need to use the :: scope resolution operator when defining member functions.

Member functions

- Have full access to both private and public data and methods.
- Defined within the scope of their class.
- Ordinary functions have file scope.

Information Hiding

- Formal mechanism for restricting user access to internal representation.

6.8 Creating Class Objects

code-classes-8-objects.cc

```
#include <iostream>

//------------------------------------------------------------------------------
class Rational{
  public:
    Rational(const int n=1, const int d=1);
    void print();

  private:
    int numerator;
    int denominator;
};

Rational::Rational(const int n, const int d){
    numerator   = n;
    denominator = d;
}
```

```
void Rational::print(){
    std::cout << numerator << "/" << denominator << std::endl;
}

//------------------------------------------------------------------------------
int main(){

    Rational  n1(1,5);
    Rational  n2(n1);
    // Same as assigment.
    n2 = n1;

    // Pointers
    Rational* n3 = new Rational(10);
    Rational* n4 = &n2;
    n2 = *n3;

    // Reference
    Rational& r5 = n1;

    n1.print();
    n3->print();

    // Delete memory.  What about n4?
    delete n3;

    return 0;
}
```

- Allocates storage sufficient to contain two data members.
- Pointers, references...

6.9 C++ Silent Functions

code-classes-8-silent.cc

```
// We declare this class....
// ----------------------------------------------------------------------------
class Rational{
  public:
    Rational(const int n=1, const int d=1);

  private:
    int numerator;
    int denominator;
};

// We actually get this class....
// ----------------------------------------------------------------------------
class Rational{
  public:
    Rational(const int n=1, const int d=1);

    Rational(const Rational &);                // A copy constructor
    ~Rational();                               // A destructor

    Rational & operator=(const Rational&);     // Copy assignment operator

  private:
    int numerator;
    int denominator;
};
```

- Defaults:
  - Default constructor.
  - Default Copy constructor.
  - Default assignment operator.
  - Default destructor.
  - Default address operator.
- Copy – copying all its elements.
- We will revisit this.

6.10 Overloading and *this* Motivation

code-classes-9-overload.cc

```
#include <iostream>

using namespace std;

//----------------------------------------------------------------------------
class Rational{
  public:
    Rational(const int n=1, const int d=1);
    void add(const Rational &);
    void print();

  private:
```

```
    int numerator;
    int denominator;
};

Rational::Rational(const int n, const int d){
    numerator   = n;
    denominator = d;
}

void Rational::print(){
    cout << numerator << " " << denominator << endl;
}

void Rational::add(const Rational &B){
    numerator = numerator*B.denominator + denominator*B.numerator;
    denominator = denominator*B.denominator;
}

//------------------------------------------------------------------------------
int main(){

    Rational  n1(1,5);
    Rational  n2(2,3);
    Rational  n3(3);
    n1.add(n2);
    n1.print();
    n1.add(n3);
    n1.print();

    return 0;
}
```

Verbose and Overload

- Methods must occur as separate statements.
- Unnecessarily verbose.
- Prefer concatenation of member function calls.
- Make your code look more natural.

6.11 The Implicit *this*

code-classes-10-this.cc

```
#include <iostream>

using namespace std;

//------------------------------------------------------------------------------
class Rational{
  public:
    Rational(const int n=1, const int d=1);
    Rational & add(const Rational &);
    Rational & print();

  private:
    int numerator;
    int denominator;
};

Rational::Rational(const int n, const int d){
    numerator   = n;
```

```
    denominator = d;
}

Rational & Rational::print(){
    cout << numerator << " " << denominator << endl;
    return *this;
}

Rational & Rational::add(const Rational &B){
    numerator = numerator*B.denominator + denominator*B.numerator;
    denominator = denominator*B.denominator;
    return *this;
}

//------------------------------------------------------------------------
int main(){

    Rational  n1(1,5);
    Rational  n2(2,3);
    Rational  n3(3);
    n1.add(n2).print().add(n3).print();

    return 0;
}
```

Implicit this pointer

- Each member function contains a pointer of its class type named *this*.
- Contains the address of the class object through which the member function has been invoked.
- Return the class object that has been invoked.
- It is possible to overwrite the this pointer (e.g. resize, linked-list).
- Returning a reference to **this* allows methods to be chained together.

6.12 Overloading the Addition Operator

code-classes-11-addition.cc

```
#include <iostream>

using namespace std;

//------------------------------------------------------------------------
class Rational{
  public:
    Rational(const int n=1, const int d=1);
    Rational & print();

    // Let's try this
    Rational & operator+(const Rational &);

  private:
    int numerator;
    int denominator;
};

Rational::Rational(const int n, const int d): numerator(n), denominator(d){};

Rational & Rational::print(){
    cout << numerator << " " << denominator << endl;
    return *this;
```

```
}

Rational & Rational::operator+(const Rational &B){
    numerator = numerator*B.denominator + denominator*B.numerator;
    denominator = denominator*B.denominator;
    return *this;
}

//------------------------------------------------------------------------------
int main(){

    Rational  n1(1,5);
    Rational  n2(2,3);

    // This should work...
    n1 = n2 + n1;
    n1.print();

    // What about this?
    n1 = n1 + 3;
    n1.print();

    return 0;
}
```

Discussion

- There is an *implicit type conversion*.
- The `Rational(3)` constructor is called.
- But, there is not function associated with `int  3`.
- `explicit` would disallow this conversion.

What is the value of `n2`?

- This is actually behaving like a += operator!

6.13 Non-Member functions and Friends

Non-member functions

- Pass two arguments, return a constant reference.
- Must be a non-member function.
- How can we access private members?

Friends of a class

- Information hiding is too prohibitive.
- Friend mechanism gives nonmember functions of a class access to the nonpublic members of a class.
- Left operand of every member function is an object or pointer to an object of its class.
- This is confusing....
- But, a nonmember function does not have access to nonmembers.
- We may not want to provide a member function to this information.

code-classes-12-friend.cc

```
#include <iostream>

using namespace std;

//------------------------------------------------------------------------------
class Rational{
```

```
  friend
    Rational & operator+(const Rational& ,const Rational &);

  public:
    Rational(const int n=1, const int d=1);
    Rational & print();

  private:
    int numerator;
    int denominator;
};

Rational::Rational(const int n, const int d): numerator(n), denominator(d){};

Rational & Rational::print(){
    cout << numerator << " " << denominator << endl;
    return *this;
}

// WARNING.... BAD CODE!!!
Rational & operator+(const Rational &A, const Rational &B){
    int num = A.numerator*B.denominator + A.denominator*B.numerator;
    int den = A.denominator*B.denominator;
    Rational result(num,den);
    return result;
}

//--------------------------------------------------------------------------
int main(){

    Rational  n1(1,5);
    Rational  n2(1,3);
    n1 = n2 + n1;
    n1.print();
    n2.print();

    return 0;
}
```

Heap-based

- Who will delete the pointer?
- What about: `x*y*z;` ?

Static Rational object

- Comparing two results will always be the same!

6.14 The correct way to add

code-classes-15-friend.cc

```
#include <iostream>

using namespace std;

//--------------------------------------------------------------------------
class Rational{

  friend
    const Rational operator+(const Rational& ,const Rational &);

  public:
```

```
    Rational(const int n=1, const int d=1);
    Rational & print();

  private:
    int numerator;
    int denominator;
};

Rational::Rational(const int n, const int d): numerator(n), denominator(d){};

Rational & Rational::print(){
    cout << numerator << " " << denominator << endl;
    return *this;
}

const Rational operator+(const Rational &A, const Rational &B){
    int num = A.numerator*B.denominator + A.denominator*B.numerator;
    int den = A.denominator*B.denominator;
    return Rational(num,den);
}

//------------------------------------------------------------------------
int main(){

    Rational  a(1,5);
    Rational  b(1,3);
    Rational  c(10);

    c = a + b;
    a.print();
    b.print();
    c.print();

    c = 3 + b;
    a.print();
    b.print();
    c.print();

    return 0;
}
```

Notice.

- Returning an object.
- Implicit conversion for `3 + a` is fixed.

Guidelines

- Declare non-member functions when type conversions should apply to all parameters.
- Don't pass a reference when you must return an object.

6.15 Overloading $<<$

- Consider this statement: `cout << n1 << endl;`.
  - Concatenation: a reference to ostream must be returned.
  - Left operand must be `cout`.
- Non-member friend function!

code-classes-13-output.cc

```
#include <iostream>

using namespace std;

//-------------------------------------------------------------------------------
class Rational{

    friend ostream  & operator<<(ostream &, const Rational &);

  public:
    Rational(const int n=1, const int d=1);

  private:
    int numerator;
    int denominator;
};

Rational::Rational(const int n, const int d){
    numerator   = n;
    denominator = d;
}

ostream & operator<<(ostream & os, const Rational &A){
   os << A.numerator << "/" << A.denominator;
   return os;
}

//-------------------------------------------------------------------------------
int main(){

    Rational  n1(1,5);
    cout << n1 << endl;

    return 0;
}
```

6.16 Const

code-classes-14-const.cc

```
#include <iostream>

using namespace std;

//-------------------------------------------------------------------------
class Rational{

  friend
    ostream  & operator<<(ostream &, const Rational &);

  public:
    Rational(const int n=1, const int d=1);
    bool operator==(const Rational &) const;

  private:
    int numerator;
    int denominator;
};

Rational::Rational(const int n, const int d){
    numerator   = n;
    denominator = d;
}

ostream & operator<<(ostream & os, const Rational &A){
   os << A.numerator << "/" << A.denominator;
   return os;
}

bool Rational::operator==(const Rational &A) const{
    if( numerator == A.numerator && denominator == A.denominator )
        return true;
    else
        return false;
}


//-------------------------------------------------------------------------
int main(){

    Rational        n1(1,5);
    const Rational  n2(2,3);
    n1 =  n2;
    if( n1 == n2 )
        cout << n1 <<  " and " << n2 << " are equal" << endl;

    if( n2 == n1 )
        cout << n1 <<  " and " << n2 << " are equal" << endl;
    /* // Does not compile....
    */

    return 0;
}
```

Const

- Class designer indicates which member functions are safe by specifying them as const.
- Protects a const class object.

- Only functions specified as const can be invoked by a const class object.
- Const functions can be overloaded with nonconst instance that defines the same signature.
- Constructors and destructors are the exception.

6.17 Mutable

code-classes-16-mutable.cc

```
#include <iostream>
#include <string>

using namespace std;

//---------------------------------------------------------------------------
class Employee{

  public:
      Employee(const string &n, const double s=0.0) : name(n), salary(s){};

      // const methods
      string getName()   const {return name;}
      double getSalary() const {return salary;}
      void   print()     const;

      // Non-const methods
      void changeEmployee (const string &n, const double s);
      void   changeSalary (const double s);

  private:
      string name;
      double salary;
};

inline void Employee::print() const{
        cout << name << " earns $" << salary << endl;
}

void Employee::changeSalary(const double s){
    salary = s;
}

void Employee::changeEmployee(const string &n, const double s){
    name   = n;
    salary = s;
}


//---------------------------------------------------------------------------
int main(){

    Employee temp("Jim",100.0);
    temp.print();

    temp.changeEmployee("Mary",100.0);
    temp.print();
    temp.changeSalary(200.0);
    temp.print();

    const Employee John("John",100.0);
    John.print();
    //John.changeSalary(200.0);
    John.print();

    return 0;
}
```

Solution

```
class Employee{

  public:
      Employee(const string &n, const double s=0.0) : name(n), salary(s){};

      // const methods
      string getName()   const {return name;}
      double getSalary() const {return salary;}
      void   print()     const;
      void   changeSalary (const double s) const;

      // Non-const methods
      void changeEmployee (const string &n, const double s);

  private:
      string name;
      mutable double salary;
};
```

6.18 Type Conversion Functions

code-classes-3-conversion.cc

```
#include <iostream>

using namespace std;

class Rational{

  public:
    Rational(const int n=0, const int d=1): num(n), den(d) {};
    double real(){ return (double) num/den;}
    operator double() const;

  private:
    int num,den;
};

Rational::operator double () const{
    return (double) num/den;
}

int main(){

    Rational n1(1,3);

    // Conversion using constructor.
    n1 = 5;

    // Using a conversion function.
    double r1 = n1;

    // Explicit conversion
    Rational n2(1,4);
    double r2 = n2.real();

return 0;
}
```

6.19 Static Members and Functions

code-classes-17-static.cc

```
#include <iostream>

//-------------------------------------------------------------------------
class WebBrowser{
  public:
    WebBrowser();           // Constructor
    ~WebBrowser();          // Destructor
    static int getCount();

  private:
    static int count;
};

// Define and initialize static data member and function.
int WebBrowser::count = 0;
int WebBrowser::getCount(){ return count; }

// Constructor and destructor.
WebBrowser::WebBrowser() { ++count; }
WebBrowser::~WebBrowser(){ --count; }

//-------------------------------------------------------------------------
int main(){

    std::cout <<  "Number of Browsers open: ";
    std::cout << WebBrowser::getCount() << std::endl;

    WebBrowser *w1 = new WebBrowser();
    WebBrowser *w2 = new WebBrowser();

    std::cout <<  "Number of Browsers open: ";
    std::cout << WebBrowser::getCount() << std::endl;

    delete w1;
    delete w2;

    std::cout <<  "Number of Browsers open: ";
    std::cout << WebBrowser::getCount() << std::endl;

    return 0;
}
```

6.20 Dynamic Data Members – String Class

code-classes-19-string.cc

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

//-------------------------------------------------------------------------
class String{

  friend
    ostream &operator<<(ostream& os, const String& s);

  public:

    String(const char* s = "");                // Constructor with default
```

```
    String(const String&);                      // Copy Constructor
    ~String();                                  // Destructor
    const String &operator=(const String &);    // Assignment operator

  private:
    char * strPtr;                              // the 'string'
};

//---------------------------------------------------------------------------
ostream &operator<<(ostream &os, const String &s){
    os << s.strPtr;
    return os;
}

//---------------------------------------------------------------------------
int main(){

    //--------- Review of character arrays  --------------------------------

    char p1 [] = "This is a long character array";
    int length = strlen(p1);

    char *p2 = new char[length+1];

    strcpy(p2,p1);
    cout << p1 << endl;
    cout << p2 << endl;

    //---------------------------------------------------------------------

    // Create the strings
    String s1;
    s1 = "Hello";
    String s2("Brad");
    cout << s1 << " " << s2 << endl;

    // Swap them
    String tmp = s1 = s2;
    cout << tmp << " " << s1 << endl;
    s1 = s2;
    s2 = tmp;
    cout << s1 << " " << s2 << endl;

    return 0;
}
```

6.21 BONUS: Overloading ++

code-classes-2-plusplus.cc

```
#include <iostream>

using namespace std;

class Rational{
  friend
    ostream & operator<<(ostream &, const Rational &);

  public:
    Rational(const int n=0, const int d=1): num(n), den(d) {};

    Rational& operator++();   // prefix increment
    Rational  operator++(int); // postfix increment

  private:
    int num,den;
};

ostream & operator<<(ostream &os, const Rational &r){
    os << r.num << "/" << r.den;
    return os;
}

Rational & Rational::operator++(){
    num = num + den;
    return *this;
}

Rational Rational::operator++(int){
    Rational temp = *this;
    num = num + den;
    return temp;
}

int main(){

    Rational n1(1,3);
    n1 = 3;
    cout << n1++ << endl;
    cout << n1   << endl;

    n1 = 3;
    cout << ++n1 << endl;
    cout << n1   << endl;

    return 0;
}
```

7 Inheritance

7.1 Simple Public

Why inheritance?

- Reuse existing (well-tested) code.
- Eliminate duplicate code – fix one method only.

Alternatives?

- Copy and paste.
- What if you do not have the source code?

Relationships

- *is-a*:
  - A student *is-a* person.
  - A square *is-a* shape.
- *has-a*:
  - A car *has-a* tire.
  - Implement through composition.
- *is-implemented-in-terms-of*:

Example

- A square *is-a* rectangle.
  - What if rectangle has a `changeHeight()` method?
- A penguin *is-a* bird.
  - Does a `fly()` method makes sense here?

code-inheritance-1-public.cc

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

class Person{
  public:
    void eat(){ cout << "Eating" << endl;}
};

class Student: public Person{
  public:
    void study(){cout << "Studying" << endl;}
};

int main(){

    Person  p;
    Student s;
    p.eat();
    s.eat();
    s.study();
    //p.study();    // Compile error

    return 0;
}
```

Key points

- `: public` specifies public inheritance.
- Student inherits eat() from person.
- Other inherited classes all use the same implementation of eat().
  - copy and paste is error prone.

7.2 Creation order

code-inheritance-2-order.cc

```
#include <iostream>

using namespace std;

class Person{
  public:
    Person() { cout << "Person Constructor" << endl;}
    ~Person(){ cout << "Person Destructor"  << endl;}
    void eat(){ cout << "Eating" << endl;}
};

class Student: public Person{
  public:
    Student() { cout << "Student Constructor" << endl;}
    ~Student(){ cout << "Student Destructor"  << endl;}
    void study(){cout << "Studying" << endl;}
};

int main(){

    Person  p;
    Student s;
    p.eat();
    s.eat();

    return 0;
}
```

Points

- Base object is created first.
- Derived object makes an implicit call to the base objects constructor and destructor.

7.3 Base classes without default constructors

code-inheritance-3-problem.cc

```
#include <iostream>
#include <string>

using namespace std;

//-------------------------------------------------------------------------
class Person{
  public:
    Person(string n): name(n){};
    void eat(){ cout << name << " is eating" << endl;}
    string getName(){ return name;}

  private:
    string name;
};

//-------------------------------------------------------------------------
class Student: public Person{
  public:
    Student(){};
    void study(){cout << this->getName() << " is studying" << endl;}
};

//-------------------------------------------------------------------------
int main(){  // DOES NOT COMPILE!

    Person  p("Tom");
    p.eat();

    Student s;
    s.study();

    return 0;
}
```

- Student tries to call default constructor.
- Student uses `getName()` because name is private.
- How do we fix this?
  - Add a default constructor.
  - Place a call to constructor in initialization list.

7.4 Creating objects

code-inheritance-4-using.cc

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

//-------------------------------------------------------------------------
class Person{
  public:
    Person(string n): name(n){};
    Person(const Person &p){
        this->name = p.name;
        cout << name << " copy Constructor called" << endl;
```

```
    }

    const Person& operator=(const Person& p){
        this->name = p.name;
        cout << name << " assignement called" << endl;
    }

    void eat(){ cout << name << " is eating" << endl;}
    string getName(){ return name;}

  private:
    string name;
};

//------------------------------------------------------------------------------
class Student: public Person{
  public:
    Student(string n):Person(n){};
    void study(){cout << this->getName() << " is studying" << endl;}
};

//------------------------------------------------------------------------------
int main(){

    // Pointers and references
    Student s1("Kathy");
    Person  &p1 = s1;
    Person  *p2 = &s1;
    p1.eat();
    p2->eat();
    s1.study();

    //Calls to copy and assignment
    Person  p3(s1);
    Student s2("Jim");
    p3 = s2;
    return 0;
}
```

- What?

7.5 Overloading Copy and Assignment

code-inheritance-5-copy.cc

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

//------------------------------------------------------------------------------
class Person{
  public:
    Person(string n=""): name(n){};
    string getName(){ return name;}

  private:
    string name;
};

//------------------------------------------------------------------------------
class Student: public Person{
```

```
  public:
    // Constructor
    Student(string n, double g): Person(n), grades(g){};

    // Copy Constructor
    Student(const Student& s){
        grades = s.grades;
    };

    // Assignment operator
    const Student& operator=(const Student& s){
        grades = s.grades;
        return *this;
    }

    // Print...
    double print(){
        cout << this->getName() << " gets a " << grades << "%" << endl;
    };
  private:
    double grades;
};

//------------------------------------------------------------------------------
int main(){

    Student s1("Britney",90.1);
    Student s2("Justin", 87.2);

    s1.print();
    s2.print();

    s1 = s2;
    s1.print();
    Student s3(s1);
    s3.print();

    return 0;
}
```

7.6 Overriding

code-inheritance-6-overriding.cc

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

class Base{
  public:
    void method1(){ cout << "Base method1()" << endl;}
};

class Derived: public Base{
  public:
    void method1(){ cout << "Derived method1()" << endl;}
};

int main(){

    Derived d;
    d.method1();
```

```
    return 0;
}
```

7.7 Hiding names

code-inheritance-7-problem.cc

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

class Base{
  public:
    void method1()    { cout << "Base method1()" << endl;}
    void method1(int) { cout << "Base method1(int)" << endl;}
};

class Derived: public Base{
  public:
    void method1(){ cout << "Derived method1()" << endl;}
};

int main(){

    Derived d;
    d.method1();
    //d.method1(8); //Produces an error
    return 0;
}
```

7.8 Using declaration

```
class Derived: public Base{
  public:
    // makes all things in Base named method1 visible and public
    // in Derived's scope
    using Base::method1;

    void method1(){ cout << "Derived method1()" << endl;}
};
```

7.9 Scope resolution

code-inheritance-9-selective.cc

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

class Base{
  public:
    void method1()    { cout << "Base method1()"     << endl;}
    void method1(int) { cout << "Base method1(int)"  << endl;}
    void method1(char){ cout << "Base method1(char)" << endl;}
};

class Derived: public Base{
  public:
    void method1()     { cout << "Derived method1()" << endl;}
    void method1(int i){ Base::method1(i);}
};
```

```
int main(){

    Derived d;
    d.method1();        // Calls method1 in Derived class
    d.method1(8);       // Calls method1 in Base class
    //d.method1('a');   // ERROR!
    return 0;
}
```

Discussion

- Public inheritance means *is-a*.
  - Everything that applies to `B` also applies to object `D`.
  - Classes derived from `B` should inherit both interface and implementation.
- If `D` redefines a member function in `B` → contradiction.
  - If the implementation really needs to be different, then not every `D` *is-a* `B`.
- If the member function really varies, then it should be `virtual`.

Message

- Be careful when overriding an inherited non-virtual function.

7.10 Private Inheritance

- Does not mean *is-a*
- Does not convert a derived class object into a base class object.
- Inherited members become private.
- Means *is-implemented-in-terms-of*.
- Take advantage of the features in B, not that there is a conceptual relationship.
- Use composition when you can and and private inheritance when you must.

code-inheritance-11-private.cc

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

class Person{
  public:
    void eat(){ cout << "Eating" << endl;}
};

class Student: private Person{
  public:
    void study(){ cout << "Studying" << endl; }
    void internalSnack(){ Person::eat();}
};

int main(){

    Person  p;
    Student s;
    p.eat();
    s.study();
    s.internalSnack();
    //s.eat(); //Error - Eat is a private method now
    return 0;
}
```

- See Figure 12.27 for summary of Public, Private, Protected.

7.11 String Class Example

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

void compare(const string &a, const string &b){
    if( a == b )
        cout << a << " == " << b << endl;
    else
        cout << a << " != " << b << endl;
}

int main(){

    string s1,s2;

    s1 = "Steve";
    s2 = "StEve";
    compare(s1,s2);

    s1 = "Steve's";
    compare(s1,s2);

    return 0;
}
```

Output

```
Steve != StEve
Steve's != StEve
```

Question:

- How can we resolve this issue?

Non-Member function

- Overload the compare operator.
- Can we access string class elements — Yes!

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

void compare   (const string&, const string&);
bool operator==(const string&, const string&);

int main(){

    string s1,s2;

    s1 = "Steve";
    s2 = "StEve";
    compare(s1,s2);

    s1 = "Steve's";
    compare(s1,s2);

    return 0;
}

//-------------------------------------------------------------------------
void compare(const string &a, const string &b){
    if( a == b )
        cout << a << " == " << b << endl;
    else
        cout << a << " != " << b << endl;
}

//-------------------------------------------------------------------------
bool operator==(const string &a, const string &b){
    if( a.length() != b.length())
        return false;
    else{
        int l=0;
        while(l < a.length()){
            if( tolower(a[l]) != tolower(b[l]) ){
                return false;
            }
            l++;
        }
    }
    return true;
}
```

Inheritance Take-home Quiz

Task

- Write a class called `word` that inherits from the base class `string`.
- Override the compare operator as a member function of the `word` class.
- Make sure your implementation works with the following main program.

```
#include <iostream>
#include "word.h"
using namespace std;

//-------------------------------------------------------------------------------
void compare(const word &a, const word &b){
        if( a == b )
                cout << a << " == " << b << endl;
        else
                cout << a << " != " << b << endl;
}

int main(){

    word s3;
    word s1("Steve");
    word s2 = "StEve";
    s3 = "Steve";

    compare(s1,s2);
    compare(s1,s3);

    return 0;
}
```

Output

```
Steve == StEve
Steve == Steve
```

Here is some code to help you get started.
*w*ord.h

```
#ifndef WORD_H
#define WORD_H
#include <string>
using namespace std;

class word : public string{
  public:
    // Your code goes here
  private:
    // I did not need a private section
};

#endif
```

*w*ord.cc

```
#include <iostream>
#include "word.h"
using namespace std;
```

(over)

What do I turn in?

- On a linux machine, type:

```
$cat word.h word.cc | a2ps -o word.ps
$lpr -P<PRINTERNAME> word.ps
```

- Hand in the `word.ps`.

8 Polymorphism

8.1 Java

Shapes.java

```
// Polymorphism in Java.

//-----------------------------------------------------------------------------
abstract class Shape {
  abstract public void draw();
  public void area(){ System.out.println("Shape.area()");}
}

//-----------------------------------------------------------------------------
class Square extends Shape {
  public void draw() { System.out.println("Square.draw()"); }
  public void area() { System.out.println("Square.area()"); }
}

//-----------------------------------------------------------------------------
class Triangle extends Shape {
  public void draw() {
    System.out.println("Triangle.draw()");
  }
}

//-----------------------------------------------------------------------------
//  Main()
//-----------------------------------------------------------------------------
public class Shapes {
  public static void main(String[] args) {

            Shape[] s = new Shape[2];
            s[0] = new Square();
            s[1] = new Triangle();

            s[0].draw();
            s[0].area();

            s[1].draw();
            s[1].area();
  }
}
```

Discussion points

- What is the output?
- Shape is an abstract class — what does that mean?
- draw() vs. area()?
- Triangle and Square are shapes!

Changes

- What happens if shape is no longer abstract?
- Can we add a shape object to the array?

What is polymorphism?

- A *polymorphic* variable is a variable for which the static type, the type associated with a declaration, may differ from the dynamic type, the type associated with the value currently being held by the variable.
- The methods of a polymorphic variable may change their behavior at run time.

8.2 Package Example

You are a developer for the **Pack-N-ship** store. You need to write a program that addresses the following store requirements.

- There are three types of packages: Ground, TwoDay, and NextDay.
  - The cost of ground shipping is: `weight*rate`, where `rate` is $1.5/`weight`.
  - The cost of TwoDay is $1.5 \cdot$`weight*rate`.
  - The cost of NextDay is `weight*rate` $+ \$10$.
- Each package contains a `name` and a package `weight`.
- The `cost()` of shipping a package is based on the type of package.
- Ideally, you would like to overload the ostream output operator `<<` such that all package information will be printed to an ostream operator. This makes printing labels easier.
- You would also like to store each package created in a container (e.g. array) that can be passed to various functions at the end of the day for processing all the packages shipped.

What is the best way to proceed?

8.3 Using Inheritance

code-poly-1-inheritance.cc

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

//-----------------------------------------------------------------------------
class Package{
  public:
    Package(const string n,
            const double w,
            const string m): name(n), weight(w), method(m) {};

    double getCost()   const{ return weight*rate; }
    string getName()   const{ return name; }
    string getMethod() const{ return method;}

  private:
    string name;
    string method;
    double weight;
    static const double rate = 1.5;
};


//-----------------------------------------------------------------------------
class TwoDay : public Package{
  public:
    TwoDay(const string s, const double w, const string m): Package(s,w,m) {};
    double getCost() const{ return 1.5*Package::getCost();}
};

//-----------------------------------------------------------------------------
int main(){

    Package* p1 = new Package("Jim",10,"Ground");
    TwoDay*  p2 = new TwoDay ("Tom",10,"TwoDay");

    cout << p1->getName() << "'s package" << endl;
    cout << p1->getMethod() << endl;
    cout << p1->getCost() << endl << endl;

    cout << p2->getName() << "'s package" << endl;
    cout << p2->getMethod() << endl;
    cout << p2->getCost() << endl << endl;

    delete p1;
    p1 = p2;
    cout << p1->getName() << "'s package" << endl;
    cout << p1->getMethod() << endl;
    cout << p1->getCost() << endl << endl;

    return 0;
}
```

Important: The output of the `getCost()` method depends on the type of handle used to invoke the function, not the type of object to which it points.

Discussion

- It is an error to attempt to call a derived method from a base pointer.
- Java?

8.4 Overloading ostream

code-poly-2-inheritance.cc

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

//-----------------------------------------------------------------------------
class Package{
  friend
    ostream & operator<<(ostream &os, const Package &p);

  public:
    Package(const string n,
            const double w,
            const string m): name(n), weight(w), method(m) {};

    double cost() const { return weight*rate; }

  private:
    string name;
    string method;
    double weight;
    static const double rate = 1.5;
};

//-----------------------------------------------------------------------------
ostream & operator<<(ostream &os, const Package &p){
    os << " Customer: " << p.name   << endl;
    os << " Weight:   " << p.weight << endl;
    os << " Method:   " << p.method << endl;
    os << " Cost:    $" << p.cost() << endl;
    return os;
}

//-----------------------------------------------------------------------------
class TwoDay : public Package{
  public:
    TwoDay(const string s, const double w, const string m): Package(s,w,m) {};
    double cost() const{ return 1.5*Package::cost();}
};

//-----------------------------------------------------------------------------
int main(){

    Package* p1 = new Package("Jim",10,"Ground");
    TwoDay*  p2 = new TwoDay ("Tom",10,"TwoDay");

    cout << "Jim's cost = $" << p1->cost() << endl;
    cout << "Tom's cost = $" << p2->cost() << endl;
    cout << endl;

    cout << *p1 << endl;
    cout << *p2 << endl;  // We are casting p2 from TwoDay --> Package

    return 0;
}
```

8.5 Virtual Functions

code-poly-3-virtual.cc

```
class Package{
  friend
    ostream & operator<<(ostream &os, const Package &p);

  public:
    Package(const string n,
            const double w,
            const string m): name(n), weight(w), method(m) {};

    virtual double cost() const { return weight*rate; }

  private:
    string name;
    string method;
    double weight;
    static const double rate = 1.5;
};
```

Details

- The Package pointer invokes the Package cost() method even though the pointer is aimed at the TwoDay object.
- With virtual functions, the type of object being pointed to, not the type of the handle, determines which version of the virtual function is being called.
- Dynamically determines the behavior of the Package pointer.
- Make the `cost()` method `virtual`.

8.6 Abstract Classes

code-poly-4-virtual.cc

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

//-------------------------------------------------------------------------
//  Package (Abstract) class
//-------------------------------------------------------------------------
class Package{
  friend
    ostream & operator<<(ostream &os, const Package &p);

  public:
    Package(const string n, const double w, const string m):
            name(n), weight(w), rate(1.5), method(m) {};
    double baseCost() const { return weight*rate; }

    virtual double cost() const = 0;        // Pure virtual --- abstract

  private:
    string name;
    string method;
    double weight;
    double rate;
};

ostream & operator<<(ostream &os, const Package &p){
    os << " Customer: " << p.name   << endl;
    os << " Weight:   " << p.weight << endl;
```

```
    os << " Method:    " << p.method << endl;
    os << " Cost:     $" << p.cost() << endl;
    os << endl;
    return os;
}

//-------------------------------------------------------------------------
//  Types of packages
//-------------------------------------------------------------------------
class Ground : public Package{
  public:
    Ground(const string s="", const double w=0): Package(s,w,"Ground") {};
    double cost() const{ return Package::baseCost();}
};

class TwoDay : public Package{
  public:
    TwoDay(const string s="", const double w=0): Package(s,w,"TwoDay") {};
    double cost() const{ return 1.5*Package::baseCost();}
};

class Overnight : public Package{
  public:
    Overnight(const string s="", const double w=0): Package(s,w,"Overnight") {};
    double cost() const{ return Package::baseCost() + 10.0;}
};

//-------------------------------------------------------------------------
int main(){

    Package* data [5];

    data[0] = new Overnight("Jim",    10);
    data[1] = new TwoDay   ("Karen", 10);
    data[2] = new Ground    ("Matt",  10);
    data[3] = new Ground    ("James",  5);
    data[4] = new Ground    ("Dillon",20);

    double total = 0.0;
    for(int i=0; i<5; ++i){
        cout << *data[i] << endl;
        total += data[i]->cost();
        delete data[i];
    }

    cout << "Total $" << total << endl;

    return 0;
}
```

8.7 Another View of Inheritance

Public inheritance

- Member function interfaces are always inherited.
- `pure virtual` – A derived class inherits the function interface *only*.
  - Must be redefined in a concrete class.
  - Have no definition.
- `simple virtual` – A derived class inherits the interface and a *default* implementation.
  - Provides an implementation that a derived class may override.
- `non-virtual` – A function that is *invariant over specialization*. Behavior that is not supposed to change.
  - A derived class inherits a function interface as well as a mandatory implementation.
  - Never override an inherited non-virtual method.

code-poly-5-diff.cc

```
class Shape{
  public:
    virtual void draw() const = 0;
    virtual void error(const std::string& msg);
    int objectID() const;
};

class Rectangle : public Shape {...};
class Triangle  : public Shape {...};
```

8.8 Polymorphic Destructors

Problem

- Destructors can be declared `virtual`.
- If a derived class object with a non-virtual destructor is destroyed explicitly by applying the delete operator to the base-class pointer, the C++ standard is *undefined*.
- The derived class destructors will be virtual *even though they do no have the same name as the base-class destructor*.

code-poly-6-destructors.cc

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

//-----------------------------------------------------------------------------
class TimeKeeper{
  public:
    TimeKeeper()              { cout << "TimeKeeper ctor" << endl;}
    ~TimeKeeper()             { cout << "TimeKeeper dtor" << endl;}
    virtual void time() const { cout << "Printing time"  << endl;}

    static TimeKeeper* getTimeKeeper(const string &s);     // Factor function...
};

class Watch : public TimeKeeper{
  public:
    Watch() { cout << "Watch ctor" << endl;}
    ~Watch(){ cout << "Watch dtor" << endl;}
};

//-----------------------------------------------------------------------------
TimeKeeper* TimeKeeper::getTimeKeeper(const string &s){
```

```
    TimeKeeper* ptr = 0;

    if( s == "Watch")
        ptr = new Watch();

    return ptr;
};

int main(){

    TimeKeeper* myTimePtr = TimeKeeper::getTimeKeeper("Watch");
    myTimePtr->time();

    delete myTimePtr;
}
```

9 C++ Casting

9.1 Static Cast

code-cast-0-static.cc

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

int main(){

        int total = 500;
        int days  = 9;

        cout << total/days << endl;
        cout << static_cast<double>(total)/days << endl;

        return 0;
}
```

code-cast-1-static.cc

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

class Account{
  public:
   void name() const { cout << "Account Name" << endl;}

};

class Savings : public Account{
  public:
    void type() const { cout << "Savings Account" << endl; }
};


void baseToDerived(Account * a){
    a->name();
    Savings* s = static_cast<Savings*>(a);
    s->type();

    static_cast<Savings*>(a)->type();
};

int main(){

    Savings a1;
    a1.name();
    a1.type();

    baseToDerived(&a1);

    return 0;
}
```

code-cast-2-static.cc

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

class Base{
  public:
    void see(){ cout << "see" << endl; }
};

class Derived : private Base{
  public:
    void look(){ cout << "look "; Base::see();}

};

int main(){

    Derived* d = new Derived();
    d->look();
    //d->see();  // No!

    // Problem...
    Base* b = (Base*)d;
    b->see();

    // Solution
    b = static_cast<Base*>(d);

        return 0;
}
```

9.2 Const cast

code-cast-3-const.cc

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

void print(double &d){
    d++;
    cout << " print = " << d << endl;
}

int main(){

    double myVal = 10.5;

    // Create a constant reference
    const double &myRef = myVal;
    cout << " myRef = " << myRef << endl;

    // 1. make a copy....
    double tmp = myRef;
    print(tmp);

    // 2. Cast away const-ness
    print(const_cast<double&>(myRef));

    cout << " myRef = " << myRef << endl;

    return 0;
}
```

9.3 Dynamic cast

code-cast-4-dynamic.cc

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

class Account{
  public:
   virtual ~Account(){};
   void name() const { cout << "Account Name" << endl;}

};

class Savings : public Account{
  public:
    void type() const { cout << "Savings Account" << endl; }
};


void baseToDerived(Account &a){
    a.name();
    Savings& s = dynamic_cast<Savings&>(a);
    s.type();
};

int main(){

    Savings a1;
    a1.name();
    a1.type();

    baseToDerived(a1);

    return 0;
}
```

9.4 Using dynamic_cast to Determine Type

code-cast-5-dynamic.cc

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

//-------------------------------------------------------------------------
class Package{
  public:
    virtual ~Package(){};
};

class Ground : public Package{};
class TwoDay : public Package{};

//-------------------------------------------------------------------------
int main(){

    Package* data [5];

    data[0] = new TwoDay();
    data[1] = new TwoDay();
```

```
    data[2] = new Ground();
    data[3] = new TwoDay();
    data[4] = new Ground();

    for(int i=0; i<5; ++i){
        Ground* gptr = dynamic_cast<Ground*>(data[i]);
        if( gptr != NULL ){
            cout << "Ground : " << i << endl;
        }
        delete data[i];
    }

    return 0;
}
```

9.5 Using typeid to Determine Type

code-type-6-name.cc

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

class Package{
  public:
    virtual ~Package(){};
};

class Ground : public Package{};
class TwoDay : public Package{};

//-----------------------------------------------------------------------------
int main(){

    Package *p1 = new TwoDay();
    Package *p2 = new Ground();

    cout << " p1 = " << typeid(p1).name() << endl;
    cout << " p2 = " << typeid(p2).name() << endl;
    cout << endl;
    cout << " *p1 = " << typeid(*p1).name() << endl;
    cout << " *p2 = " << typeid(*p2).name() << endl;

    return 0;
}
```

9.6 Reinterpret Cast

code-cast-6-reinterpret.cc

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

int main(){

    double d=1830;
    cout << "Address of d = " << &d << endl;

    double* b = reinterpret_cast <double*>(&d);
    cout << "b = " << *b << endl;

    int *p = reinterpret_cast <int*> (&d);
    cout << "p = " << *p << endl;

    for (int j=0; j<sizeof(double); ++j){
      cout<< p[j] << endl;
    }

    return 0;
}
```

10 The *string* Class

10.1 Word class

code-string-1-quiz.cc

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

//-----------------------------------------------------------------------------
class word : public string{
  public:
      word(){};
      word(const char *s):string(s){};
};

bool operator==(const word &a, const word &b){
    if( a.length() != b.length())
        return false;
    else{
        int l=0;
        while(l < a.length()){
            if( tolower(a[l]) != tolower(b[l]) )
                return false;
            l++;
        }
    }
    return true;
}

int main(){

    // Construct several words
    word w1("Steve");
    word w2 = "StEve";
    word w3;
    word w4(w1);
    w3 = "Stev";

    // Use the compare operator
    cout << "Word class" << endl;
    cout << w1 << (w1==w2? " == ": " != ") << w2 << endl;
    cout << w1 << (w1==w3? " == ": " != ") << w3 << endl;
    cout << w1 << (w1==w4? " == ": " != ") << w4 << endl;
    cout << endl;

    // The string class
    string s1(w1.c_str());
    string s2(w2.c_str());
    string s3(w3.c_str());
    string s4(w4.c_str());

    cout << "String class" << endl;
    cout << s1 << (s1==s2? " == ": " != ") << s2 << endl;
    cout << s1 << (s1==s3? " == ": " != ") << s3 << endl;
    cout << s1 << (s1==s4? " == ": " != ") << s4 << endl;
    cout << endl;

    return 0;
}
```

10.2 Assignment

code-string-2-assignment.cc

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

int main(){

    string s1;

    // Assignment
    s1 = "Hello";
    s1.assign("Hello");

    s1[0]    = 'h';
    s1.at(0) = 'h';

    // Concatenation
    s1 += " world";
    s1 = s1 + '!';
    s1.append("!");

    cout << s1 << endl;

    return 0;
}
```

10.3 Compare

code-string-3-compare.cc

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

int main(){

    string s1("A string to test.");
    string s2("The other string");

    if( s1 == s2 )
        cout << "s1 == s2" << endl;
    else
        cout << "s1 != s2" << endl;

    int result = s1.compare(s2);
    if( result == 0 )
        cout << "s1 == s2" << endl;
    else if( result < 0 )          // Negative when s1 is less than s2
        cout << "s1 < s2" << endl; // Meaning T is greater than A
    else
        cout << "s1 > s2" << endl;

    return 0;
}
```

10.4 Substring

code-string-4-substring.cc

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

int main(){

    string s1("I shall never see a poem lovely as a tree");

    cout << s1.substr(2,5)   << endl;
    cout << s1.substr(14,3) << endl;
    cout << s1.substr(20,4) << endl;
    cout << "as" << endl;
    cout << s1.substr(s1.length()-4,4) << endl;

    return 0;
}
```

10.5 Swap

code-string-5-swap.cc

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

int main(){

    string s1("peanut butter");
    string s2("chocolate");

    cout << "You put your " << s1 << " in my " << s2 << endl;

    cout << "...no" << endl;
    s1.swap(s2);
    cout << "You put your " << s1 << " in my " << s2 << endl;

    return 0;
}
```

10.6 Characteristics

code-string-6-characteristics.cc

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

void printStats(const string& s){
    cout << "-------------------------------" << endl;
    cout << "string     = " << s            << endl;
    cout << "capacity() = " << s.capacity() << endl; // Implementation dependent
    cout << "length()   = " << s.length()   << endl;
    cout << "empty()    = " << s.empty()    << endl;
    cout << "size()     = " << s.size()     << endl << endl;
}

int main(){

    string s1("This is my peanut butter cup");
    string s2("chocolate");
```

```
    string s3;

    printStats(s1);
    printStats(s2);
    printStats(s3);

    // Resize the string....
    s1.resize(static_cast<int>(1.2*s1.length()));
    printStats(s1);

    // Erase..
    s1.erase(0,11);
    printStats(s1);

    // Clear
    s1.clear();
    printStats(s1);

    // Reserve
    s3.reserve(10);
    printStats(s3);

    return 0;
}
```

10.7 Finding

code-string-7-find.cc

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

int main(){

    string s1 = "He was late——and dirty——which made his mother mad";

    cout << s1.find("dirty") << endl;
    cout << s1.find_first_of("dirty") << endl;

    string letters = "abcdefghijklmnopqrstuvwxyz";   // lower
    letters += "ABCDEFGHIJKLMNOPQRSTUVWXYZ";         // upper
    letters += " ";                                  // space

    cout << s1.find_first_not_of(letters) << endl;
    cout << s1.find_last_not_of(letters) << endl;

    return 0;
}
```

11 Function Templates

11.1 Motivation

Consider the following code segment: *code-templates-1-intro.cc*

```
#include <iostream>
//-------------------------------------------------------------------------------
int main(){

    std::cout << maximum(10,5)     << std::endl;
    std::cout << maximum(9.6,5.3) << std::endl;
    std::cout << maximum('a','A') << std::endl;

    return 0;
}
```

Implementation

- We can overload the `maximum()` function.
- We can make `maximum()` a template function.

Why Templates?

- The behavior of `maximum()` is identical for all types.
- Templates can express this more convenient and compact way.

Performance

- When the object code is created, is the object file larger using templates?

11.2 Template Functions

Here is the solution when we make `maximum()` a template function. *code-functions-24-template.cc*

```
template < class T >
T maximum(T value1, T value2){
    if(value1 < value2)
        return value2;
    else
        return value1;
}
```

Key Points

- You must use the *keyword* `typename` or the *keyword* `class` before the `T`.
- `T` is called the *type template parameter*.
- The compiler will make a type substitution.

11.3 Considerations

code-templates-2.cc

```
#include <iostream>

// This is the template function
//-------------------------------------------------------------------------------
template < class T >
T maximum(T value1, T value2){
    std::cout << "calling template" << std::endl;
    if(value1 < value2)
        return value2;
    else
        return value1;
}
```

```
// Same functionality, but only for ints
//-------------------------------------------------------------------------
int maximum(int a, int b){
    std::cout << "calling non-template" << std::endl;
    if( a < b )
        return b;
    else
        return a;
}

//-------------------------------------------------------------------------
int main(){

    std::cout << maximum(10,5)     << std::endl;
    std::cout << maximum(9.6,5.3) << std::endl;
    std::cout << maximum('a','A') << std::endl;

    return 0;
}
```

- Ordinary function is used if it matches a template function.

11.4 User defined types

Caution

- If you attempt to call `maximum()` with a user defined type and
- The type has not overloaded the `operator<()`, then
- Compilation error

12 Class Templates

12.1 Motivation

code-templates-3.cc

```
#include <iostream>

#include "intStack.h"
#include "strStack.h"
#include "doubleStack.h"

//-----------------------------------------------------------------------------
int main(){

    intStack s1;
    s1.push(10);

    strStack s2;
    s2.push("Templates");

    doubleStack s3;
    s3.push("10.5");

    return 0;
}
```

This is undesirable because...

- It involves writing a new class for each new type.
- Cumbersome to deal with all the *.h files.
- How about one class that does it all.

12.2 STL — a taster

code-templates-4.cc

```
#include <iostream>
#include <stack>

using namespace std;

//-----------------------------------------------------------------------------
int main(){

    // Integer stack...
    stack<int> s1;
    s1.push(10);
    s1.push(15);
    s1.push(7);
    s1.push(33);

    cout << s1.top() << endl;
    s1.pop();
    cout << s1.top() << endl;

    // Character stack
    stack<char> s2;
    s2.push('a');

    return 0;
}
```

Notice

- One class that changes based on the *type*.
- I am not dealing with memory management.

12.3 Creating a Class Template

code-templates-5-main.cc

```
#include <iostream>
#include "Stack.h"

//-----------------------------------------------------------------------------
int main(){

    Stack<int> s1;
    s1.push(10);
    s1.pop();

    return 0;
}
```

Stack.h

```
//-----------------------------------------------------------------------------
//      Stack.h
//      The entire class must be placed in the *.h file.
//-----------------------------------------------------------------------------
#ifndef STACK_H
#define STACK_H

template< typename T >
class Stack{
  public:
        Stack(){};
        ~Stack(){};

        bool push(const T&);
        bool pop();
};

template< typename T >
bool Stack<T>::push(const T& value){
    return true;
}

template< typename T >
bool Stack<T>::pop(){
    return true;
}

#endif
```

Details

- User-defined types must have.
  - A default constructor.
  - Support assignment operator.
- Member functions defined outside the class begin with: `template< typename T >`
- The class scope operator is `Stack<t>::` instead of `Stack::`.
- Everything in one file.

  The following is from http://www.cplusplus.com/doc/tutorial/templates.html. From the point of view of the compiler, templates are not normal functions or classes. They are compiled on demand, meaning that the code of a template function is not compiled until an instantiation with specific template arguments is required. At that moment, when an instantiation is required, the compiler generates a function specifically for those arguments from the template.

  Because templates are compiled when required, this forces a restriction for multi-file projects: the implementation (definition) of a template class or function must be in the same file as its declaration. That means that we cannot separate the interface in a separate header file, and that we must include both interface and implementation in any file that uses the templates.

12.4 Nontype parameters

code-templates-6-main.cc

```
#include <iostream>
#include "Stack-2.h"

//-----------------------------------------------------------------------------
int main(){

    Stack<int> s1;

    Stack<double,200> s2;

    Stack<> s3;

    return 0;
}
```

Stack-2.h

```
//-----------------------------------------------------------------------------
//      Stack.h
//      The entire class must be placed in the *.h file.
//-----------------------------------------------------------------------------
#ifndef STACK_H
#define STACK_H
#include <iostream>
using std::cout;
using std::endl;

template< typename T=int, int S=100>
class Stack{
  public:
        Stack(){ cout << S << endl;}
        ~Stack(){};

        bool push(const T&);
        bool pop();
};

template< typename T, int S >
bool Stack<T,S>::push(const T& value){
    return true;
```

```
}

template< typename T, int S >
bool Stack<T,S>::pop(){
    return true;
}

#endif
```

output

```
100
200
100
```

- You may specify defaults.
- Nontype parameters that have defaults are treated as `const`.

12.5 Static members

- Static means one copy.
- Each class-template specialization has its own copy of each static member.
- Must be defined and initialized at file scope.

code-templates-8-static.cc

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

template< typename T>
class Example{
  public:
    Example(){ num++;}
    int count(){ return num; }

  private:
    static int num;

};

// Initialization... defined at file scope
template< typename T> int Example<T>::num = 0;

int main(){

    Example<int> e1,e2,e3;
    Example<char> c1,c2;

    cout << e1.count() << endl;
    cout << c2.count() << endl;

    return 0;
}
```

12.6 Inheritance and Friends

- Inheritance — 4 points on page 762.
- Friends — some rules on page 763.

12.7 Template Constant Expressions

code-templates-7-constant.cc

```
#include <iostream>

template< int H, int W>
class Screen{
  public:
    Screen(){};
    int height(){ return H;}
    int width() { return W;}

  private:
    int myScreen [H][W];  // Not dynamically allocated
};

int main(){

    // The values must be known at compile time.
    Screen<1024,80> data;

    std::cout << " The screen is " << data.height() ;
    std::cout << " by " << data.width() << std::endl;

    int height, width;
    std::cin >> height;
    std::cin >> width;
    // This will not work because height and width are not known at
    // compile time.

    //Screen< height, width> data2;  //ERROR!

    return 0;
}
```

13 Motivation for the STL

13.1 Problem 1

Write a small program that reads words (stored as strings) into a contiguous array. You may only read the file once.

code-stl-2-array.cc

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

//-----------------------------------------------------------------------------
int main(){

    int capacity = 1;
    int size     = 0;
    string* data = new string[capacity];
    ifstream infile("/usr/share/dict/words");
    string temp;
    while(getline(infile,temp,'\n')){

        if( size < capacity){
            data[size] = temp;
            size++;
        }
        else{
            // Allocate new space
            string* d2 = new string[capacity*2];

            // Copy the data items
            for(int i=0; i<capacity; ++i){
                d2[i] = data[i];
            }
            // Delete old data
            delete [] data;

            // Assign data to new array and update capacity.
            data = d2;
            capacity*=2;

            data[size] = temp;
            size++;
        }
    }
    infile.close();

    cout << size << " " << capacity << endl;

    delete [] data ;

    return 0;
}
```

Remember...

- Adding one element each time was slow.
- When we double the `capacity` the peformance is similar to the `vector` class.

13.2 Vector class

See handout *code-stl-5-vector.cc* .

13.3 Using reserve()

code-stl-4-compare.cc

```
#include <iostream>
#include <vector>
#include "Timer.h"
using namespace std;

//-----------------------------------------------------------------------------
double fillVector(vector<int> &);
static const int MAX = 5000000;

//-----------------------------------------------------------------------------
int main(){

    vector<int> v1;
    vector<int> v2;
    v2.reserve(MAX);

    cout << fillVector(v1) << endl;
    cout << v1.size() << " " << v1.capacity() << endl;

    cout << fillVector(v2) << endl;
    cout << v2.size() << " " << v2.capacity() << endl;

    return 0;
}

//-----------------------------------------------------------------------------
double fillVector(vector<int> &v){
    Timer clock;
    clock.begin();
    for(int i=0; i<MAX; i++){
        v.push_back(i);
    }
    clock.end();
    return clock.time();
}
```

- `v2` is faster because it does not need to reallocate and copy during `fillVector`.

13.4 Problem 2

Write a template function named `find` that searches an array for a specific value. The function returns true if the value is found and false otherwise. Here is our solution.

code-stl-6-findarray.cc

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

template <typename T>
bool find(const T A[], const int size, const T &value ){
    for(int i=0; i<size; ++i){
        if( A[i] == value)
            return true;
    }
    return false;
}

int main(){

    // Array of integers
    const int SIZE = 5;
    int a1[SIZE] = {1,4,7,3,9};

    cout << find(a1,SIZE,3) << endl;
    cout << find(a1,SIZE,8) << endl;

    // Array of characters
    char name[] = "Alexander";

    cout << find(name,strlen(name),'b') << endl;
    cout << find(name,strlen(name),'A') << endl;

    return 0;
}
```

13.5 Problem 3

Write another template function, also named `find`, that is searches through and *array or a vector* for a specific value.

code-stl-7-find.cc

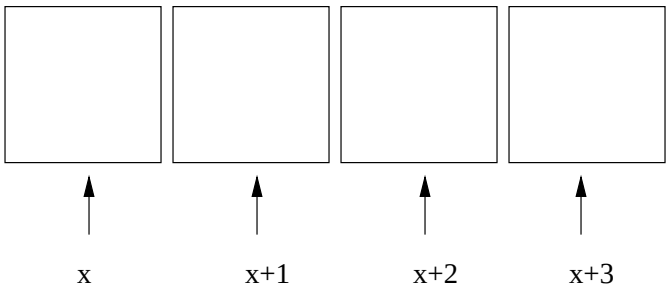
```
#include <iostream>
#include <string>
#include <vector>
#include <deque>

using namespace std;

template< typename T , typename N>
bool find(T* start, T* end, const N& value){
    T val = static_cast<T>(value);
        for(T* i=start; i < end; i++){
                if(*i == val)
                        return true;
        }
        return false;
}

int main(){

    // Array of integers
    const int SIZE = 5;
    int a1[SIZE] = {1,4,7,3,9};

    int* a_begin = a1;
    int* a_end   = a1+SIZE;

    cout << find(a_begin,a_end,3) << endl;
    cout << find(a_begin,a_end,8) << endl;

    // Vector of strings
    vector<string> names;
    names.push_back("John");
    names.push_back("Ellen");
    names.push_back("Tim");
    names.push_back("Mike");

    cout << find(&names[0],&names.back(),"Tim") << endl;
    cout << find(&names[0],&names.back(),"Jimmy") << endl;

    return 0;
}
```

13.6 Where are we going?

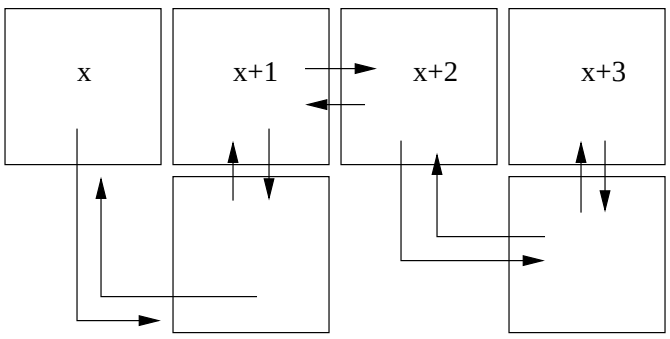
OVERLOADING	*TEMPLATES*	*GENERIC*
find(*int* data[]) find(*string* data[]) ⋮ find(*double* data[]) find(**vector**<*int*> &data) find(**vector**<*string*> &data) ⋮ find(**vector**<*double*> &data)	find(*T* data[]) find(**vector**<*T*> &data)	find(*?*)

13.7 linked lists

Vectors and Arrays



Linked−lists



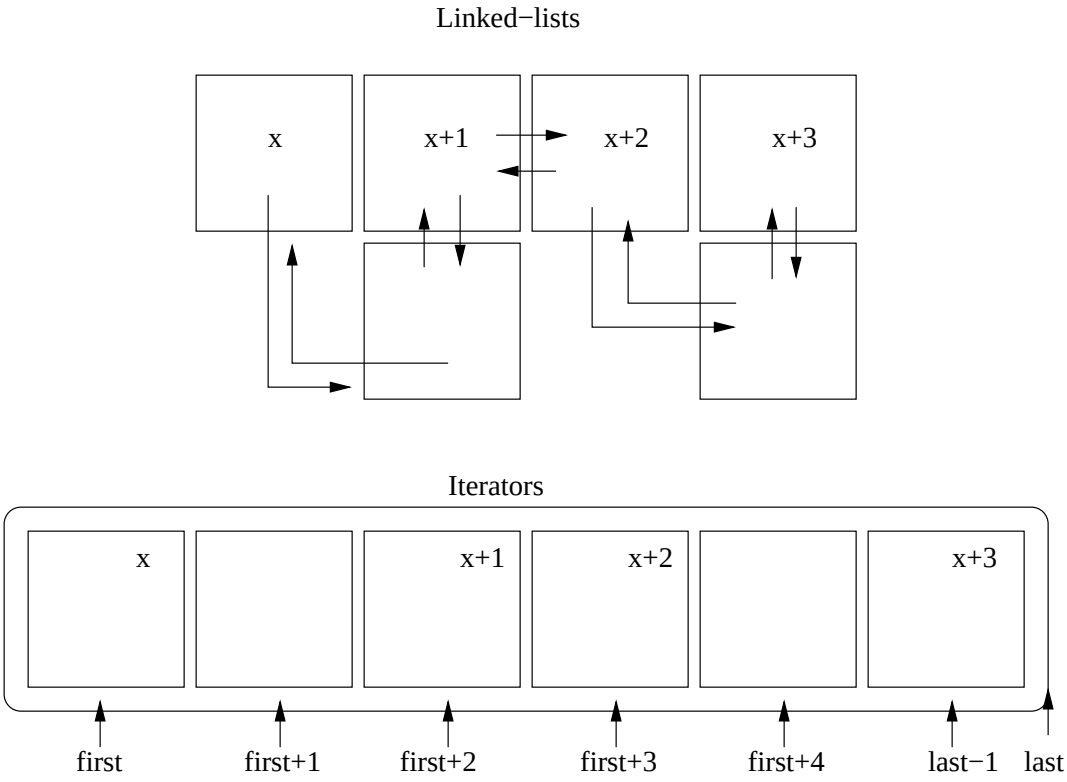
code-stl-8-list.cc

```
#include <iostream>
#include <string>
#include <vector>
#include <list>
using namespace std;

template< typename iterType>
void print(iterType start, iterType end){
        for(; start != end; ++start)
        cout << "      " << (int)&(*start) << " ---->   " << *start << endl;
}

int main(){

    string data[] = {"John","Ellen","Tim","Mike"};

    // List of strings
    list<string> nameList(data, data+2);

    // Vector of strings
    vector<string> names(data, data+4);

    // Add more to the list
    nameList.push_back(data[2]);
    nameList.push_back(data[3]);

    cout << "\nPrint( vector<string> )" << endl;
    print(names.begin(), names.end());

    cout << "\nPrint( list<string> )" << endl;
    print(nameList.begin(), nameList.end());

    return 0;
}
```

Output

```
Print( vector<string> )
      148164744 --->   John
      148164748 --->   Ellen
      148164752 --->   Tim
      148164756 --->   Mike

Print( list<string> )
      148164720 --->   John
      148164736 --->   Ellen
      148164776 --->   Tim
      148164792 --->   Mike
```

13.8 Iterators – a pointer abstraction



13.9 Iterator functionality

code-stl-9-find.cc

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

namespace learn{ // Avoid name clash with std::find()

    // Our find function....
    template< typename iterType , typename elemType>
    bool find(iterType first, iterType last, const elemType &value){
        while( first != last){
            if( value == *first)
                return true;
            ++first;
        }
        return false;
    }
}

//-------------------------------------------------------------------------------
int main(){

    const int SIZE = 5;

    int d1[SIZE] = {1,4,7,3,9}; // Array
    vector<int> d2(d1,d1+SIZE); // Vector
    list<int>   d3(d1,d1+SIZE); // List

    cout << learn::find(d1,          d1+SIZE,  3) << endl;
    cout << learn::find(d2.begin(), d2.end(), 3) << endl;
    cout << learn::find(d3.begin(), d3.end(), 3) << endl;

    return 0;
}
```

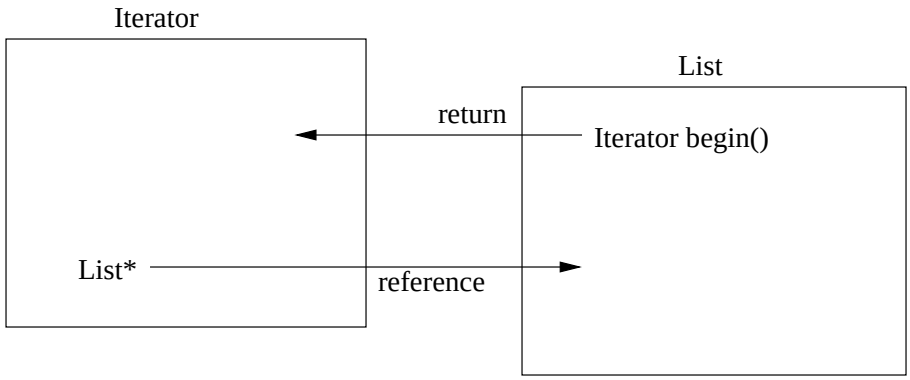
What functionality do we need?

- lines 30,31
  - `container.begin()`
  - `container.end()`
- line 11
  - Are two iterators equal? `operator==()`
- line 12
  - What is the value of the element? `operator*()`
- line 14
  - Where is the next element? `operator++()`

13.10 Building an iterator

code-stl-12-simpleList.cc

```
#include <iostream>
#include <iterator>
#include <algorithm>
#include "ListT.h"

using namespace std;

int main(){

    // declare and fill a list of integers
    ListT items;
    for(int i=0; i<10; i++)
            items.push_back(i*10);

    // define a list iterator
    ListT::listIterator i;

    // output each member of the list
    for( i = items.begin(); i != items.end(); ++i){
        cout << *i << endl;
    }

}
```



13.11 Problem 4 – Interconnected classes

code-stl-13-separate.cc

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

// forward declaration
class Iter;

// The list class
// ------------------------------------------------------------------------
class List{
  public:
    List():data(0){};
    Iter begin();
    int getData(){ return data;}
    void setData(int d){ data = d;}

  private:
    int data;
};

// Iter class
// ------------------------------------------------------------------------
class Iter{
  public:
    Iter(List* l=0): myList(l){};
    int getData(){ return myList->getData();}

  private:
    List* myList;
};

// ------------------------------------------------------------------------
Iter List::begin(){
    return Iter(this);
}

// ------------------------------------------------------------------------
int main(){

    List l;          // Create the list
    Iter i1(&l);     // Create the iterator
    cout << i1.getData() << endl;

    l.setData(10);
    cout << i1.getData() << endl;

    Iter i2 = l.begin();
    cout << i2.getData() << endl;

    return 0;
}
```

13.12 Problem 5 – Nested classes

code-stl-14-nested.cc

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

// forward declaration
class Iter;

// The list class
// ------------------------------------------------------------------------
class List{
  public:
    List():data(0){};

    int getData(){ return data;}
    void setData(int d){ data = d;}

    // Nested Iter class
    // ------------------------------------------------------------------------
    class Iter{
      public:
        Iter(List* l=0): myList(l){};
        int getData(){ return myList->getData();}

      private:
        List* myList;
    };

    Iter begin(){return Iter(this);}

  private:
    int data;
};

// ------------------------------------------------------------------------
int main(){

    List l;
    List::Iter i1(&l);
    cout << i1.getData() << endl;

    l.setData(10);
    cout << i1.getData() << endl;

    List::Iter i2 = l.begin();
    cout << i2.getData() << endl;

    return 0;
}
```

13.13 Problem 6 – List Iterator

ListTiter.h

```
#ifndef LISTT_H
#define LISTT_H

typedef int T;

// The node for the list
class node{
  public:
    node():next(0){};
    T data;
    node* next;
};

// list class
// ------------------------------------------------------------------------
class ListT{
  public:
    ListT();
    ~ListT();
    void push_back(const T&);

    // Nested iterator class
    // ------------------------------------------------------------------
    class listIterator{
      friend
        bool operator!=(const listIterator&, const listIterator&);

      public:
        listIterator(node* l=0);
        listIterator& operator++();
        T& operator*();
        void address(){ std::cout << link << std::endl;}

      typedef T   value_type;
      typedef T*  pointer;
      typedef T&  reference;

      private:
        node* link;

    };
    // ------------------------------------------------------------------

    listIterator begin(){ return ListT::listIterator(firstPtr);}
    listIterator end()  { return ListT::listIterator(lastPtr);}

  private:
    node* firstPtr;
    node* lastPtr;
    int size;
};

// ------------------------------------------------------------------------
//  Implementation details ::list::listIterator
// ------------------------------------------------------------------------
ListT::listIterator::listIterator(node* l): link(l){};

ListT::listIterator & ListT::listIterator::operator++(){
```

```
    link = link->next;
    return *this;
}

T& ListT::listIterator::operator*(){
    return link->data;
}

bool operator!=(const ListT::listIterator &A, const ListT::listIterator &B){
    return !( A.link == B.link);
}


// ------------------------------------------------------------------------
//  Implementation details ::list
// ------------------------------------------------------------------------

ListT::ListT(): size(0), firstPtr(0), lastPtr(0){}

ListT::~ListT(){
    if(size > 0){
        node* current = firstPtr;
        while( current != NULL ){
            node* tmp = current;
            std::cout << "      deleting " << tmp->data << std::endl;
            current = current->next;
            delete tmp;
        }
    }
}

void ListT::push_back(const T &value){
    // Create a node holding value
    node* tmp = new node;
    tmp->data = value;
    if( size == 0){
        firstPtr = lastPtr = tmp;
        ++size;
    }
    else{
        lastPtr->next = tmp;
        lastPtr = tmp;
        ++size;
    }
}

#endif
```

14 STL – Standard Template Library

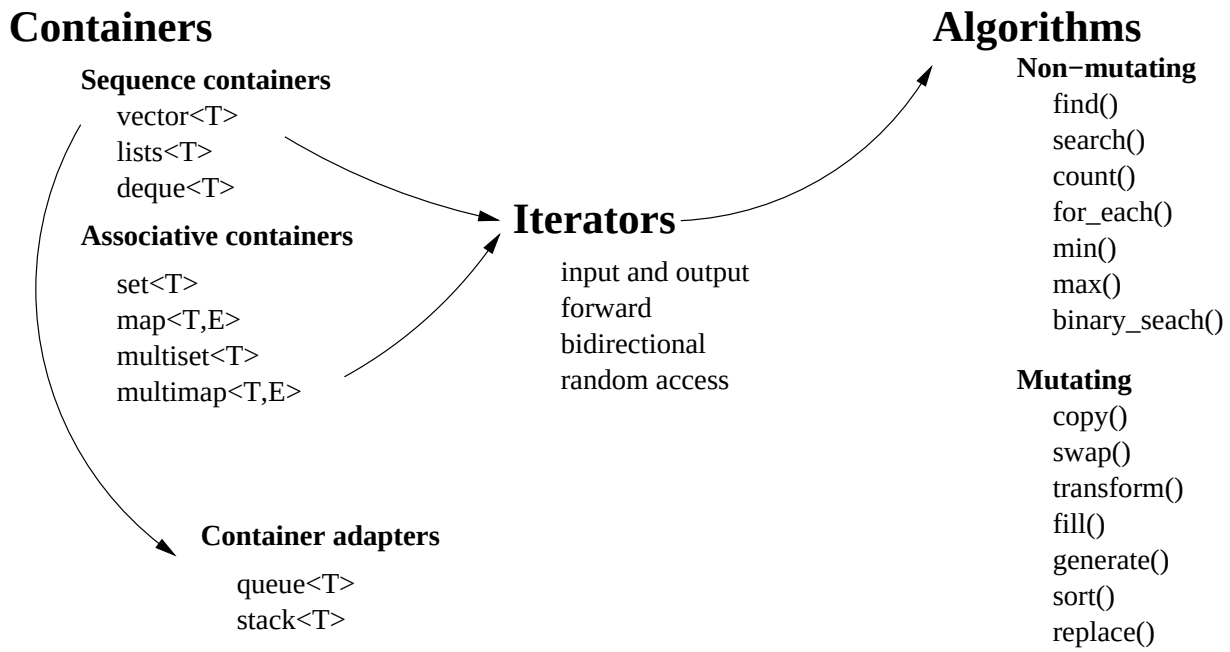
14.1 Overview

The STL use three components:

- Containers (e.g. vectors, lists, deques)
- Algorithms
- Iterators (e.g. we know these now)

Additionally, the STL uses

- Adapters – provide different interfaces
- Functors – objects as functions



14.2 Iterators

Input and output

- The ”container” is an input or output sequence.
- Forward direction – can be used with ”one-pass” algorithms.
- Read(write) from(to) a container

stl-code-1-io.cc

```
#include <iostream>
#include <iterator>  // anytime you use iterators

int main(){

    // Here is an example of an output stream.  You can also use an
    // input stream:
    //      istream_iterator<>
    // Pass cout, cin, or files


    std::ostream_iterator< int > output(std::cout,"\n");

    *output = 10;
    *output = 20;
    *output = 30;

    return 0;
}
```

Forward

- Only moves forward.
- More than “one pass”.
- Cannot step backwards.

Bidirectional

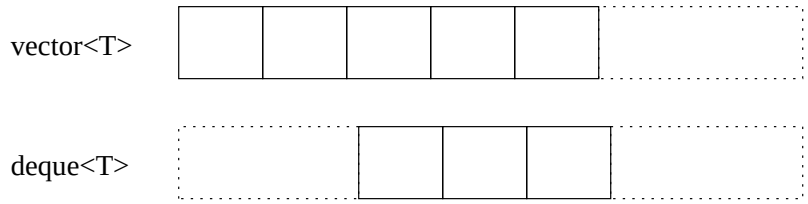
- Moves forward and backward (one step).
- More than “one pass”.

Random access

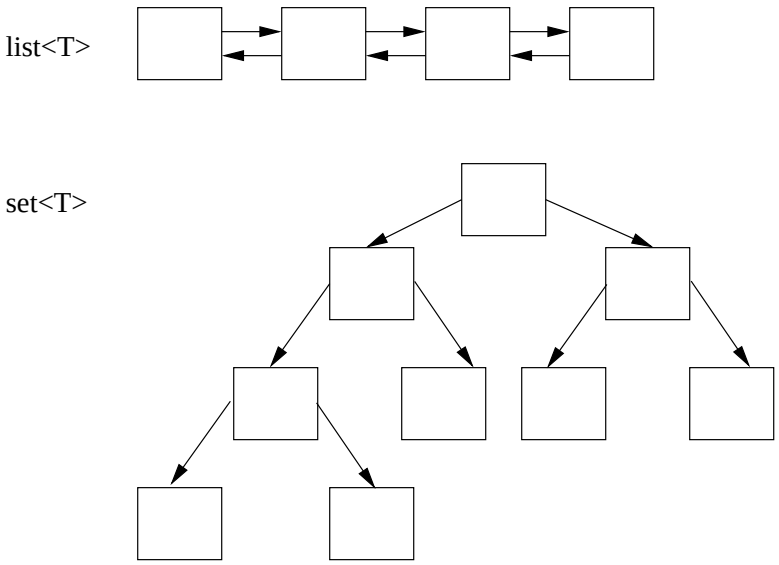
- Arbitrary steps.
- Difference between iterators.

14.3 Containers

random access iterator



bidirectional iterator



What happened to *input/output* and *forward* iterators?

- Input and output iterators are used with input and output stream iterators.
- Forward iterators, even if they aren't supplied by any container, allow algorithms to make it clear that they only require the iterators to go forward.

vector

- May allocate with *capacity*.
- Random access iterators.

Task	constant time	linear time
Random access to all elements	X	
Insertion and deletion		X
Insertion and deletion: end	X	
Insertion and deletion: start		X

deque

- No *capacity*.
- Random access iterators.

Task	constant time	linear time
Random access to all elements	X	
Insertion and deletion		X
Insertion and deletion: end	X	
Insertion and deletion: start	X	

list

- Bidirectional iterators.

Task	constant time	linear time
Random access to all elements		X
Insertion and deletion	X	
Insertion and deletion: end		X
Insertion and deletion: start	X	

set, multiset

- Stores sorted key values.
- Bidirectional iterators.
- Cannot use modifying algorithms on *k*eys.
- Logarithmic insertion, deletion, access to elements.

map, multimap

- Stores sorted key/value pairs.
- Bidirectional iterators.
- Cannot use modifying algorithms on *k*eys.
- Logarithmic insertion, deletion, access to elements.

14.4 Algorithms

- Operate on a range of elements defined by a pair of iterators.
- Mutating and non-mutating.
- Important because you cannot apply a *mutating* algorithm to an associative container.

14.5 Functions and Functors

- Generalize algorithms by adding function or functor parameters.
  - Execute same function on each element.
  - Count the number of elements in a container that satisfy a predicate (boolean condition).
- Binary function – Takes two arguments.
- Unary function – Takes one argument.

code-stl-1-functors.cc

```
#include <iostream>
#include <fstream>
#include <iterator>

using namespace std;

// Generic version of std::accumulate
//-----------------------------------------------------------------------------
template< typename T, typename Iterator, typename BinaryOp >
T accum(T init, Iterator start, Iterator end, BinaryOp fun){
    for(Iterator i=start; i != end; ++i)
        init = fun(init, *i);
    return init;
}

// Function
int sumInt(int a, int b){
    return a+b;
}

// Template
template< typename T>
T sumTemplate(const T& a, const T& b){
    return a+b;
}

// Object
template< typename T>
class sumObject{
  public:
    T operator()(const T&a, const T& b){
        return a+b;
    }
};


//-----------------------------------------------------------------------------
int main(){

     int data[10] = {1,2,3,4,5,6,7,8,9,10};

     // Use the function
     cout << accum(1, data, data+10, sumInt             ) << endl;

     // Use the template function
     cout << accum(1, data, data+10, sumTemplate<int>   ) << endl;

     // Use the function object -- functor
     cout << accum(1, data, data+10, sumObject<int>()   ) << endl;

    return 0;
}
```

Why Functors?

- Functors are more efficient.
- Functors can maintain state.
- Two functors with the same signature that overload `operator()` are still different.

STL support

- `plus<T>`
- `minus<T>`
- `divides<T>`
- `multiplies<T>`
- ...

Predicates

- A functor that returns a boolean value.
- The STL supports a full set of predicates.
  - `equal_to<T>`
  - `greater<T>`
  - `less<T>`
  - `not_equal<T>`
  - `less_equal<T>`
  - ...

code-stl-2-functors.cc

```
#include <iostream>
#include <fstream>
#include <iterator>

using namespace std;

//-----------------------------------------------------------------------------
int main(){

    plus<int> p;
    cout << p(2,3) << endl;

    equal_to<string> check;
    cout << check("Hello","Hello") << endl;

    return 0;
}
```

14.6 Adapters

Container adapters

- Queue
- Stack
- Priority queue.

Iterator adapters

- Standard iterators expect there is enough memory.
- Insert iterators are designed to *insert* instead of *overwrite*.
  - `front_inserter(container)`
  - `back_inserter(container)`
  - `inserter(container,pos)`

Functor adapters

- Instead of creating a new function, it is easier to adapt an existing one.
- Binders and negaters.

code-stl-3-functionadapters.cc

```
#include <iostream>
#include <fstream>
#include <iterator>

using namespace std;

//-----------------------------------------------------------------------------
int main(){

    // Binders....

    // x < 7
    cout << "\nx < 7" << endl;
    for(int i=6; i<9; i++)
        cout << i << " < 7 : ---> " << bind1st(less<int>(),7)(i) << endl;

    // 0.1*x
    double x = 100;
    cout << "\nf(x) = 0.1*x" << endl;
    cout << " x = " << x << " : ---> ";
    cout << bind2nd(multiplies<double>(),0.1)(x) << endl;

    return 0;
}
```

14.7 Examples

14.7.1 Searching, Finding, Counting

example-stl-1-find.cc

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

//-----------------------------------------------------------------------
//  Searching, Finding, Counting
//-----------------------------------------------------------------------
int main(){

    // Create a vector of elements.
    int array[10] = {3,2,5,6,8,9,1,7,4,11};
    vector<int> data(array, array+10);

    //  Does the number 8 exist in the array data?
    cout << ( find( data.begin(),
                    data.end(),
                    8) != data.end() ) << endl;

    // What is the first number greater than the number 8?
    cout << ( find_if( data.begin(),
                       data.end(),
                       bind2nd(greater<int>(), 8)) != data.end() ) << endl;

    // Does the sequence "8,9" exist?
    int y[2] = {8,9};
    cout << ( search(data.begin(),
                     data.end(),
                     y,
                     y+2) != data.end() ) << endl;

    // How many elements equal 8?
    cout << count(data.begin(),
                  data.end(),
                  8) << endl;

    // How many elements are greater than 5?
    cout << count_if(data.begin(),
                     data.end(),
                     bind2nd(greater<int>(),
                     5)) << endl;

    return 0;
}
```

14.7.2 Generate, Fill, Transform

example-stl-2-generate.cc

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

//-----------------------------------------------------------------------------
int numberGenerator(){
    static int i=1;
    return i*=2;
}

//-----------------------------------------------------------------------------
//  Generate, fill, Transform
//-----------------------------------------------------------------------------
int main(){

    // Create a vector of elements of 10 elements and initialize to zero.
    vector<int> data(10,0);
    vector<int> copyData(data);

    // Generate
    generate(data.begin(), data.end(), numberGenerator);

    // Fill with the number -1
    fill(data.begin(), data.end(), -1);

    // data = negate(data) e.g. make the elements positive
    transform(data.begin(),
              data.end(),
              data.begin(),
              negate<int>());

    // copyData = data*2
    transform(data.begin(),
              data.end(),
              copyData.begin(),
              bind2nd(multiplies<int>(),2));

    // data = data + copyData
    transform(data.begin(),
              data.end(),
              copyData.begin(),
              data.begin(),
              plus<int>());

    return 0;
}
```

14.7.3 For_each and Accumulate

example-stl-4-accumulate.cc

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>  // accumulate
#include <iterator>
using namespace std;

//-----------------------------------------------------------------------------
void print(int a){
    cout << a << " ";
}

//-----------------------------------------------------------------------------
// accumulate and for_each
//-----------------------------------------------------------------------------
int main(){

    // Create a vector of elements of 10 elements and initialize to 20.
    vector<int> data(10,20);

    // Print() each element
    for_each(data.begin(), data.end(), print);

    // sum the elements...
    cout << " sum = " << accumulate(data.begin(), data.end(), 0) << endl;

    return 0;
}
```

14.7.4 Sorting options

example-stl-3-sort.cc

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

//-----------------------------------------------------------------------------
// sorting
//-----------------------------------------------------------------------------
int main(){

    // Create a vector of elements of 10 elements and initialize to zero.
    vector<double> data(10);
    generate(data.begin(), data.end(), rand);

    // Sort the elements... based on operator<()
    sort(data.begin(), data.end());

    // Sort the other way
    sort(data.begin(), data.end(), greater<double>());

    // What is the are the 5 smallest values?
    partial_sort(data.begin(), data.begin()+5, data.end(), less<double>());

    return 0;
```

```
}
```

14.7.5 Copy

example-stl-5-copy.cc

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;


//-----------------------------------------------------------------------------
// Copy, replace...
//-----------------------------------------------------------------------------
int main(){

    // Create a vector of elements of 10 elements and initialize to 20.
    vector<int> data(10,20);

    // Copy vector to cout...
    ostream_iterator<int> output(cout,"\n");
    copy(data.begin(), data.end(), output);

    // Copy to another vector -- NOT THE BEST WAY TO DO THIS!
    vector<int> copyData(data.size());
    copy(data.begin(), data.end(), copyData.begin());

    // What if the vector is not the correct size?
    copyData.resize(0);
    copy(data.begin(), data.end(), back_inserter(copyData));

    return 0;
}
```

14.7.6 Binary Search and Equal Range

example-stl-6-binary.cc

```
#include <iostream>
#include <fstream>
#include <vector>
#include <algorithm>
#include <iterator>
using namespace std;

int randomInt(){
    return static_cast<int>(100.0*(rand()/(RAND_MAX+1.0)));
}

//-----------------------------------------------------------------------------
// Binary Search and Equal Range
//-----------------------------------------------------------------------------
int main(){

    // Create a vector of elements of 100 elements.
    vector<int> data(100);
    generate(data.begin(), data.end(), randomInt);

    // The must be sorted
    sort(data.begin(), data.end());
    copy(data.begin(), data.end(), ostream_iterator<int>(cout,"\n"));

    // Binary Search: Does the value 6 exist?
    cout << binary_search(data.begin(), data.end(),6) << endl;

    // Equal Range: Where does the value exist?
    typedef vector<int>::iterator vi;
    pair<vi,vi> results = equal_range(data.begin(), data.end(),6);

    if( results.first != results.second ){
        cout << " Range is not empty and 6 exists" << endl;

        cout << " The first value is " << *results.first << endl;

        cout << " There are " << distance(results.first,results.second)
             << " instances " << endl;
    }


    return 0;
}
```

14.7.7 Compare

example-stl-7-compare.cc

```
#include <iostream>
#include <vector>
#include <list>
#include <algorithm>
#include <iterator>

using namespace std;

bool ignoreCase(char a, char b){ return tolower(a) == tolower(b); }

//-------------------------------------------------------------------------------
int main(){

    string s1 = "Hello";
    string s2 = "heLLO-world";

    cout << "\n equal() uses operator==() \n";

    if( !equal(s2.begin(), s2.end(), s1.begin(), equal_to<char>()) )
        cout << "Strings are different" << endl;

    if( !equal(s1.begin(), s1.end(), s2.begin(), equal_to<char>()) )
        cout << "Strings are different" << endl;

    if( equal(s1.begin(), s1.end(), s2.begin(), ignoreCase) )
        cout << "Strings are the same!" << endl;

    if( !equal(s2.begin(), s2.end(), s1.begin(), ignoreCase) )
        cout << "Strings are different" << endl;


    cout << "\n lexicographical_compare() uses operator<()\n" ;

    if( lexicographical_compare(s1.begin(), s1.end(), s2.begin(), s2.end(), ignoreCase) )
        cout << "Strings are the same" << endl;

    if( lexicographical_compare(s1.begin(), s1.end(), s2.begin(), s2.end(), equal_to<char>()) )
        cout << "Strings are different" << endl;


    return 0;
}
```

14.7.8 Permutations

example-stl-8-perm.cc

```
#include <iostream>
#include <deque>
#include <algorithm>
#include <iterator>

using namespace std;

//-----------------------------------------------------------------------------
int main(){

    deque<bool> bitstring;
    for(int i=0; i<10; ++i)
        bitstring.push_back( (i%2 ? 1 : 0) );

    // Create a random bitstring...
    for(int t=0; t<10; t++){
        random_shuffle(bitstring.begin(), bitstring.end());
        copy(bitstring.begin(), bitstring.end(), ostream_iterator<bool>(cout));
        cout << endl;
    }

    return 0;
}
```

14.7.9 Other algorithms

- replace, replace_if
- replace_copy, replace_copy_if
- remove, remove_if

14.8 Practial Uses

14.8.1 How many lines are in a file?

practical-stl-1.cc

```
#include <iostream>
#include <fstream>
#include <iterator>
using namespace std;
//-----------------------------------------------------------------------------
int main(){

    ifstream inFile("/usr/share/dict/words");

    istreambuf_iterator<char> start(inFile);
    istreambuf_iterator<char> eof;

    // Count the number of lines in a file...
    cout << count(start, eof, '\n') << " Lines exist" << endl;

    inFile.close();

    return 0;
}
```

14.8.2 Store a file in a container?

practical-stl-2.cc

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <vector>
using namespace std;
//-----------------------------------------------------------------------------
int main(){

    // store words in a vector...
    ifstream inFile("/usr/share/dict/words");
    istream_iterator<string> eof;

    vector<string> words(istream_iterator<string>(inFile), eof);

    inFile.close();

    // How many words?
    cout << words.size() << endl;

    // What if you don't know the size?
    inFile.open("/usr/share/dict/words");
    vector<string> data;

    copy(istream_iterator<string>(inFile),
         istream_iterator<string>(),
         back_inserter(data));

    inFile.close();

    // How many words?
    cout << data.size() << endl;
```

```
    return 0;
}
```

14.8.3 What is the mean and standard deviation?

practical-stl-3.cc

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <vector>
#include <numeric>
using namespace std;

double randomInt(){
    return static_cast<double>(static_cast<int>(100.0*(rand()/
                                (RAND_MAX+1.0)))+500);
}

//-----------------------------------------------------------------------------
int main(){

    vector<double> values(20);
    generate(values.begin(), values.end(),randomInt);
    copy(values.begin(), values.end(), ostream_iterator<double>(cout," "));
    cout << endl;

    // Mean
    const double mean =  accumulate(values.begin(), values.end(),0.0)/
                          static_cast<double>(values.size());

    // Standard deviation
    vector<double> diff(values.size());
    transform(values.begin(), values.end(), diff.begin(), bind2nd(minus<double>(),mean));
    const double std = inner_product(diff.begin(), diff.end(), diff.begin(), 0.0)/mean;

    // Output
    cout << mean << "( " << std << " )" << endl;

    return 0;
}
```

14.9 Const and Reverse Iterator

vector.cc

```
// Fig. 23.14: Fig23_14.cpp
// Demonstrating Standard Library vector class template.
#include <iostream>
using std::cout;
using std::endl;

#include <vector> // vector class-template definition
using std::vector;

// function template for outputting vector elements
template < typename T > void printVector( const vector< T > &integers2 ){
   typename vector< T >::const_iterator constIterator; // const_iterator

   // display vector elements using const_iterator
   for ( constIterator = integers2.begin();
      constIterator != integers2.end(); ++constIterator )
      cout << *constIterator << ' ';
} // end function printVector

int main()
{
   const int SIZE = 6; // define array size
   int array[ SIZE ] = { 1, 2, 3, 4, 5, 6 }; // initialize array
   vector< int > integers(array,array+6); // create vector of ints

   // function push_back is in every sequence collection
   integers.push_back( 2 );
   integers.push_back( 3 );
   integers.push_back( 4 );
   cout << "\nOutput vector using iterator notation: ";
   printVector( integers );

   cout << "\nReversed contents of vector integers: ";
   // two const reverse iterators
   vector< int >::const_reverse_iterator reverseIterator;
   vector< int >::const_reverse_iterator tempIterator = integers.rend();

   // display vector in reverse order using reverse_iterator
   for ( reverseIterator = integers.rbegin();
      reverseIterator!= tempIterator; ++reverseIterator )
      cout << *reverseIterator << ' ';

   cout << endl;
   return 0;
} // end main
```

14.10 Container Examples

14.10.1 List

list.cc

```
// Fig. 23.17: Fig23_17.cpp
// Standard library list class template test program.
#include <iostream>
using std::cout;
using std::endl;

#include <list> // list class-template definition
#include <algorithm> // copy algorithm
#include <iterator> // ostream_iterator

// printList function template definition; uses
// ostream_iterator and copy algorithm to output list elements
template < typename T > void printList( const std::list< T > &listRef ){
      std::ostream_iterator< T > output( cout, " " );
      cout << "list = ";
      std::copy( listRef.begin(), listRef.end(), output );
      cout << endl;
} // end function printList

int main(){
   const int SIZE = 4;
   int array[ SIZE ] = { 2, 6, 4, 8 };
   std::list< int > values; // create list of ints
   std::list< int > otherValues; // create list of ints

   // insert items in values
   values.push_front( 1 );
   values.push_front( 2 );
   values.push_back( 4 );
   values.push_back( 3 );

   cout << "values contains: \n";
   printList( values );

   values.sort(); // sort values
   cout << "values.sort()\n";
   printList( values );

   // insert elements of array into otherValues
   otherValues.insert( otherValues.begin(), array, array + SIZE );
   cout << "After insert, otherValues contains: \n";
   printList( otherValues );

   // remove otherValues elements and insert at end of values
   values.splice( values.end(), otherValues );
   cout << "After splice()\n";
   printList( values );
   printList( otherValues );

   // insert elements of array into otherValues
   otherValues.insert( otherValues.begin(), array, array + SIZE );
   otherValues.sort();
   cout << "After insert, otherValues contains: \n";
   printList( otherValues );

   // remove otherValues elements and insert into values in sorted order
```

```
   values.merge( otherValues );
   cout << "After merge():\n";
   printList( values );
   printList( otherValues );

   values.pop_front(); // remove element from front
   values.pop_back(); // remove element from back
   cout << "After pop_front and pop_back:\n";
   printList( values );
   printList( otherValues );

   // remove duplicate elements
   cout << "After unique()\n";
   values.unique();
   otherValues.unique();
   printList( values );
   printList( otherValues );

   // swap elements of values and otherValues
   cout << "swap()\n" << endl;
   values.swap( otherValues );
   printList( otherValues );

   // replace contents of values with elements of otherValues
   values.assign( otherValues.begin(), otherValues.end() );
   cout << "After assign, values contains: \n";
   printList( values );

   // remove otherValues elements and insert into values in sorted order
   values.merge( otherValues );
   cout << "After merge, values contains: \n";
   printList( values );

   values.remove( 4 ); // remove all 4s
   cout << "After remove( 4 ), values contains: \n";
   printList( values );
   cout << endl;
   return 0;
} // end main
```

14.10.2 Deque

deque.cc

```
// Fig. 23.18: Fig23_18.cpp
// Standard Library class deque test program.
#include <iostream>
using std::cout;
using std::endl;

#include <deque> // deque class-template definition
#include <algorithm> // copy algorithm
#include <iterator> // ostream_iterator

int main()
{
   std::deque< double > values; // create deque of doubles
   std::ostream_iterator< double > output( cout, " " );

   // insert elements in values
   values.push_front( 2.2 );
   values.push_front( 3.5 );
   values.push_back( 1.1 );

   cout << "values contains: ";
   std::copy( values.begin(), values.end(), output );

   values.pop_front(); // remove first element
   cout << "\nAfter pop_front, values contains: ";
   std::copy( values.begin(), values.end(), output );

   // use subscript operator to modify element at location 1
   values[ 1 ] = 5.4;
   cout << "\nAfter values[ 1 ] = 5.4, values contains: ";
   std::copy( values.begin(), values.end(), output );
   cout << endl;
   return 0;
} // end main
```

14.10.3 Multiset

multiset.cc

```
// Fig. 23.19: Fig23_19.cpp
// Testing Standard Library class multiset
#include <iostream>
using std::cout;
using std::endl;

#include <set> // multiset class-template definition

// define short name for multiset type used in this program
typedef std::multiset< int, std::less< int > > Ims;

#include <algorithm> // copy algorithm
#include <iterator> // ostream_iterator

int main()
{
   const int SIZE = 10;
   int a[ SIZE ] = { 7, 22, 9, 1, 18, 30, 100, 22, 85, 13 };
   Ims intMultiset; // Ims is typedef for "integer multiset"
   std::ostream_iterator< int > output( cout, " " );

   cout << "There are currently " << intMultiset.count( 15 )
      << " values of 15 in the multiset\n";

   intMultiset.insert( 15 ); // insert 15 in intMultiset
   intMultiset.insert( 15 ); // insert 15 in intMultiset
   cout << "After inserts, there are " << intMultiset.count( 15 )
      << " values of 15 in the multiset\n\n";

   // iterator that cannot be used to change element values
   Ims::const_iterator result;

   // find 15 in intMultiset; find returns iterator
   result = intMultiset.find( 15 );

   if ( result != intMultiset.end() ) // if iterator not at end
      cout << "Found value 15\n"; // found search value 15

   // find 20 in intMultiset; find returns iterator
   result = intMultiset.find( 20 );

   if ( result == intMultiset.end() ) // will be true hence
      cout << "Did not find value 20\n"; // did not find 20

   // insert elements of array a into intMultiset
   intMultiset.insert( a, a + SIZE );
   cout << "\nAfter insert, intMultiset contains:\n";
   std::copy( intMultiset.begin(), intMultiset.end(), output );

   // determine lower and upper bound of 22 in intMultiset
   cout << "\n\nLower bound of 22: " << *( intMultiset.lower_bound( 22 ) );
   cout << "\nUpper bound of 22: " << *( intMultiset.upper_bound( 22 ) );

   // p represents pair of const_iterators
   std::pair< Ims::const_iterator, Ims::const_iterator > p;

   // use equal_range to determine lower and upper bound
   // of 22 in intMultiset
```

```
   p = intMultiset.equal_range( 22 );

   cout << "\n\nequal_range of 22:" << "\n   Lower bound: "
      << *( p.first ) << "\n   Upper bound: " << *( p.second );
   cout << endl;
   return 0;
} // end main
```

14.10.4 Set

set.cc

```
// Fig. 23.20: Fig23_20.cpp
// Standard Library class set test program.
#include <iostream>
using std::cout;
using std::endl;

#include <set>

// define short name for set type used in this program
typedef std::set< double, std::less< double > > DoubleSet;

#include <algorithm>
#include <iterator> // ostream_iterator

int main()
{
   const int SIZE = 5;
   double a[ SIZE ] = { 2.1, 4.2, 9.5, 2.1, 3.7 };
   DoubleSet doubleSet( a, a + SIZE );;

   std::ostream_iterator< double > output( cout, " " );
   cout << "doubleSet contains: ";
   std::copy( doubleSet.begin(), doubleSet.end(), output );

   // p represents pair containing const_iterator and bool
   std::pair< DoubleSet::const_iterator, bool > p;

   // insert 13.8 in doubleSet; insert returns pair in which
   // p.first represents location of 13.8 in doubleSet and
   // p.second represents whether 13.8 was inserted
   p = doubleSet.insert( 13.8 ); // value not in set
   cout << "\n\n" << *( p.first )
      << ( p.second ? " was" : " was not" ) << " inserted";
   cout << "\ndoubleSet contains: ";
   std::copy( doubleSet.begin(), doubleSet.end(), output );

   // insert 9.5 in doubleSet
   p = doubleSet.insert( 9.5 ); // value already in set
   cout << "\n\n" << *( p.first )
      << ( p.second ? " was" : " was not" ) << " inserted";
   cout << "\ndoubleSet contains: ";
   std::copy( doubleSet.begin(), doubleSet.end(), output );
   cout << endl;
   return 0;
} // end main
```

14.10.5 Multimap

multimap.cc

```
// Fig. 23.21: Fig23_21.cpp
// Standard Library class multimap test program.
#include <iostream>
using std::cout;
using std::endl;

#include <map> // map class-template definition

// define short name for multimap type used in this program
typedef std::multimap< int, double, std::less< int > > Mmid;

int main()
{
   Mmid pairs; // declare the multimap pairs

   cout << "There are currently " << pairs.count( 15 )
      << " pairs with key 15 in the multimap\n";

   // insert two value_type objects in pairs
   pairs.insert( Mmid::value_type( 15, 2.7 ) );
   pairs.insert( Mmid::value_type( 15, 99.3 ) );

   cout << "After inserts, there are " << pairs.count( 15 )
      << " pairs with key 15\n\n";

   // insert five value_type objects in pairs
   pairs.insert( Mmid::value_type( 30, 111.11 ) );
   pairs.insert( Mmid::value_type( 10, 22.22 ) );
   pairs.insert( Mmid::value_type( 25, 33.333 ) );
   pairs.insert( Mmid::value_type( 20, 9.345 ) );
   pairs.insert( Mmid::value_type( 5, 77.54 ) );

   cout << "Multimap pairs contains:\nKey\tValue\n";

   // use const_iterator to walk through elements of pairs
   for ( Mmid::const_iterator iter = pairs.begin();
      iter != pairs.end(); ++iter )
      cout << iter->first << '\t' << iter->second << '\n';

   cout << endl;
   return 0;
} // end main
```

14.10.6 Map

map.cc

```
// Fig. 23.22: Fig23_22.cpp
// Standard Library class map test program.
#include <iostream>
using std::cout;
using std::endl;

#include <map> // map class-template definition

// define short name for map type used in this program
typedef std::map< int, double, std::less< int > > Mid;

int main()
{
   Mid pairs;

   // insert eight value_type objects in pairs
   pairs[15] = 2.7;
   pairs[30] = 111.11;
   pairs.insert( Mid::value_type( 5, 1010.1 ) );
   pairs.insert( Mid::value_type( 10, 22.22 ) );
   pairs.insert( Mid::value_type( 25, 33.333 ) );
   pairs.insert( Mid::value_type( 5, 77.54 ) ); // dup ignored
   pairs.insert( Mid::value_type( 20, 9.345 ) );
   pairs.insert( Mid::value_type( 15, 99.3 ) ); // dup ignored
   pairs[100] = 20.5;


   cout << "pairs contains:\nKey\tValue\n";

   // use const_iterator to walk through elements of pairs
   for ( Mid::const_iterator iter = pairs.begin();
      iter != pairs.end(); ++iter )
      cout << iter->first << '\t' << iter->second << '\n';

   pairs[ 25 ] = 9999.99; // use subscripting to change value for key 25
   pairs[ 40 ] = 8765.43; // use subscripting to insert value for key 40

   cout << "\nAfter subscript operations, pairs contains:\nKey\tValue\n";

   // use const_iterator to walk through elements of pairs
   for ( Mid::const_iterator iter2 = pairs.begin();
      iter2 != pairs.end(); ++iter2 )
      cout << iter2->first << '\t' << iter2->second << '\n';

   cout << endl;
   return 0;
} // end main
```

14.11 Case Study: Graph class

14.11.1 Example

graphExample.cc

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

#include <fstream>
#include <string>
#include <iterator>
#include <map>

// Graph class
//-----------------------------------------------------------------------------
template< typename V, typename E>
class graph{
  public:

    graph(){};
    ~graph(){};

    graph& addEdge(const V &, const V &, const E&);
    graph& dotOutput(const char *);

  private:

};

// Format for dot output
//-----------------------------------------------------------------------------
template<typename V, typename E>
graph<V,E>& graph<V,E>::dotOutput(const char * filename){

    // Create the file object
    ofstream outFile(filename);
    outFile << "digraph graph1 {" << endl;

    // Finish and clean
    outFile << "}" << endl;
    outFile.close();

    return *this;
}

// Add undirected edge from a to b with weight w
//-----------------------------------------------------------------------------
template<typename V, typename E>
graph<V,E>& graph<V,E>::addEdge(const V &a, const V &b, const E &w){
    // Add the two vertices

    return *this;
}

int main(){

    // Traveling salesman...
    // ---------------------------------------------------------------------
    graph<string,double> cities;
```

```
    cities.addEdge("A","B",10.0);
    cities.addEdge("A","C",10.0);
    cities.addEdge("A","D",15.0);

    cities.addEdge("B","C",5.0);
    cities.addEdge("B","F",10.0);
    cities.addEdge("B","G",5.0);

    cities.addEdge("C","D",10.0);
    cities.addEdge("C","E",20.0);
    cities.addEdge("C","F",15.0);

    cities.addEdge("D","E",10.0);
    cities.addEdge("F","E",10.0);
    cities.addEdge("F","G",5.0);

    cities.dotOutput("test.dot");

    return 0;
}
```

14.11.2 Solution

graphSolution.cc

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

#include <fstream>
#include <string>
#include <iterator>
#include <map>

// Graph class
//-----------------------------------------------------------------------------
template< typename V, typename E>
class graph{
  public:

    graph(){};
    ~graph(){};

    graph& addEdge(const V &, const V &, const E&);
    graph& dotOutput(const char *);

  private:
    typedef map<V, E>      inner;
    typedef map<V, inner> outer;
    outer myGraph;

};

// Format for dot output
//-----------------------------------------------------------------------------
template<typename V, typename E>
graph<V,E>& graph<V,E>::dotOutput(const char * filename){

    // Create the file object
    ofstream outFile(filename);
    outFile << "digraph graph1 {" << endl;
    // Walk though the map
    typename outer::iterator i;
    typename inner::iterator j;
    for(i = myGraph.begin(); i != myGraph.end(); ++i){
        V startNode = i->first;
        inner tmp = i->second;
        for(j = tmp.begin(); j != tmp.end(); ++j){
            V endNode = j->first;
            if( startNode != endNode){
                outFile << "\"" << startNode << "\" -> \"";
                outFile << endNode << "\"" << endl;
            }
        }
    }

    // Finish and clean
    outFile << "}" << endl;
    outFile.close();

    return *this;
}
```

```
// Add undirected edge from a to b with weight w
//-----------------------------------------------------------------------------
template<typename V, typename E>
graph<V,E>& graph<V,E>::addEdge(const V &a, const V &b, const E &w){
    // Add the two vertices
    myGraph[a][b] = w;
    return *this;
}

int main(){

    // Traveling salesman...
    // ----------------------------------------------------------------------------
    graph<string,double> cities;

    cities.addEdge("A","B",10.0);
    cities.addEdge("A","C",10.0);
    cities.addEdge("A","D",15.0);

    cities.addEdge("B","C",5.0);
    cities.addEdge("B","F",10.0);
    cities.addEdge("B","G",5.0);

    cities.addEdge("C","D",10.0);
    cities.addEdge("C","E",20.0);
    cities.addEdge("C","F",15.0);

    cities.addEdge("D","E",10.0);
    cities.addEdge("F","E",10.0);
    cities.addEdge("F","G",5.0);

    cities.dotOutput("test.dot");

    return 0;
}
```

15 Midterm II Review Topics

Take-home Quiz: Write **one** test question and bring it to class on Thursday.

15.1 Object-Based Programming

- Constructors: assignment and initialization lists, default parameters.
- Chaining methods and the *this* pointer.
- Operator overloading.
- Non-member functions, friends, and they *static* keyword.
- Returning a *local object*!
- Dynamic data members: what should you worry about?
- Nested classes.

15.2 Using Inheritance

- Make sure public inheritance models an "is-a" relationship.
- Base and derived constructors and destructors (e.g. default constructors).
- Hiding inherited names.
- Problems when overriding inherited *non-virtual* functions.
- Make sure you copy everything.

15.3 Polymorphism

- What is polymorphism?
- Differentiate between *non-virtual*, *virtual*, *pure-virtual*.
- Polymorphic base class destructors must be *virtual*.
- "Slicing problem" (we haven't discussed this yet).

15.4 Template Programming

- Understand compile-time polymorphism (e.g. code generation).
- Type and Non-type parameters and defaults.
- Constant expressions.

15.5 Generic Programming

- Dynamic array (similar to string example).
- Overloading, templates, generic.
- Iterator concepts

15.6 Example questions

1. **GENERIC** The generic algorithm `find` searches a sequence of elements for a specific value. The first two arguments are iterators that define the sequence of elements to be searched. The third argument is the value of the element you are searching for.

   (a) Write the `find` generic function. Your function should return `true` if the element exists and `false` if it does not.

   (b) What type of iterator does your method require (circle one)?

   - Input/output
   - Forward
   - Bidirectional
   - Random access

   (c) What operations must the iterator support (circle one or more)?

   - operator==()
   - operator!=()
   - operator¡()
   - operator¿()
   - operator*()
   - operator++()
   - operator++(int)

   (d) Why will this algorithm work for any container that supports iterators?

2. **TEMPLATE** The `max` function returns the greater of two types. For example, an integer `max` function may look like:

```
int max(const int x, const int y){
    if( x < y )
        return y;
    return x;
}
```

   (a) Write a template function that generalizes max for any type.

   (b) What property must a type support in order to use the template function?

3. **Polymorphism**

   (a)

   (b) Differentiate between *non-virtual*, *virtual*, and *pure-virtual* in term of how they

16 Inheritance and Polymorphism Review

16.1 Virtual and non-virtual methods

Interface and implementation details for public inheritance

- **Non-virtual**: A function that is *invariant over specialization* (e.g. the behavior is not supposed to change). The derived class inherits the interface and implementation and it is the same for all derived instances (e.g. Not polymorphic).
- **Virtual**: A derived class inherits the interface and a *default* implementation. That is, the behavior may change in the derived class.
- **Pure-virtual**: A derived class inherits the function interface *only*. This means that a derived class must define the function. The base class is called an *abstract* class.

Important: under public inheritance, member function interfaces are always inherited.

16.2 Public inheritance: non-virtual

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

//-------------------------------------------------------------------------
class person{
  public:
    void eat(){cout << " eat()" << endl;}
};

//-------------------------------------------------------------------------
class student : public person{
  public:
    void study(){ cout << " study()" << endl;}
};

//-------------------------------------------------------------------------
int main(){

    student s1;
    person  p1;

    person* p2 = &s1;
    person& p3 = s1;
    person  p4 = s1;

    s1.eat();
    s1.study();
    p1.eat();

    p2->eat();
    p3.eat();
    p4.eat();

    return 0;
}
```

Output

```
eat()
study()
eat()
eat()
eat()
eat()
```

Important concepts

- Make sure *public inheritance* models an *is-a* relationship.
  - A student *is a* person.
  - A Fox *is-a* Element of the forest.
- If you want a *has-a* relationship, *composition* is more appropriate.
  - A car *has-a* tire.

16.3 Dangerous temptations

- It is tempting (and reasonable) to want to change the behavior of how a person eats based on the derived class. e.g. a student eats cheap food and a doctor lives it up.
- **But, this is polymorphic behavior** — not public non-virtual inheritance.

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

//-------------------------------------------------------------------------
class person{
  public:
    void eat(){ cout << "person eat()" << endl;}
};

//-------------------------------------------------------------------------
class student : public person{
  public:
    void eat(){ cout << "student eat()" << endl;}
};

//-------------------------------------------------------------------------
int main(){

    student s1;
    person  p1;

    person* p2 = &s1;
    person& p3 = s1;
    person  p4 = s1;

    s1.eat();
    p1.eat();

    p2->eat();
    p3.eat();
    p4.eat();

    return 0;
}
```

Output

```
student eat()
person eat()
person eat()
person eat()
person eat()
```

- If the `eat()` method is not *invariant over specialization*, then the method needs to be virtual.

16.4 Public inheritance: simple-virtual

Solution

- Make `person::eat()` virtual.
- Student inherits the interface and a default implementation.

Output

```
student eat()
student eat()
student eat()
person eat()
```

Very important concept

- In the non-virtual `person::eat()` method, the behavior of the `eat()` method was determined by the *declaration type*. This is called **static binding** (or early binding).
- When methods are virtual, the *type of object being pointed to* determines the behavior of the object. This is called **dynamic binding** (or late binding).

Slicing problem

- Notice that `person p4` called the `person::eat()` method.
- The polymorphic behavior of `person p4` conflicts with stack-based memory allocation because the derived object (`student`) is larger than the base object.
- To maintain efficiency, the additional overhead of the derived class is *sliced* away.

16.5 Hiding base-class names

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

//-----------------------------------------------------------------------------
class Base{
  public:
    virtual void m1()      { cout << "Base m1()" << endl;}
    virtual void m1(int x) { cout << "Base m1(x)" << endl;}

    void f1()         { cout << "Base f1()" << endl;}
    void f1(double x) { cout << "Base f1(x)" << endl;}
};

//-----------------------------------------------------------------------------
class Derived: public Base{
  public:
    virtual void m1(){ cout << "Derived m1()" << endl;}
    void f1()        { cout << "Derived f1()" << endl;}
};

int main(){

    Derived d;
    d.m1();
    //d.m1(8); ERROR!

    d.f1();
    //d.f1(8.0); ERROR!

    return 0;
}
```

Output

```
Derived m1()
Derived f1()
```

Notice

- Overriding a base class method hides other methods that *should* be inherited.

Solution

- **Never override a non-virtual method**.
- You will almost always want to override the C++ hiding mechanism for virtual functions. You can do this with the `using Base::m1;` command in the public section of your derived classes.

16.6 Destructors

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

//-----------------------------------------------------------------------------
class base{
  public:
    base() { cout << "base-ctor" << endl;}
    ~base(){ cout << "base-dtor" << endl;}

    virtual void print(){ cout << "print() ---> base" << endl;}
};
```

```
//-------------------------------------------------------------------------------
class derived : public base{
  public:
    derived(){  cout << "derived-ctor" << endl;}
    ~derived(){ cout << "derived-dtor" << endl;}
    void print(){ cout << "print() ---> derived" << endl;}
};

//-------------------------------------------------------------------------------
int main(){

    base* b1 = new derived();
    b1->print();
    delete b1;

    return 0;
}
```

Output

```
base-ctor
derived-ctor
print() ---> derived
base-dtor
```

- If a derived class object with a non-virtual destructor is destroyed explicitly by applying the delete operator to the base-class pointer, the C++ standard is *undefined*.
- The derived class destructors will be virtual *even though they do no have the same name as the base-class destructor*.

16.7 Conclusion

- Make sure public inheritance models an *is-a* relationship.
- Understand when you want *inheritance* and when you need *polymorphic* behavior.
- Non-virtual methods should never be redefined in a derived class.
- Avoid *hiding* inherited names. Overriding virtual base class methods prohibits the inheritance of other overloaded methods with the same name (e.g. it hides them). Resolve this with the `using` declaration.
- Always declare destructors virtual in polymorphic base classes.

17 Exceptions

17.1 Motivation

OptCG.C

```
// Copyright (C) 1993:
// Sandia National Laboratories

//-------------------------------------------------------------------------
int OptCG::computeStep(ColumnVector sk){

  int step_type = linesearch(nlp, optout, sk, sx, &stp_length, stpmax, stpmin,
                                        itnmax, ftol, xtol, gtol);
  if (step_type < 0) {
    setMesg("OptCG: Step does not satisfy sufficient decrease condition");
    ret_code = -1;
    setReturnCode(ret_code);
    return(-1);
  }
  return(step_type);
}

// Later, in another function...
//-------------------------------------------------------------------------
void OptCG::calling(){

        if ((step_type = computeStep(search)) < 0) {
                setMesg("nlcg: Step does not satisfy sufficient decrease condition");
                ret_code = step_type;
        setReturnCode(ret_code);
                return;
    }
}
```

linesearch.C

```
//-------------------------------------------------------------------------
int linesearch(NLP1* nlp, ostream *optout,
               ColumnVector& search_dir, ColumnVector& sx,
               double *step_length, double stpmax, double stpmin,
               int itnmax, double ftol, double xtol, double gtol){

  int step_type;
  int expensive_function = nlp->getIsExpensive();

  if (expensive_function)

        step_type = backtrack(nlp, optout, search_dir, sx, step_length,
                                    itnmax, ftol, stpmax, stpmin);
  else {

    step_type = mcsrch(nlp, search_dir, optout, step_length, itnmax,
                           ftol, xtol, gtol, stpmax, stpmin);
  }
  return(step_type);
}
```

mcsrch.C

```
int mcsrch(NLP1* nlp, ColumnVector& s, ostream *fout, double *stp,
           int itnmax, double ftol, double xtol, double gtol, double stpmax,
           double stpmin){
/**********************************************************************
 *   subroutine mcsrch
 *     info is an integer output variable set as follows:
 *       info =-1 improper input parameters.
 *       info = 1  the sufficient decrease condition and the
 *                 directional derivative condition hold.
 *       info =-2  relative width of the interval of uncertainty
 *                 is at most xtol.
 *       info =-4  the step is at the lower bound stpmin.
 *       info =-5  the step is at the upper bound stpmax.
 *       info =-6  rounding errors prevent further progress.
 *
 **********************************************************************/

  infoc = 1;

  /*    check the input parameters for errors. */
  if (n <= 0 || *stp <= zero || ftol < zero || gtol < zero ||
      xtol < zero || stpmin < zero || stpmax < stpmin ) {
    infoc = -1;
    return infoc;
  }

  if (dginit >= zero) {
    return -1;
  }

    if (brackt && (*stp <= stmin || *stp >= stmax) || infoc == 0) {
      info = -6;  // CPJW 12/10/2003 original stmt info = 6
    }
    if (*stp == stpmax && fvalue <= ftest1 && dg <= dgtest) {
      info = -5;  // CPJW 12/10/2003 original stmt info = 5
    }
    if (*stp == stpmin && (fvalue > ftest1 || dg >= dgtest)) {
      info = -4;  // CPJW 12/10/2003 original stmt info = 4
    }
    if (brackt && stmax - stmin <= xtol * stmax) {
      info = -2;  // CPJW 12/10/2003 original stmt info = 2
    }
    if (fvalue <= ftest1 && fabs(dg) <= gtol * (-dginit)) {
      info = 1;
    }

    /*     check for termination. */
    if (info < 0) {
        return(-1);
    }
    if (info != 0) {
      if (siter == 1) return(Newton_Step);
      else return(Backtrack_Step);
    }
  return (-1);  // too many iterations;

}  /* last line of subroutine mcsrch. */
```

17.2 Simple Translation

code-exception-1.cc

```
1  #include <iostream>
2  #include <fstream>
3  #include <new>
4  using namespace std;
5
6  //-----------------------------------------------------------------------------
7  int work( const char * filename, const int size){
8
9      // Task 1: Open a file
10     ifstream file(filename);
11     if( file == 0 )
12         return -1;  // Could not open file
13
14     // Task 2: allocate some memory
15     double* array[50];
16     for(int i=0; i<50; ++i){
17         array[i] = new (nothrow) double[size];
18         if( array[i] == 0 )
19             return -2; // Memory problems
20     }
21     return 1;
22 }
23
24 //-----------------------------------------------------------------------------
25 int main(){
26
27     int error = 0;
28
29     // Try working...
30     error = work("data",100);
31     if( error == -1 )
32         cout << "Could not open file" << endl;
33     else if ( error == -2 )
34         cout << "Memory problems" << endl;
35     else
36         cout << "Complete" << endl;
37
38     // Try working again...
39     error = work("code-exception-1.cc",1000000000);
40     if( error == -1 )
41         cout << "Could not open file" << endl;
42     else if ( error == -2 )
43         cout << "Memory problems" << endl;
44     else
45         cout << "Complete" << endl;
46
47     return 0;
48 }
```

Problems

- Check after every call — work!
- May forget or ignore.

17.3 Using Exceptions

The `exception` class.

- Use `#include <exception>`.
- Contains a `virtual` method called `what()`.
- The Standard Library derives classes from this base class.
  - `runtime_error`
    - e.g. arithmetic `overflow_error`.
    - user defined.
  - `logic_error`
    - e.g. `out_of_range`.
  - `bad_alloc`
- **Note:** Exceptions to not have to inherit from the base `exception` class.

When should we use them?

- To process uncommon or low-frequency events.
- Examples:
  - out-of-bounds
  - arithmetic overflow
  - division by zero
  - memory allocation.
  - file not found.
- When no exceptions occur, the performance penalty is small.
- Best not to use exceptions for common errors.

code-exception-2.cc

```
1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4
5  //-----------------------------------------------------------------------------
6  void work( const char * filename, const int size){
7
8      // Task 1: Open a file
9      ifstream file(filename);
10     file.exceptions( ifstream::failbit );
11
12     // Task 2: allocate some memory
13     double* array[50];
14     for(int i=0; i<50; ++i){
15         array[i] = new double[size];
16     }
17 }
18
19 //-----------------------------------------------------------------------------
20 int main(){
21
22     cout << "Trying to work..." << endl;
23     try{
24         // Try working...
25         work("data",100);
26     }
27     catch(exception& e){
28         // we could also use: catch(ifstream::failure e)
```

```
29          cout << "----------------------------------" << endl;
30          cout << "Cannot open file: " << e.what() << endl;
31      }
32
33      cout << "Trying to work again..." << endl;
34      try{
35          // Try working again...
36          work("code-exception-1.cc",1000000000);
37      }
38      catch(exception& e){
39          // we could also use: catch()
40          cout << "----------------------------------" << endl;
41          cout << "Cannot open file: " << e.what() << endl;
42      }
43
44      return 0;
45  }
```

Output

```
Trying to work...
----------------------------------
Cannot open file: basic_ios::clear
Trying to work again...
----------------------------------
Cannot open file: St9bad_alloc
```

- We only really need one `try/catch` block. I used two for demonstration.
- We can reference using the base class `exception` or the actual class generating the error.
- What happens when we `catch (exception e)`?

17.4 STL Exceptions

code-exception-3.cc

```
1  #include <iostream>
2  #include <fstream>
3  #include <stdexcept> // For vector out-of-bounds
4  #include <vector>
5  using namespace std;
6
7  //-----------------------------------------------------------------------------
8  int main(){
9
10     try{
11         vector<double> myData(5,2);
12         for(int i=0; i<7; ++i)
13             cout << myData.at(i) << endl;
14     }
15     catch(out_of_range& e){
16         cout << "Vector index is out of range" << endl;
17     }
18     catch(exception& e){
19         cout <<  e.what() << endl;
20     }
21
22     return 0;
23 }
```

- Can only catch this exception with `.at()`.
- Probably less efficient...

17.5 Creating Exceptions

code-exception-4.cc

```
1  #include <iostream>
2  #include <stdexcept> // stdexcept header file contains runtime_error
3  using namespace std;
4
5  // DivideByZeroException objects should be thrown by functions
6  // upon detecting division-by-zero exceptions
7  // ------------------------------------------------------------------------
8  class DivideByZeroException : public runtime_error{
9  public:
10    // constructor specifies default error message
11    DivideByZeroException() :runtime_error( "attempted to divide by zero" ) {}
12 };
13
14 // perform division and throw DivideByZeroException object if
15 // divide-by-zero exception occurs
16 // ------------------------------------------------------------------------
17 double quotient( int numerator, int denominator ){
18    // throw DivideByZeroException if trying to divide by zero
19    if ( denominator == 0 )
20       throw DivideByZeroException(); // terminate function
21
22    // return division result
23    return static_cast< double >( numerator ) / denominator;
24 }
25
26 // ------------------------------------------------------------------------
27 int main(){
28
29     try{
30         cout <<  quotient( 100, 10) << endl;
31         cout <<  quotient( 100, 5) << endl;
32         cout <<  quotient( 100, 0) << endl; // oops...
33     }
34     catch ( DivideByZeroException &e ){
35         cout << "Exception occurred: " << e.what() << endl;
36     }
37
38    return 0;
39 }
```

- You can throw any type of object (e.g. int, string, yourClass(), ect.)
- You will need to catch the type or use
- `catch( ... )` to catch everything.
  - Cannot determine the exception!

17.6 Unwinding

code-exception-5.cc

```
1  #include <iostream>
2  using std::cout;
3  using std::endl;
4
5  #include <stdexcept>
6  using std::runtime_error;
7
8  //-------------------------------------------------------------------------
9  // function3 throws run-time error
10 void function3(){
11    cout << "In function 3" << endl;
12    throw runtime_error( "runtime_error in function3" );
13 }
14
15 // function2 invokes function3
16 void function2() throw ( runtime_error ){
17    cout << "function3 is called inside function2" << endl;
18    function3();
19 }
20
21 // function1 invokes function2
22 void function1(){
23    cout << "function2 is called inside function1" << endl;
24    function2();
25 }
26
27 // demonstrate stack unwinding
28 //-------------------------------------------------------------------------
29 int main()
30 {
31    // invoke function1
32    try
33    {
34       cout << "function1 is called inside main" << endl;
35       function1(); // call function1 which throws runtime_error
36    }
37    catch ( runtime_error &e ){
38       cout << "Exception occurred: " << e.what() << endl;
39       cout << "Exception handled in main" << endl;
40    }
41
42    return 0;
43 } // end main
```

Output

```
function1 is called inside main
function2 is called inside function1
function3 is called inside function2
In function 3
Exception occurred: runtime_error in function3
Exception handled in main
```

17.7 The Andrew Sutton Method

code-exception-6.cc

```
1  #include <stdexcept>
2  using std::runtime_error;
3
4  // Step 1: Create a generic Exception class
5  //-------------------------------------------------------------------------
6  class Exception : public runtime_error{
7    public:
8      Exception(const char* s) : runtime_error(s){}
9  };
10
11 // Use it in functions and other classes...
12 //-------------------------------------------------------------------------
13 #include <iostream>
14 using namespace std;
15
16 void work(int a){
17     if(a==1) throw Exception("function1: a==1");
18 }
19
20 void task(int a){
21     if(a>5) throw -1;
22 }
23
24 void search(int a){
25     if(a<4) throw Exception("search(): a<4");
26 }
27
28 //-------------------------------------------------------------------------
29 int main(){
30
31     for(int num=0; num<10; ++num){
32
33       // Try everything...
34       try{
35         work(num);
36         task(num);
37         search(num);
38       }
39       // Catch the errors you specify
40       catch( exception &e){
41         cout << "Exception occured at " << num << " : " << e.what() << endl;
42       }
43       catch( int e){
44         cout << "Exception occured at " << num << endl;
45       }
46
47     }
48     return 0;
49 }
```

Output

```
Exception occured at 0 : search(): a<4
Exception occured at 1 : function1: a==1
Exception occured at 2 : search(): a<4
Exception occured at 3 : search(): a<4
Exception occured at 6
Exception occured at 7
Exception occured at 8
```

```
Exception occured at 9
```

STL: Things of Things

things.cc

```
#include <iostream>
#include <vector>
#include <list>
#include <map>
#include <iterator>
using namespace std;

int main(){

    // Vector..
    cout << "-------------------------------------------------------" << endl;
    cout << "        Vector                                         " << endl;
    cout << "-------------------------------------------------------" << endl;
    vector<int> temp(10,0);
    vector< vector<int> > data(10,temp);

    for(int r=0; r<data.size(); ++r){
        for(int c=0; c<data[r].size(); ++c){
            cout << data[r][c] << " ";
        }
        cout << endl;
    }









    cout << "-------------------------------------------------------" << endl;
    cout << "        List                                           " << endl;
    cout << "-------------------------------------------------------" << endl;
    typedef list<int>   inner;
    typedef list<inner> outer;

    inner tempList;
    tempList.assign(10,1);
    outer dataList;
    dataList.assign(10,tempList);

    for(outer::iterator r=dataList.begin(); r != dataList.end(); ++r){
        inner temp = *r;
        for(inner::iterator c = temp.begin(); c!=temp.end(); ++c){
            cout << *c << " ";
        }
        cout << endl;
    }
```

```
59
60
61
62
63
64
65
66
67      cout << "-------------------------------------------------------------" << endl;
68      cout << "          Map                                                " << endl;
69      cout << "-------------------------------------------------------------" << endl;
70      typedef map<int,double>  imap;
71      typedef map<int,imap>    omap;
72      omap dataMap;
73
74      for(int r=0; r<10; ++r){
75          for(int c=0; c<10; ++c){
76              dataMap[r][c] = 2.0;
77          }
78      }
79
80      for(omap::iterator r=dataMap.begin(); r!=dataMap.end(); ++r){
81          imap temp = r->second;
82          for(imap::iterator c=temp.begin(); c!=temp.end(); ++c){
83              cout << c->second << " ";
84          }
85          cout << endl;
86      }
87
88      return 0;
89  }
```

18 Boost

The C++ standard

- Language and library defined in 1998.
- Version 2.0 is expected in 2009.
- TR1 - technical report one contains many of these new features.
- 2/3rds of this comes from BOOST.

18.1 Where's the final clause?

code-auto-1.cc

```
1  #include <iostream>
2  #include "base.h"
3  using namespace std;
4
5  //---------------------------------------------------------------------------
6  void throwException(){
7      throw "breaking the try block";
8  }
9
10 //---------------------------------------------------------------------------
11 int main(){
12
13     try{
14         // Create a base class pointer
15         base* ptr = new derived();
16         ptr->behavior();
17
18         // Throw an exception
19         throwException();
20
21         cout << "Delete ptr" << endl;
22         delete ptr;
23     }
24     catch( ... ){
25         cout << "Memory leak" << endl;
26     }
27     return 0;
28 }
```

Problem

- `delete ptr` is never called.
- We need to put the ptr inside an object whose destructor will automatically release the memory.

18.2 Smart pointers (*std::auto_ptr*)

code-auto-2.cc

```
1  #include <iostream>
2  #include "base.h"
3  using namespace std;
4
5  //-----------------------------------------------------------------------------
6  int main(){
7
8      std::auto_ptr<base> p1(new base());
9      std::auto_ptr<base> p2(new derived());
10
11     p1->behavior();
12     p2->behavior();
13
14     std::auto_ptr<base> p3(p1);
15     // p1->behavior();  p1 is NULL
16     p3->behavior();
17
18     return 0;
19 }
```

18.3 Smarter pointers (*boost::shared_ptr*)

code-boost-4-shared.cc

```
1  #include <iostream>
2  #include "boost/shared_ptr.hpp"
3  using namespace std;
4
5  int main(){
6
7      boost::shared_ptr<int> ptr1(new int(10));
8      boost::shared_ptr<int> ptr2;
9
10     ptr2  = ptr1;
11     *ptr1 = 100;
12
13     cout << *ptr1 << " " << *ptr2 << endl;
14
15 }
```

boost::shared_ptr

- Reference counting smart pointer.
- Keeps track of how many objects point to a resource.
- Similar to *garbage collection*.

Bonus

- Since they work “as expected”, you can use in STL containers.

code-boost-5-STL.cc

```
1 #include <iostream>
2 #include <vector>
3 #include "boost/shared_ptr.hpp"
4 #include "base.h"
5 using namespace std;
6
7 int main(){
8
9     vector< boost::shared_ptr<base> > v;
10
11    v.push_back(boost::shared_ptr<base>(new base()));
12    v.push_back(boost::shared_ptr<base>(new base()));
13    v.push_back(boost::shared_ptr<base>(new base()));
14    v.push_back(boost::shared_ptr<base>(new derived()));
15    v.push_back(boost::shared_ptr<base>(new derived()));
16    v.push_back(boost::shared_ptr<base>(new derived()));
17
18    for(int i=0; i<v.size(); ++i)
19        v[i]->behavior();
20
21 }
```

Limitation

- Cannot use with dynamically allocated arrays...
- If you really need one, here it is:

code-boost-5-array.cc

```
1 #include <iostream>
2 #include "boost/shared_array.hpp"
3 #include "base.h"
4 using namespace std;
5
6 int main(){
7
8     int*  array1 = new int [10];
9     boost::shared_array<int> array2 (array1);
10
11    // Check address
12    cout << ( array1 == array2.get() ) << endl;
13    cout << ( array1 == &array2[0]   ) << endl;
14
15    // Check values
16    array1[5] = 100;
17    cout << ( array1[5] == array2[5]   ) << endl;
18
19 }
```

I bet you can't wait!

18.4 Rational Numbers

code-boost-3-rational.cc

```
1  #include <iostream>
2  #include "boost/rational.hpp"
3
4  int main(){
5
6      boost::rational<int> r1(10,6);
7      boost::rational<int> r2(25,4);
8
9      std::cout << "r1 = " << r1 << std::endl;
10
11     r2 = r1 + 15 + r2;
12
13     std::cout << "r2 = " << r2 << std::endl;
14
15     std::cout << "The gcd of 254 and 127 is: ";
16     std::cout << boost::gcd(254,127) << std::endl;
17
18     return 0;
19 }
```

Output

```
r1 = 5/3
r2 = 275/12
The gcd of 254 and 127 is: 127
```

18.5 String tokenizers

code-boost-6-tok.cc

```
1  #include <iostream>
2  #include <string>
3  #include "boost/tokenizer.hpp"
4  using namespace std;
5
6  int main(){
7
8      string s = "This is a test. This is, only a test.";
9
10     // Default separator is space and punctuation.
11     boost::tokenizer<> tok(s);
12     for(boost::tokenizer<>::iterator i=tok.begin(); i!=tok.end(); ++i){
13         cout << "<" << *i << "> ";
14     }
15     cout << endl;
16
17     s = ";;Hello|world||-foo--bar;yow;baz|";
18
19     // Specify the separator...
20     boost::char_separator<char> sep("-;|");
21     typedef boost::tokenizer<boost::char_separator<char> > tokenizer;
22     tokenizer tok2(s,sep);
23     for (tokenizer::iterator i = tok2.begin(); i != tok2.end(); ++i)
24         cout << "<" << *i << "> ";
25     cout << endl;
26
27 }
```

18.6 Casting

code-boost-8-cast.cc

```
1  #include <iostream>
2  #include <iterator>
3  #include <vector>
4  #include <boost/lexical_cast.hpp>
5
6  int main(int argc, char** argv){
7
8      // Lexical cast....
9      std::vector<int> args;
10
11     for(int i=1; i<argc; ++i){
12         try{
13           args.push_back(boost::lexical_cast<int>(argv[i]));
14         }
15         catch( ... ){}
16     }
17
18     copy(args.begin(), args.end(), std::ostream_iterator<int>(std::cout,"\n"));
19
20
21     return 0;
22 }
```

- Safer than `atoi` or `atof`.
- Also check out `numeric_cast`, `ploymophic_cast`, and `ploymophic_downcast`

18.7 uBLAS

What is it?

- Provides **B**asic **L**inear **A**lgebra **S**ubprograms (BLAS) functionality.

code-boost-7-ublas.cc

```
1  #include <iostream>
2  #include <algorithm>
3  #include <boost/numeric/ublas/vector.hpp>
4  #include <boost/numeric/ublas/matrix.hpp>
5  #include <boost/numeric/ublas/io.hpp>
6  #include <boost/numeric/ublas/triangular.hpp>
7  using namespace std;
8  namespace ublas = boost::numeric::ublas;
9
10 typedef  ublas::vector<double> realVector;
11 typedef  ublas::matrix<double> realMatrix;
12
13 int main(){
14
15     const int N = 10;
16     realVector x = ublas::scalar_vector<double>(N,10);
17     realVector y = ublas::scalar_vector<double>(N,20);
18     realMatrix A = ublas::identity_matrix<double>(N);
19
20     // ------ BLAS Level 1 ----------------------------------------------
21     // What is the distance between x and y?
22     cout << sqrt(ublas::inner_prod(x,y)) << endl;
23
24     // What is the Euclidean norm of x?
25     cout << ublas::norm_2(x) << endl;
26
27     // axpy...
28     const double a = 0.2;
29     realVector z = a*x + y;
30
31     // ------ BLAS Level 2 ----------------------------------------------
32     // A^t x
33     x = prod(trans(A), x);
34
35     // Solve linear system of equations... (symmetric case only)
36     y = solve(A,x,ublas::lower_tag());
37
38     // ------ BLAS Level 3 ----------------------------------------------
39     A = 10*A;
40     realMatrix B = prod(A, trans(A));
41
42     // ------ Use generic algorthims....
43     x(N-1) = 5;
44     sort(x.begin(), x.end());
45     cout << x(0) << " " << x(1) << endl;
46
47     return 0;
48 }
```

18.8 Random Number Generator

randomNumber.h

```
1  // Boost random number generator
2  #include <boost/generator_iterator.hpp>
3  #include <boost/random.hpp>
4  #include <boost/random/variate_generator.hpp>
5  #include <boost/random/mersenne_twister.hpp>
6  #include <boost/random/normal_distribution.hpp>
7
8  class randomNumber{
9    public:
10     randomNumber();
11     double gaussian();
12
13   private:
14
15     // Create a generator
16     boost::mt19937 generator;
17
18     // Create a gaussian distribution
19     boost::variate_generator<boost::mt19937&,
20     boost::normal_distribution<> > randn;
21
22 };
```

randomNumber.cc

```
1  #include "randomNumber.h"
2
3  randomNumber::randomNumber():randn(generator, boost::normal_distribution<>()){}
4
5  double randomNumber::gaussian(){
6      const double value = randn();
7      return value;
8  }
```

code-boost-2-randomNumber.cc

```
1  #include <iostream>
2  #include "randomNumber.h"
3
4  int main(){
5
6      randomNumber rnd;
7      for(int i=0; i<1000; ++i)
8          std::cout << rnd.gaussian() << std::endl;
9      return 0;
10 }
```

What have we learned?

C

- The differences between pointers and references and how to use them (and how make them `const`-correct).
- Understand how to use C-style arrays and C-style strings.
- Returning a local variable is dangerous.
- Variable scope and extent.
- How to create variable on the free-store and the stack.
- The passing objects to functions is best done through a reference or a const reference.
- Procedural programming often requires passing several objects and variables, which can be difficult to manage.
- How and when to overload a function.
- Default parameters allow for flexibility.

Object-Based C++

- Classes help us think about representation first, and implementation second.
- The role of the constructor and destructor.
- Initialization list are desirable (efficiency) and often necessary (e.g. const members).
- We need to be concerned with constructors, copy constructors, assignment operators, and destructors when dealing with dynamically allocated memory.
- Sometimes it makes sense to have member functions other times it does not.
- Sometimes a non-member function needs to be a `friend` of a class so that it may access private members.
- Some operators, like the assignment operator, should return a `this pointer`.
- Returning a local object is dangerous. Sometimes we must return a copy.

Inheritance

- Some objects exhibit an *is-a* relationship. In this case, public inheritance can decrease the complexity and length of the code you need to write. This relationship should always be *invariant over specialization.*
- We should never override a function when using public inheritance. This violates the *is-a* relationship and leads to name hiding.
- The base constructor is always called first, while its destructor is always called last.
- When creating an assignment operator or copy constructor, we need to be careful that we copy all the elements in our derived and base classes.

Polymorphism

- Sometimes we want to inherit the interface to a class, but may want, or require, the implementation to change. This can be done with **dynamic binding** using the `virtual` keyword.
- We should always make polymorphic base class destructors virtual so that we call the derived class destructors when a based class pointer or reference is destroyed.
- Polymorphic base classes should be a pointer or reference to avoid *slicing* part of the object away.
- Using the base scope operator (`using base::name;`) will allow overloaded names to be visible to derived classes.

Template Programming

- When a function or a class exhibits similar behavior on different types, it makes sense to write a template function or a template class.
- This is really just a code generation technique if the compiler does not see the signature of the function that it needs.
- Constants may be passed to templates as non-type parameters. These parameters can be used to create static arrays.

Standard Template Library (STL)

- Overloading is a powerful code generation technique, but is limited when particular data structures (containers) are used. The STL uses generic programming concepts such that one algorithm (function) can be used on several **containers** of various **types**.
- This is done through **iterators**, which abstract the way a container is stored in memory.
- When writing a generic function, a template can be used to store the type of object used to step-though a particular container. This could be an iterator or a pointer.
- This is why C-arrays and C-strings can be used with several generic algorithms even though they do not have iterators.
- There are advantages and disadvantages to each container.
- Containers can be sequential or associative.
- We can used iterators to traverse `vector`, `list`, `map`, and `set` containers.
- If a container is sorted, algorithms like `binary_search` and `equal_range` are more efficient than `find`.
- Often a container class will implement its own member functions when the corresponding generic algorithm cannot be used with its iterator type (e.g. list::sort()).

I do not expect you to memorize all the STL algorithms. If I ask you to use an algorithm, I will provide you with some documentation.

Exception Handling

- Exception handling should be used to process low-frequency or uncommon events.
- We should always throw/catch a reference or pointer exception because the `what()` methods is virtual.
- Andrew Sutton's method is great for students because it:
  - Does not require you to write a separate class for each exception.
  - Meaning that you will use it often, and
  - This will make finding exceptions in your code easier than debugging.
  - You only need a giant `try` statement in order to use this because exceptions unwind.
- Using an `auto_ptr` can ensure that if an exception is thrown, you will not have a *memory* or *resource* leak.

Homework Topics

- Reading data using `iostream`.
- Procedural programming styles.
- Template classes.
- Stepping through complex containers.
- Polymorphic pointers and factory functions.