C++ Programming for Scientists

Science & Technology Support
High Performance Computing

Ohio Supercomputer Center
1224 Kinnear Road
Columbus, OH 43212-1163

OSC

Features of C++ -- A Better C

1) **A Better C**

- Providing features that make common C errors unlikely
- C++ is a **superset** of C
- Extensions to C
- Improvements to C performance/ease-of-use.

Features of C++ -- Object-Oriented Programming

2) **Enabling Object-Oriented Programming (OOP)**

- Entirely new approach to programming: next major programming style beyond functional decomposition
  - **Model real world by making data types that replicate real objects**
  - **Facilitates code reuse: reuse and enhance existing libraries**
- Main program consists of creation and manipulation of objects
- ***Object = data + functions*** which allow you to interact with the data
- In general, data controlled ***only*** through these interface functions: **Data Encapsulation**
- C operators can work with your objects: **Operator Overloading**
- "Advanced" objects can incorporate features of "Simpler" objects: **Inheritance**
- With the same syntax, objects can act differently in different situations: **Polymorphism**

Features of C++ -- Templates

- **Function templates** allow the C++ programmer to write a single function that will work for all types of data.

- Similarly, **class templates** allow the C++ programmer to represent a family of classes.

- **The Standard Template Library (STL)** is a collection of extremely useful class templates, function templates and algorithms, allowing the programmer a wide variety of capabilities.

Table of Contents

A Better C

General:

- End-of-line Comments
- Boolean Data Type
- Type-casting Syntax
- Input/Output Streams
- Omnipresent Variable Declaration
- **`const`** Keyword
- Memory Allocation Syntax

Function-related:

- Improved Argument Type Checking/Strict Function Prototyping
- Interpretation of the **`void`** keyword
- Type-safe Linkage
- Default Arguments
- Function Overloading
- Inlined Functions
- Reference Declarations
- Operator Overloading!!

End-of-Line Comments

- Single line form of a comment begun with the // symbol
  - Everything from the delimiter // to the end-of-the-line is ignored
  - Reminiscent for the Fortran 90 use of ! for one-line commenting
- Familiar, multi-line style of commenting still works in C++

Warning: /* … */ commenting may not nest with some compilers

End-of-Line Comments Sample Program

```
#include <stdio.h>
main() {
   float r, theta; // polar coordinates
   float x,y,z; // Cartesian coordinates
                // only works for one line
   /* code might need later
   x=10; y=100;
   z=x*3+2-y/4;
   printf("z is %d \n",z);
   */
}
```

Boolean Data Type

- C++ now supports a Boolean data type identified by the keyword **bool**. Variables of this type will take on the values true and false. As in both C and C++ "**false**" is defined to be **0**. A "**true**" Boolean variable will take on the value **1**. Below is a small program that demonstrates how Boolean variables work:

```
#include <iostream.h>
main() {
   int i=4;
   int j=78;
   bool logic1;
   bool logic2;
   logic1 = (i==j);
   cout << "logic1=" << logic1 << endl;
   logic2 = (i<j);
   cout << "logic2=" << logic2 << endl;
}
```

```
logic1=0
logic2=1
```

Type-casting Syntax

- In C, override type casting was possible with the following (awkward) syntax:

```
int i;
(double) i
```

- In C++, a second (less awkward) syntax can also be used:

```
double(i)
char(3+'A')
```

- Additional usefulness of this new syntax will be seen later in the OOP Section.

Input/Output Streams

- Probably the strangest thing you would notice when looking at a C++ for the first time…
- Replacement of the **`stdio`** library of C with the **`iostream`** library of C++. For example the C++ version of the classic "Hello World" program looks as follows:

```
#include <iostream.h>
main() {
   cout << "Hello World\n";
}
```

Notice the 3 changes:

1 Include **`iostream.h`** instead of **`stdio.h`**

2 The **`cout`** keyword indicates that the output should go to standard out. There are corresponding keywords **`cin`** and **`cerr`** which represent standard in and standard error, respectively

3 Use of the ***insertion operator*** **`<<`** as opposed to the printf function. The corresponding ***extraction operator*** for input is **`>>`**

Advantages of `iostream` Approach

- Can extend the insertion/extraction operators to work exactly the way you want for your own data types (structures variables/objects)
- Can just use default format for quick output display (analogous to Fortran **`print *`**)
- Stream approach more type-safe. Avoid run-time mismatch of format specifiers and their "matching" arguments in **`printf`** and **`scanf`**
- Better performance: avoids run-time interpretation of format specifiers in **`stdio`** library functions

I/O Streams: Default Formats

```
#include <iostream.h>
main() {
   int i=44;
   float f=10.40;
   double d=2.718281828459045;
   char c='E';
   char *s="Logan";
   cout << c << endl;
   cout << s << endl;
   cout << f << endl;
   cout << d << endl;
   cout << i << endl;
   cout << s << c << f << i << endl;
   cout << &i << endl; }
```

```
E
Logan
10.4
2.71828
44
LoganE10.444
0xae41
```

Formatting I/O Streams

- Formatted output is performed by including **format manipulators** (such as the just seen **`endl`**) into the stream.
- Include the **`iomanip.h`** file in your code header.
- Some popular manipulators and their functions:

Format Manipulator	*Function*
`" "`	String of desired spaces
`"\t"`	Horizontal tab
`setw(w)`	Set field width to **`w`**
`setfill(c)`	Set fill character to **`c`** (blank is default)
`setprecision(p)`	Set float precision to **`p`**
`dec, oct, hex`	Use the indicated base

Formatting I/O Streams Example

```
#include <iostream.h>
#include <iomanip.h>
main() {
   cout << "[" << setw(6) << setfill('*') << 192;
   cout << "]" << endl;
   cout << hex << "[" << setw(6);
   cout << setfill('^') << 192 << "]" << endl;
   cout << setprecision(4) << 3.14159 << endl;
}
```

```
[***192]
[^^^^c0]
3.142
```

The Input Stream

- Analogous to output stream in syntax and use of format manipulators.

```
int c,d;
cin >> c >> d;
```

- **Advantage**: Avoid the common error of forgetting that the arguments to **`scanf`** are addresses.

Compare

```
cout << "Number of time steps:";
cin >> T;
```

to

```
printf("Number of time steps:");
scanf("%d",&T);
```

- The I/O stream approach also exists for files **`<fstream.h>`** and strings **`<strstream.h>`**

Using the cerr Stream

```
#include <iostream.h>
main() {
   int number=34;
   float divisor;
   cout << number << " divided by 2.3 equals "
        << number/2.3 << endl;
   cout < "Please enter your own divisor: ";
   cin >> divisor;
   if (divisor==0.0)
      cerr << "Cannot divide by zero" << endl;
   else
      cout << " Result is " << number/divisor << endl;
}
```

```
(981) conan% a.out
34 divided by 2.3 equals 14.7826
Please enter your own divisor: 0.0
Cannot divide by zero
(982) conan% a.out
34 divided by 2.3 equals 14.7826
Please enter your own divisor: 78.2
Result is 0.434783
```

Movable Variable Declarations

- Variables can be declared anywhere in C++ code. Local variables need not all be declared at the beginning of a function (or block).
  - Similar to implicit-typing and variable creation/use in Fortran.
- Variables come into existence when declared and cease existence when the present code block is exited.
  - Don't tie-up memory until you ***really*** need it and free it up when you don't need it anymore.
  - Improve code readability ("intuitively right" places for declaration)

Movable Variable Declarations Example

- Popular use of this feature is with loop counter.

```
float sum(float *a, int N) {
   float res=0.0;
   for (int i=0;i<N;++i)
      res += a[i];
   return(res);
}
```

Warning (*compiler-dependent*): scope of int extends to end of entire **`sum`** function. So if included another "**`int i`**" statement anywhere else in the **`sum`** function, would get compiler error for redeclaration of the same variable.

- Possible performance improvement: make loop counter **`register`** class as well

```
for (register int i=0;….
```

Symbolic Constants in C++

- In C, there exists the standard macro method for declaring symbolic constants:

```
#define LIMIT 2300
```

- Two problems with this approach are that macros lack type and the macro name is typically not available to a debugger.
- In C++ (and ANSI C), the `const` keyword can be used at declaration to indicate that the identifier **cannot be subsequently changed**:

```
const int LIMIT=2300;
```

`LIMIT` now has scope and type and is tracked in the name space used by the compiler and most debuggers. Making symbolic constants ALL CAPITAL LETTERS has long been a C/C++ convention.

- The `const` keyword can also be used in function argument list to insure that the argument cannot be changed in the body of the function:

```
void bobba(const int i) {
   i=5; // illegal!
   ...
```

C++ Memory Allocation Functions

- In C++, the **`malloc`** library of dynamic memory allocation functions are replaced with the **`new`** and **`delete`** operators built into the language.
- The justification for this change is that the "new" operators are easier to use: avoiding
  - the ancillary use of the C **`sizeof`** function and
  - extraneous type-casting
  - operator versus function call overhead
- Above is especially true with user-defined types.

Warning: Do not mix & match C's **`malloc()`** and **`free()`** with C++'s **`new`** and **`delete`**.

Dynamic Allocation Examples: Single Variable and Arrays

- Sample code for C++ dynamic allocation of a single variable type

```
float *fp;
fp=new float;
*fp=9.87;
cout << *fp << endl;
delete fp;
```

- As this program demonstrates, the **new** operator returns the address of a block of memory large enough to hold a variable of the argument type. The **delete** operator then frees up the memory when it is no longer needed.
- A similar procedure with slightly different syntax is used for dynamic memory allocation for entire arrays:

```
float *a;
a=new float[50];
a[44]=34.5;
delete [] a;
```

Rule of Thumb: Whenever **new** is followed by **[]**, **delete** should be too.

Dynamic Allocation Example: User-defined Data Types

- Perhaps, the most convincing proof of how easily new & delete allow dynamic memory control, is when working with user-defined data types. Consider this classic linked list example:

```
#include <iostream.h>

struct node {
   int id;
   double value;
   node *next;
};

main() {
   node *current;
   current=new node;
   current->id=78;
   current->value=45.67;
   cout << current->id << endl;
   cout << current->value << endl;
   delete current;
}
```

Type Checking

- In general, C++ uses very **strict type checking**.
- With respect to functions, C++ (as well as ANSI C) **requires that functions be prototyped** before their use. This is to insure that the compiler can check that
  - the ***proper number*** of arguments are passed
  - where possible and if needed, cast the actual arguments to the ***proper type***
  - the ***return type*** is properly used in the calling function

Type Checking Illustration

- To illustrate the power of function prototyping and argument checking, the following code

```
 1  #include <iostream.h>
 2  int firstchar(char *s);
 3
 4  main() {
 5     float name=3.0;
 6     int ascii;
 7     ascii=firstchar(name);
 8     cout << "Ascii code of first char is " << ascii << endl;
 9  }
10  int firstchar(char *s) {
11        return(int(s[0]));
12     }
```

- produces these compiler error messages

```
type.C: In function ‘int main()’:
type.C:7: argument passing to ‘char *’ from ‘float’
```

Functions with no arguments in C and C++

- In C, a function prototype with an empty argument list

```
extern int freak)();
```

  indicates that the argument list of the declared function is not prototyped and could have **any number of arguments** (it is also defined in a separate file).

- In C, a function prototype with a **`void`** argument list

```
extern int freak(void);
```

  indicates that the function has **no arguments**.

- Because C++ maintains such strict argument checking, an empty argument list means there are literally no arguments. So in C++, the two above **declarations are equivalent**.

Strict Argument Checking Example

- For example, consider the freak function declared in its own file **`freak.C`**:

```
int freak(int a) {
      return(a+5);
   }
```

- and a main program in a file called **`tstfrk.C`** that uses the freak function,

```
1  #include <stdio.h>
2  extern int freak();
3  main() {
4     int x=23;
5     printf("%d\n",freak(x));
6  }
```

- when compiled with the **`gcc`** command (for example), the following compile-time error messages are produced

```
gcc tstfrk.C freak.C
tstfrk.C: In function ‘int main()’:
tstfrk.C:2: too many arguments to function ‘int freak()’
tstfrk.C:5: at this point in file
```

- In C, the code in these two files would have run successfully.

Type-safe Linkage Example

- C++ takes type checking to a further level beyond ANSI C, by comparing a function prototyped in the “main” file and actually defined in a separate file.

- Consider the function **`colum`** defined in a file called **`colum.c`**

```
double colum(double a, double b) {
      return(a/2.0*(b+42.0));
   }
```

- and further consider the main source file, **`tstcol.c`**, which uses the **`colum`** function:

```
1  #include <stdio.h>
2  extern double colum(int a, double b);
3  main() {
4     double a=5.5,b=6.7,c;
5     c=colum(a,b);
6     printf("%f\n",c);
7  }
```

Type-safe Linkage Example Discussion

- In **`tstcol.c`**, the function prototype incorrectly declares the first dummy argument to be an **`int`** instead of a **`double`**. **A C compiler will not find this error and an incorrect value for `c` will be produced**.

NOTE: C programmers who get in the habit of putting function prototypes in their own include file can generally avoid the error.

- If this same code is compiled using a **C++ compiler, it will find the error** and prevent the creation of an incorrect executable. This is accomplished at link time through a process known as **name-mangling**. The function names actually used by the linker are encoded with argument-typing information.

Type-safe Linkage Example Output

- For example, using the preceding code example and the following compilation command, the following error messages would be produced.

```
CC tstcol.C colum.C
tstcol.C: In function ‘int main()’:
tstcol.C:5: warning: ‘double’ used for argument 1 of
  ‘colum(int,double)’

Undefined                       first referenced
symbol                          in file
colum_Fid                       /var/tmp/cca001cg1.o

ld: fatal: Symbol referencing errors. No output written to
  a.out
```

Default Arguments

- In C++, the programmer can set **default values for dummy arguments** in function definitions.
- If no actual value is passed to that dummy argument, the default value is used.
- Useful when a certain argument almost always has the same value when the function is used.
- In addition, this approach can reduce the number of actual arguments in the function reference.
- On the other hand, should comment on why/how the default value is used.

Default Arguments Example

- As illustration, consider the following program and its output:

```
#include <iostream.h>
void symm(float x, int k=13) {
      cout << "Argument 1 is " << x << endl;
      cout << "Argument 2 is " << k << endl;
   }
main() {
   symm(3.2,167);
   symm(5.4);
}
```

```
Argument 1 is 3.2
Argument 2 is 167
Argument 1 is 5.4
Argument 2 is 13
```

Function Overloading

- Often, a function needs to perform the same operation, but using arguments of different types.
- In C++, **multiple versions of a function with the same name but different type arguments** can be defined.
- In the main program, the "correct" version of the function is actually used by the compiler by examining the argument types. (Internally, this is done through name-mangling again…).
- Function overloading avoids having multiple functions with slightly different baroque names which essentially perform the same work.
- In the following program, a **`swap`** function is overloaded to work with both integers and doubles.

Function Overloading Example

```
#include <iostream.h>
void swap(int* i, int* j) {
      int temp;
      cout << "int swap called" << endl;
      temp=*i;
      *i=*j;
      *j=temp; }
void swap(double* x, double* y) {
      double temp;
      cout << "double swap called" << endl;
      temp=*x;
      *x=*y;
      *y=temp; }
main() {
   int a=5,b=23;
   double r=34.5,s=1245.78;
   swap (&a,&b);
   swap (&r,&s);
   cout << "a is " << a << " b is " << b << endl;
   cout << "r is " << r << " s is " << s << endl; }
```

```
int swap called
double swap called
a is 23 b is 5
r is 1245.78 s is 34.5
```

Inlined Functions

- In C++, functions can be declared as inlined. Inlining is a general optimization technique which causes any reference to a function to be **replaced by the actual code** that makes up the function. The function body is input ***in*** the ***line*** where the function reference occurs. The user programs in their own style and the compiler does the inlining automatically. (Especially useful for small functions referenced often).

- The advantages to inlining are:
  - Don't pay the overhead time-cost of the function call itself
  - Unlike macros (the alternative approach), inlined functions can be prototyped and thereby type-checked
  - Less prone to errors than macros

- The disadvantage of inlining (and macros) is the assembly language "text bloat" which will occur. In addition, the actual code must be known to the compiler, thus functions in a run-time library cannot be inlined.

Macros -- C Example

- The following C code shows that **macros can produce incorrect results** due to their literal substitution of arguments:

```
#include <stdio.h>
#define MUL(a,b) a*b
main() {
   int x,y,z;
   x=10;y=100;
   z=MUL(x*3+2,y/4);
   printf("z is %d \n",z);
}
```

```
z is 80
```

- The correct result we expected **`(10*3+2)*(100/4)=800`** was not calculated because the macro substitution resulted in the expression

  `x*3+2*y/4`

  which due to operator precedence gives the value **`80`**.

Inlined Functions -- C++ Example

- The problem is solved in the following C++ program using inlining:

```
#include <iostream.h>
inline int mul(int a, int b) {
      return a*b;
   }
main() {
   int x,y,z;
   x=10;y=100;
   z=mul(x*3+2,y/4);
   cout << z << endl;
}
```

```
800
```

Call-by-Reference

- In C, the only way actual arguments are passed to dummy arguments when referencing a function is **call-by-value**.
  - The value of the actual argument is passed into the dummy argument. Any changes the function makes to the dummy arguments will not affect the actual arguments.
- C does provide an indirect method for changing the value of the actual arguments, namely **call-by-address**.
  - Addresses are passed from the actual to the dummy argument and the indirection operator ***** is used to change the contents of the actual argument.

Call-by-Address -- C Example

- The following C code demonstrates the use of call-by-address

```
#include <stdio.h>
struct item {
      char* name;
      float price;
      int quantity;
   };
void increase(struct item* p) {
      (*p).price=1.10*(*p).price;
      p->quantity=p->quantity+10;
   }
main() {
   struct item chicken={"Holly Farms",5.00,20};
   increase(&chicken);
   printf("New price is %0.2f\n",chicken.price);
}
```

```
New price is 5.50
```

Call-by-Reference -- C Example Discussion

- The drawbacks to the C call-by-address approach is that the function body is difficult to interpret because of the necessary ***** and **->** operators. In addition, one must remember to use the address operator **&** for the actual argument in the function reference itself.

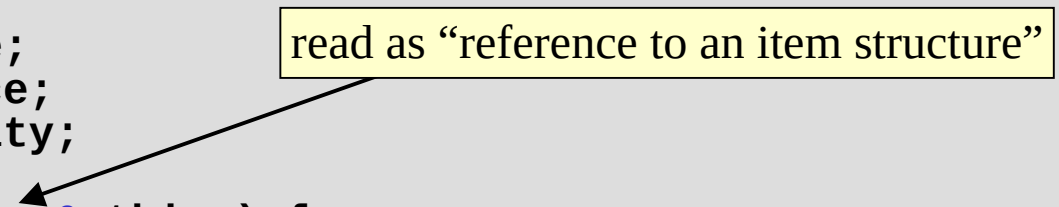
- For these reasons, C++ has developed a true **call-by-reference**. If a dummy argument is declared as a reference to the actual argument -- by using the reference operator **&** (again!) -- it picks up the address of the actual argument (automatically) and can change the contents of that address.

- ***Conceptually, the referenced actual argument just gets a new name in the function: whatever is done to the dummy argument is also done to the actual argument.***

Call-by-Reference -- C++ Example

- The C++ version of the previous code using call-by-reference is:

```
#include <iostream.h>
struct item {
      char* name;
      float price;
      int quantity;
   };
void increase(item& thing) {
      thing.price=1.10*thing.price;
      thing.quantity=thing.quantity+10;
   }
main() {
   item chicken={"Holly Farms",5.00,20};
   increase(chicken);
   cout << "New price is " << chicken.price << endl;
}
```




```
New price is 5.5
```

Operator Overloading

- On of the more powerful features of C++ is that programmers can define the basic C++ **built-in operators** -- such as **+**, *****, **[]**, **!**, for example -- to **work with user-defined types**. Of course, the operators continue to work as originally designed for basic types: thus the term overloading.
  - This capability makes for sensible and more readable main programs and is similar, in spirit, to function overloading.
  - Keep in mind also that although you can use operator overloading to define an operator to do anything you want, it is **best to make the new operator definition in some way analogous to the normal use of the operator**.
- The syntax for operator overloading looks exactly like the syntax for a function definition except that the function name is replaced by the name

```
operator<operator symbol>
```

Operator Overloading Usage

- A sample operator overload prototype might look as follows:

```
clock operator+(const clock& a, const clock& b);
```

- where `clock` is some user-defined structure and the body of this prototype will define what "`+`" will do to two clock structures (perhaps, add the hours members to correct for different time zones).

NOTE: the use of the `const` keyword in operator overload prototype. This is to insure that the operator cannot change the value of its operands, only produce a result. NOTE: also the use of call-by-reference arguments. Both noted procedures are suggested, not required.

- It should be pointed out that operator overloading **only defines what the operator does for user-defined types**. It **does not change** the operator's precedence, direction of associativity, or number of operands. In addition, you cannot make up your own new operator symbols.

Operator Overloading Example

```
#include <iostream.h>
struct complex { double real;
      double img;};
complex operator+(const complex& a, const complex& b) {
   complex res;
   res.real=a.real + b.real;
   res.img=a.img + b.img;
   return(res); }
complex operator*(const complex& a, const complex& b) {
   complex res;
   res.real = a.real*b.real - a.img*b.img;
   res.img = a.real*b.img + a.img*b.real;
   return(res); }
complex operator!(const complex& a){
   complex res;
   res.real = a.real;
   res.img = -a.img;
   return(res); }
main() {
   static complex x={1.0,2.0},y={3.0,4.0},z;
   z=x+y;
   cout << z.real << " +i" << z.img << endl;
   z=x*y;
   cout << z.real << " +i" << z.img << endl;
   z=x*(!x);
   cout << z.real << " +i" << z.img << endl; }
```

```
4 +i6
-5 +i10
5+i0
```

Object-Oriented Programming

- Classes and Objects
- Constructors and Destructors
- Object assignment & Type Casting
- Operators with Objects
- Friend Functions
- Using Objects with the I/O stream
- Static Class Members
- Inheritance
- Virtual Functions and Polymorphism

C++ Classes

- A **class is a user-defined data type**. You can think of it as an extended and improved structure data type. **An object is a variable of a certain class type**. It is often said that an object is an instantiation of the class.

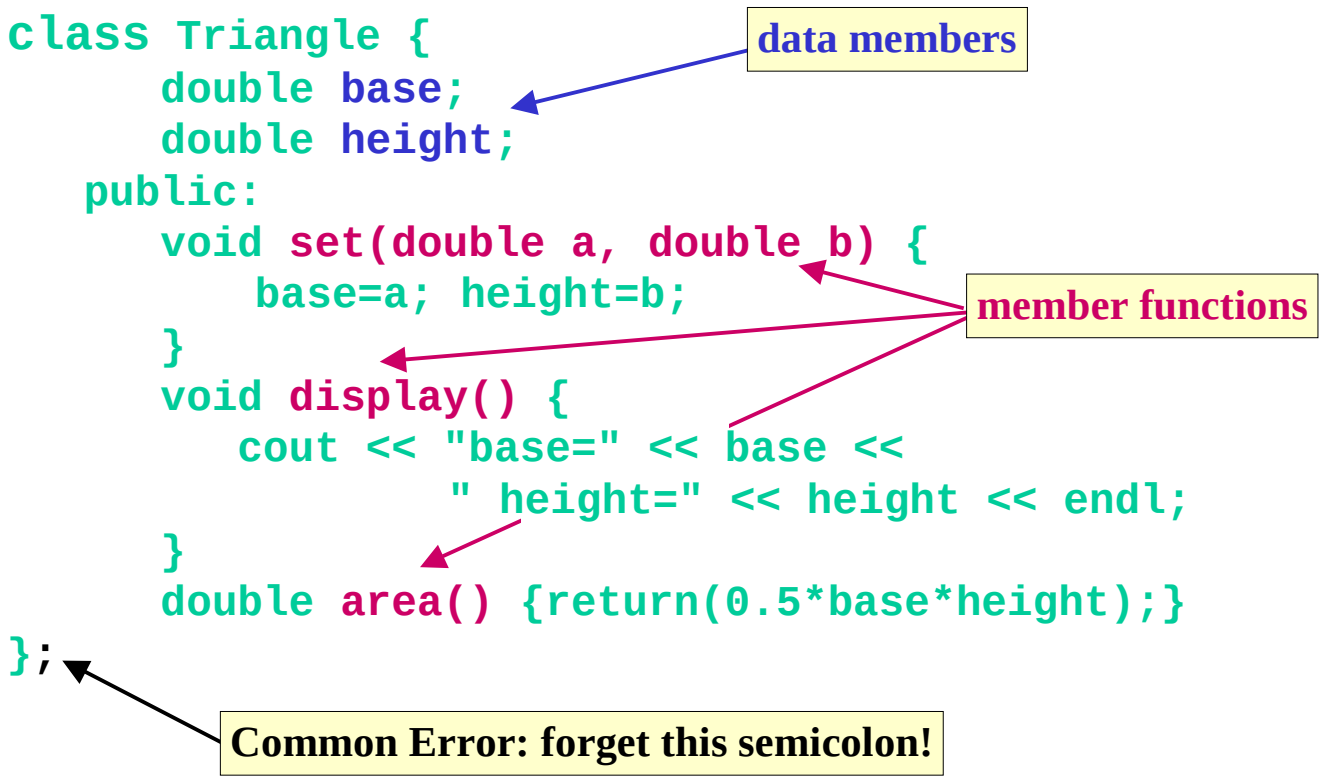
- Intrinsic data-types, **`float`** for example have values they can contain (real numbers) and a set of operations (**`+`**,**`/`**,**`*`**,etc) that can be applied to variables of type **`float`**.

- These same concepts are carried over to class types. The values a class can contain are called its **data members**. The set of operations that can be applied to a certain class object are called its **member functions**.

- After a class has been declared, we will see in the rest of the course how its objects can be treated in the same manner and syntax as “normal” data-type variables.

C++ Class Example

- In the following code fragment, a class `Triangle` is declared:



`Triangle` Class Example

- In this declaration of the class **`Triangle`** (it is customary to capitalize the first letter of a class type), there are two data members -- **`base`** and **`height`** -- and three member functions -- **`set`**, **`display`**, and **`area`**. The member functions represent three typical tasks one might want to perform with a Triangle object. In many ways the declaration of a class looks like the declaration of a structure but with functions now being allowed as members.
- Here is an example of some **`main`** code that would use the Triangle Class data type:

```
main() {
   Triangle t;
   t.set(2.0,3.0);
   t.display();
   cout << "area is " << t.area() << endl; }
```

`t` is an object of class **`Triangle`**

```
base=2 height=3
area is 3
```

- It should be noted in this code that the same "dot" operator to access a member of a structure is also used to access the member of a class. Specifically, **member functions are invoked with the traditional "dot" operator**.

Access to Class Members

- The user has complete control over which parts of a program can alter and/or use the members (data and functions) of a class. There are three types of member access allowed:

Access-Specifier	*Description*
`public`	Accessible to all functions in the program (everyone)
`private`	Accessible only to member functions of the same class
`protected`	Accessible only to functions of same or derived classes

- The default access classification for classes is **`private`**. As in the class **`Triangle`** previously declared, once an access-specifier -- like **`public`** -- appears, all members following the **`public`** keyword are declared public, until a different access-specifier is typed in. Thus, all three member functions of **`Triangle`** are public.

Controlling Access to Class Members

- Controlling the access to class members facilitates the safe reuse of classes in different programs, and avoids improper addressing and name collisions of data. The practice of using public member functions to indirectly set, change, and use private data members is called **Data Hiding**.

- Consider the following class **`Access`**:

```
class Access {
      int x;
   public:
      float y;
      int z;
      void set(int a, float b, int c) {
         x=a; y=b; z=c; }
      void display() {
         cout <<x<<" "<<y<<" "<<z << endl; }
};
```

Controlling Access Example

- Which contains a mixture of public and private data members. The proper an improper accessing of these members is illustrated in the following **`main`** code.

```
main() {
   Access w;
   Access *p;
   p=&w;
   w.set(1,2.45,3);
   w.display();
   w.y=4.343434
   w.display();
   // w.x=6;  ILLEGAL! Private member
   p->z=32
   w.display();
}
```

```
1 2.45 3
1 4.34343 3
1 4.34343 32
```

Member Functions

- As we have already seen, an interesting feature of C++ classes (unlike C structures) is that they can contain functions as members. In the class examples we have shown so far the member functions have been defined within the declaration of the class itself. The actual requirement is less severe: **member functions must be declared within the class declaration**, that is, their prototype must appear there. They **can actually be defined outside the class declaration**. One advantage of defining the member function inside the class is that it will **automatically be inlined**.

- In terms of access, member functions -- wherever they are defined -- have access to all private, protected, and public class members (data & function). When the member function is defined outside the class definition the **scope resolution operator** -- **::** -- must be used with the class name to indicate that the function belongs to that class.

Member Function Definitions/Declarations

- In the following code fragment, the member function **setx** is **defined** in the declaration of class **Play**, while the member functions **showx** and **incx** are only **declared** there:

```
class Play {
      int x;
   public:
      void setx(int a) {
         x=a;
      }
      void showx();
      void incx(int del);
};
```

- The following program uses the **Play** class and shows the syntax for defining "outside" member functions.

Member Functions Example

```
#include <iostream.h>
class Play {
      int x;
   public:
      void setx(int a){
         x=a; }
      void showx();
      void incx(int del); };
void Play::showx(){
   cout << "x is " << x << endl; }
inline void Play::incx(int del) {
   x += del; }
main() {
   Play fun;
   fun.setx(5);
   fun.showx();
   fun.incx(4);
   fun.showx();
}
```

```
x is 5
x is 9
```

The Constructor Function

Consider what happens when the following code is executed:

```
double x=4.567989;
```

- The basic-type variable `x` is brought into scope, enough memory for a double-type is allocated to `x` and that memory is initialized with a value.

- Given that an underlying philosophy of C++ is that derived-types can and should be treated the same (conceptually and syntactically) as intrinsic types, one would expect a similar initialization sequence for class variables.

- This “construction” of a class variable is accomplished through a **constructor function**.

Constructor Properties

Here are the basic properties of a constructor

- Special member function
- Has the same name as the class
- Is invoked automatically each time a new variable of its class is created (declared, dynamically-allocated, type conversion to the class)
- Cannot be invoked explicitly
- Has no return type (not even **`void`**)
- **Typically uses its arguments (if any) to initialize the new object's data members**
- Can have default arguments and be overloaded
- Default (no-op) constructor is provided by the compiler if you do not provide one.

Advice: Always define your own default constructor (and destructor) -- even if they are empty -- you may need to fill them later…

Constructor Example

```
 1  #include <iostream.h>
 2  class Bookshelf {
 3        float width; float height; int numshelves;
 4     public:
 5        Bookshelf(float i=4.0,float j=6.0, int k=5) {
 6           width=i;height=j;numshelves=k;
 7           cout << "Another Bookshelf made: " << width << "x"
 8                << height << " with " << numshelves << " shelves"
 9                << endl;} };
10  main() {
11     Bookshelf normal;  Bookshelf wide(8.0);
12     Bookshelf tall(4.0,9.0);  Bookshelf* custom;
13     custom=new Bookshelf(6.0,3.0,3);
14     for (int i=10; i<=15; i+=5) {
15        Bookshelf paperback(4.0,6.0,i); }
16  }
```

```
Another Bookshelf made: 4x6 with 5 shelves
Another Bookshelf made: 8x6 with 5 shelves
Another Bookshelf made: 4x9 with 5 shelves
Another Bookshelf made: 6x3 with 3 shelves
Another Bookshelf made: 4x6 with 10 shelves
Another Bookshelf made: 4x6 with 15 shelves
```

The Destructor Function

- The complementary function to the constructor. The destructor allows the user to perform any necessary “clean-up” after a class variable is destroyed.

- Destructors have the following properties:
  - Another special member function
  - Has the same name as its class, preceded by a tilde (**~**)
  - Implicitly invoked each time a class object is destroyed (goes out of scope [local variable], dynamically de-allocated, temporary object made during a type conversion no longer needed)
  - Has no return type (not even **`void`**)
  - **Has no arguments and cannot be overloaded**
  - Default (no-op) destructor is provided by compiler (if you don’t)
  - **Typically used for classes whose member functions perform dynamic memory allocation: destructor releases the memory**

Invoking Constructors & Destructors Example

```
 1  #include <iostream.h>
 2  class Number {
 3        int i;
 4     public:
 5        Number(int a) {
 6           i=a;
 7           cout << "Created Number "
 8                << i << endl;}
 9        ~Number() {
10           cout << "Destroyed Number "
11                << i << endl;}
12  };
13  Number x(1);
14  main() {
15     Number y(2);
16     Number* np;
17     np=new Number(3);
18     delete np;
19     for (int i=4; i<7; ++i) {
20        Number z(i); }
21  }
```

```
Created Number 1
Created Number 2
Created Number 3
Destroyed Number 3
Created Number 4
Destroyed Number 4
Created Number 5
Destroyed Number 5
Created Number 6
Destroyed Number 6
Destroyed Number 2
Destroyed Number 1
```

Deallocating Destructors Example

```
 1  #include <iostream.h>
 2  class Release {
 3        int* p;
 4     public:
 5        Release(int a) {
 6           p=new int; *p=a;
 7           cout << "Allocated int: "
 8                << *p << endl;}
 9        ~Release() {
10          cout << "Deallocated int: "
11                << *p << endl;
12          delete p; }
13  };
14
15  main() {
16     Release x(1);
17     Release* rp;
18     rp=new Release(2);
19     delete rp;
20     for (int i=3; i<5; ++i) {
21        Release z(i); }
22  }
```

```
Allocated int: 1
Allocated int: 2
Deallocated int: 2
Allocated int: 3
Deallocated int: 3
Allocated int: 4
Deallocated int: 4
Deallocated int: 1
```

Copying Objects

- Is it possible to copy one object of a class into another object of a class? The answer is yes and the assignment operator "**=**" has been designed so that the copying is done with “natural” syntax.

- If you use an assignment statement to copy one object to another a **data member by data member copy** is performed. That is, each member of the object is copied in turn.

- The program on the following page copies objects.

Copying Objects Example

```
#include <iostream.h>
class Bookshelf {
      float width; float height; int numshelves;
   public:
      Bookshelf(float i=4.0,float j=6.0, int k=5) {
         width=i;height=j;numshelves=k; }
      display() {
         cout << "Bookshelf is " << width << "x"
              << height << " with " << numshelves
              << " shelves" << endl;} };
main() {
   Bookshelf library(6.25,3.50,4);
   Bookshelf bedroom;
   bedroom.display()
   bedroom=library;
   bedroom.display()
}
```

```
Bookshelf is 4x6 with 5 shelves
Bookshelf is 6.5x3.5 with 4 shelves
```

Copying Objects with Pointer Members

- **Potential problems** with the member-by-member copying action of the assignment statement can arise when there are **pointer data members**.

- When pointers are included in the object, the data items that the pointers reference are not actually copied-only the addresses themselves. Because of this, a copied object can inadvertently alter the contents of the original object.

- Consider the code on the following page.

Copying Objects with Pointers Example

```
#include <iostream.h>
class Ints {
      int x; int *p;
   public:
      void setp(int a) { *p=a; }
      Ints (int a=1) { x=a; p=new int; *p=a; }
      void display() {
         cout << "x=" << x << " *p=" << *p << endl; }
};
main() {
   Ints s(3);
   Ints t;
   s.display();    t.display();
   t=s;
   t.display();
   t.setp(8);
   t.display();    s.display();
}
```

```
x=3 *p=3
x=1 *p=1
x=3 *p=3
x=3 *p=8
x=3 *p=8
```

Contents of **s** changed when working with copy **t**!!

Copy Constructor

- The copy constructor is used to make a new class object **safely duplicate** an existing class object. The salient properties of a copy constructor are:
  - Special type of constructor member function
  - Has exactly one argument which is a **reference** to the class type
  - Invoked with the following syntax
    ***class-name new-object-name*(*existing-object-name*)**
  - Typically defined when the class contains pointer members and the assignment operator is inappropriate
  - The copy constructor produces what is referred to as **deep copying** whereas the assignment operator performs **shallow copying**.
- The program on the following page fixes the problem just presented with our **`Ints`** class and its shallow copying. The copy constructor shown on the next page actually creates a **new address** for the copied class's **`int`** pointer and **not the same address** as the original.

Copy Constructor Example

```
#include <iostream.h>
class Ints {
      int x; int *p;
   public:
      void setp(int a) { *p=a; }
      Ints (int a=1) { x=a; p=new int; *p=a; }
      Ints(Ints& r) { p=new int; x=r.x; *p=*(r.p); }
      void display() {
         cout << "x=" << x << " *p=" << *p << endl; }
};
main() {
   Ints s(3);
   Ints t;
   s.display();
   Ints t(s);      // replaces t=s;
   t.display();
   t.setp(8);
   t.display();    s.display();
}
```

```
x=3 *p=3
x=3 *p=3
x=3 *p=8
x=3 *p=3
```

s is unchanged by actions on **t**!!

Conversion Constructor

- Yet another type of constructor **enables type conversion from other types to the class type**. It is, naturally, called a conversion constructor.

- The conversion constructor has **exactly** one argument of a ***type other than the class type***.

  – If the compiler encounters a “mixed-type” operation, the conversion constructor is used to make a temporary object of the class type and use this converted variable in the calculations.

Note: the data members of classes involved in conversion operation often have to be **made public** since they will be used in non-member functions.

Conversion Constructor Example

- Consider the following two classes for English and Metric distances and the conversion constructor to go from Metric units to English units:

```
class Metric {
   public:
      float cm;
      Metric (float x) { cm=x; }
};
class English {
   public:
      float in;
      English(float x) { in=x; }
      English(Metric m) {
         in=m.cm/2.54;
         cout << "Convert construct invoked" << endl; }
      English add(English y) { return(in+y.in); }
      void display() { cout << "Length is " << in
                            << " inches" << endl; }
};
```

Conversion Constructor Example Continued

- Here is the main program that demonstrates how the conversion works:

```
main() {
   English d1(6.25);
   English d2(3.375);
   d1=d1.add(d2);
   d1.display();
   Metric m1(30.48);
   d1=d1.add(m1);
   d1.display();
}
```

```
Length is 9.625 inches
Convert construct invoked
Length is 21.625 inches
```

Operator Overloading and C++ Classes

- As we have seen previously, C++ offers the ability to overload built-in operators so that they can work with user-defined types. Can operator overloading be applied to C++ classes? Yes: all that is needed is to **make the definition function for the operator be a member function of the class**.

- Consider in the previous program the definition of the function **add** for the **`English`** class which resulted in the somewhat awkward looking "addition" statement:

```
d1=d1.add(d2);
```

- The example on the following pages is the previous program rewritten with the **`+`** operator overloaded to work with **`English`** class variables. Notice how much more readable and sensible the main program becomes.

Operator Overloading Example

```
class Metric {
   public:
      float cm;
      Metric (float x) { cm=x; }
};
class English {
   public:
      float in;
      English() {}
      English(float x) { in=x; }
      English(Metric m) { in=m.cm/2.54; }
      English operator+(const English& a) {
         English res;
         res.in=in+a.in);
         return res; }
      void display() { cout << "Length is " << in
                            << " inches" << endl; }
};
```

Operator Overloading Example Continued

```
main() {
   English d1(6.25);
   English d2(3.375);
   d1=d1+d2;
   d1.display();
   Metric m1(30.48);
   d1=d1+m1;
   d1.display();
}
```

```
Length is 9.625 inches
Length is 21.625 inches
```

Scientific Application: Adding Stellar Magnitudes

- Since the time of the ancient Greek astronomers, the brightness of a star has been measured by its magnitude m, a real number. The lower the magnitude, the brighter the star: in fact, the brightest stars have negative magnitudes. Stellar magnitudes are actually measured on a logarithmic scale in which

$$m = 2.512 \text{ x } \log \frac{F_0}{F}$$

- where F is the flux from the star and F_0 is the flux from a zero-magnitude star.
- Often astronomers will have two stars in their field of view and would like to know the magnitude of the stars combined. One simply cannot add together the magnitudes of the separate stars since it is only physically correct to add the **fluxes** from the stars.
- In the program on the next page, the "**+**" operator is overloaded to correctly work for the addition of stellar magnitudes. Again, notice how readable and sensible the statements of the main program are.

Adding Stellar Magnitudes: Example Program

```
#include <iostream.h>
#include <math.h>
class Star {
      double mag;
   public:
      Star() {}
      Star(double x) { mag=x; }
      Star operator+(const Star& a) {
         Star res;
         res.mag=2.512*log10(1.0/(pow(10.0,-mag/2.512)+
                 pow(10.0,-a.mag/2.512)));
         return res; }
   void display() { cout << "Magnitude is " << mag <<endl; }
};
main() {
   Star Deneb(1.26);
   Star Aldebaran(0.86);
   Star Field;
   Field=Deneb+Aldebaran;
   Field.display();
}
```

```
Magnitude is 0.285582
```

Mathematical Application: Array-bounds Overflow

- A long-standing and common error made by programmers is to reference an array element that is beyond the bounds of the array. The results are unpredictable depending on what binary pattern is actually located in the (incorrect) memory location that is used. This classic problem is demonstrated in the following C program:

```
#include <stdio.h>
main() {
   int x[20];
   int i;
   for (i=0; i<20; i++)
      x[i]=2*i;
   for (i=0; i<=30; i+=5)
      printf("At index %d: value is %d\n",i,x[i]); }
```

```
At index 0: value is 0
At index 5: value is 10
At index 10: value is 20
At index 15: value is 30
At index 20: value is 5
At index 25: value is -268436436
At index 30: value is 0
```

Array-bounds Overflow: C++ Example Program

- In C++, one can redefine the array element reference operator "[]" to check to make sure that a correct index is used before actually getting an array element. Thus, we see that operators can be overloaded **not only to work normally** with classes and structures of a certain type, but that **their capabilities can be extended** to include safety and error-checking.
- Consider the same program on the previous pages, but written in terms of C++ classes and operator overloading:

```
#include <iostream.h>
class Intarray {
    int *data,size;
  public:
    Intarray(int sz=1) {
      if (sz<1) { cout << "Intarray: size must be >1, not "
                  << sz << endl; exit(1); }
      size=sz; data=new int[sz]; } }
    int& operator[](int index) {
      if (index<0 || index>=size) {
         cout << "\n\nIntarray: out of bounds, index=" << index
              << ", should be from 0 to " << size-1 << endl;
         exit(1); }
      return(*(data+index)); }
};
```

Array-bounds Overflow: C++ Example Program Continued

```
main() {
   Intarray x(20);
   for (register int i=0; i<20; i++);
      x[i]=2*i;
   for (register int i=0; i<=30; i+=5)
      cout << "At index " << i << " value is" << x[i]
           << endl;
}
```

```
At index 0: value is 0
At index 5: value is 10
At index 10: value is 20
At index 15: value is 30
At index 20: value is


Intarray: out of bounds, index=20, should be from 0 to 19
```

Conversion Operator Functions

- Recall that a Conversion Constructor can convert a variable to a class type from another type.

- To convert in the opposite direction (i.e., ***from* the class type *to* another type**), one has to use a special member operator function.

  – In place of the operator symbol the name of the type to-be-converted to is used.

- In the following program we revisit the English to Metric conversion program but this time with an English class operator function which converts inches to cm:

Conversion Operator Function Example

```
class Metric {
   public:
      float cm;
      Metric() {}
      Metric(float x) { cm=x; }
      void display() {
         cout << "Length is " << cm << " cm" << endl; }
      Metric operator+(const Metric& a) {
         Metric res;
         res.cm=cm+a.cm;
         return(res); }
};
class English {
   public:
      float in;
      English(float x) { in=x; }
      operator Metric() {
         Metric m;
         m.cm=in*2.54;
         cout << "Eng-to-Met conversion occurred" << endl;
         return(m); }
};
```

Conversion Operator Function Example Continued

```
main() {
   Metric m1(30.50);
   Metric m2(2.60);
   m1=m1+m2;
   m1.display();
   English d(3.937);
   m1=m1+d;
   m1.display();
}
```

```
Length is 33.1 cm
Eng-to-Met conversion occurred
Length is 43.1 cm
```

Friend Functions

- In general only **`public`** members of a class are accessible to functions that are not member functions.

- However, a class can declare a function to be a friend function in which case the **friend function has access to *all* members of the class** (including private).
  - Friend status is granted ***by*** the class ***to*** the function and is a feature of the class design.
  - Friend functions are often used when a calculation needs access to the private members of two objects of the same class.

- To specify a friend function, **declare** the function in the member list of the class, preceded by the **keyword `friend`**.
  - The function may actually be **defined** as a global or a member function.

Friend Function Example

```
#include <iostream.h>
#include <math.h>
class Point {
      double x,y,z;
   public:
      Point(double a,double b,double c) {
         x=a; y=b; z=c; }
      friend double distance(Point& r, Point& s);
};
double distance(Point& r, Point& s) {
   return(sqrt(pow(r.x-s.x,2)+pow(r.y-s.y,2)+pow(r.z-s.z,2)));
}
main() {
   Point center(1.3,5.6,9.8);
   Point edge(12.4,34.6,56.7);
   cout << "Distance between the points is "
        << distance(center,edge) << endl;
}
```

```
Distance between the points is 56.2478
```

Overloading the I/O stream operators

- This section will describe how to overload the output stream operator **<<** to work with your own classes. A similar procedure is used for the input stream operator **>>**.
- Before overloading **<<** for user-defined classes we need to have a little deeper understanding of how it works. There is a built-in C++ class defined for us called **ostream** (you can see the declaration of this in the **iostream.h** include file) . In fact **cout**, and **cerr** are just objects of this class. So the following code

```
int i;
cout << i;
```

- is telling us that **<<** is a binary operator where the right-hand operand is a basic type (in this case integer) and the left-hand operator is an object of class **ostream**.

Overloading the I/O stream operators Continued

- Therefore, the prototype for our overload function must have the form

```
ostream& operator<<(ostream& argl, X argr);
```

- where **X** is the class name we wish to output in a certain manner. The reason this function returns a reference to an **ostream** class is to enable "chaining together" of **<<** operations:

```
cout << X << "Just outputted my own object\n";
```

- is executed with a L-R associativity:

```
(cout << X) << "Just outputted my own object\n";
```

- Lastly, since one usually wants to output private data member the overload function is typically made a friend of the class.

Overloading the I/O stream operators Example

```
#include <iostream.h>
class Point {
      double x,y,z;
   public:
      Point() {}
      Point(double a,double b,double c) { x=a; y=b; z=c; }
      friend ostream& operator<<(ostream& os,Point p);
      friend istream& operator>>(istream& is,Point& p); };
ostream& operator<<(ostream& os,Point p) {
   os <<'('<<p.x<<','<<p.y<<','<<p.z<<')'<<endl;
   return(os); }
istream& operator>>(istream& is,Point& p) {
   cout << "Enter x coordinate of the point: "; is >> p.x;
   cout << "Enter y coordinate of the point: "; is >> p.y;
   cout << "Enter z coordinate of the point: "; is >> p.z;
   return(is); }
main() {
   Point center;
   cin >> center;    cout << center; }
```

```
Enter x coordinate of the point: 3.4
Enter y coordinate of the point: 8.9
Enter z coordinate of the point: 12.3
(3.4,8.9,12.3)
```

Static Data Members

- A data member of a class can be declared **`static`**. Such a data member is **associated with the class as a whole**, rather than each particular object of the class. A static data member is often referred to as a **"class variable"** and is typically used to store values which characterize the class in some way. For example a class variable may be used to keep track of the total number of objects that have been created.
- As with all class data members, the static one must be declared in the definition of the class. This brings the name of the class variable into scope, but **does not allocate memory** for the member. The allocation is done when the class variable is redeclared (and perhaps initialized) outside of the class definition. Because of this seemingly redundant declaration of the class variable, it is actually accessible before the first class object is created.
- A class variable can be referred to with two different syntaxes:
  - **With the class name** `class_name::static_data`
  - **With *any* object of the class** `class_object.static_data`

Static Data Members Example

```
#include <iostream.h>
class Pixel {
      int x,y;
   public:
      Pixel(int i, int j) { x=i; y=j; }
      static int xo,yo;
      int abs_x() { return(x+xo); }
      int abs_y() { return(y+yo); } };
int Pixel::xo;
int Pixel::yo;
main() {
   Pixel a(17,42);   Pixel b(55,75);
   Pixel::xo=5;      Pixel::yo=6;
   cout << "The absolute coordinates of pixel a are ("
        << a.abs_x() << "," << a.abs_y() << ")\n";
   cout << "The absolute coordinates of pixel b are ("
        << b.abs_x() << "," << b.abs_y() << ")\n";
   a.xo=0; a.yo=0;
   cout << "The absolute coordinates of pixel b are ("
        << b.abs_x() << "," << b.abs_y() << ")\n"; }
```

```
The absolute coordinates of pixel a are (22,48)
The absolute coordinates of pixel b are (60,81)
The absolute coordinates of pixel b are (55,75)
```

Inheritance

- When a new class is declared in C++, you can specify that it should be derived from an existing class.
- The new **derived class** inherits all data and function members from the **base class**, as if those class members had been declared explicitly within the new class declaration itself.
- The derived class modifies or extends the capabilities of the base class.
- It is through inheritance that **C++ enables and facilitates software reuse**. The reuse is even possible in cases where classes are made available to the user in the form of object libraries ***without*** accompanying member source code!

Derived Class Syntax

- The syntax for making a derived class is to insert a colon after the name of the derived class followed by an access method keyword and the name of the base class. For example, the following line declared the derived class **Land** from the base class **Vehicle** :

```
class Land: public Vehicle { class-member-list };
```

- The **Land** class inherits all members of **Vehicle** class except its constructors and destructor.
- A member function in the derived class can have the same name as a corresponding member function of the base class. When this occurs we say the derived class **overrides** (or redefines) the base member function. By default, the overridden function is used when a object of the derived class references it.

Any derived class can serve as a base class for other derived classes and thus a multi-layer class hierarchy can be constructed.

- The far-from-scientific program on the next page illustrates some of the basic features of derived classes.

Derived Class Example

```
#include <iostream.h>
class Golfer {
  public:
    int rounds_per_mo;  void swing() {cout << "#&%*@!\n";} };
class Serious_Golfer: public Golfer {
  public:
    int handicap;  void swing() {cout << "Ahhhh!\n";} };
class Pro_Golfer: public Serious_Golfer {
  public:
    float income;  void swing() {cout << "It's in the hole!\n";} };
main() {
   Golfer me;
   Serious_Golfer brother;
   Pro_Golfer john_cook;
   me.rounds_per_mo=1; me.swing();
   brother.rounds_per_mo=8; brother.handicap=15; brother.swing();
   john_cook.rounds_per_mo=20; john_cook.income=800000.00;
   john_cook.swing();
   john_cook.Golfer::swing(); }
```

```
#&%*@!
Ahhhh!
It's in the hole!
#&%*@!
```

Public Inheritance

- You may have noticed that with all the derived class declarations seen so far the access method public has been used before the name of the base class. Public inheritance is by far the most common form of class inheritance and has the following properties
- Public inheritance is an “is-a” relationship (A `Pro_Golfer` **is a** `Serious_Golfer` also, but more)
- The inherited ***public*** and ***protected*** members of the base class become public and protected members of the derived class, respectively
- Private members of the base class are inherited but **are accessible only to member functions of the base class**. If a member function of the derived class needs to directly work with a data member of the base class, make the base class data member protected.
- Because of the “is-a” nature of public inheritance, an object of the derived class may appear unambiguously in any context requiring a base class object
- Pointers or references of the base class type also can point or refer to derived class objects

Public Inheritance Example

```
#include <iostream.h>
class Vehicle {
      int weight;
   public:
      int getweight() { return(weight); }
      void setweight(int i) { weight=i; } };
class Land: public Vehicle {
      float speed;
   public:
      float getspeed() { return(speed); }
      void setspeed(float x) { speed=x; } };
main() {
   Land car;
// car.weight=2000;  Illegal! Weight is private
   car.setweight(2000); car.setspeed(110.5);
   Vehicle v;
   v=car;
   cout << v.getweight() << endl;
   Vehicle *vp;    vp=&car;
// vp->setspeed(65.4);  Illegal! *vp only has Vehicle capabilities
   vp->setweight(1000);   cout << car.getweight() << endl; }
```

```
2000
1000
```

Derived Class objects Example

- Since it can be said that an object of a derived class is at the same time an object of the base class, functions which use the base class will also work when derived-class objects are passed as arguments. The following code demonstrates this feature with our **Vehicle-Land** class hierarchy.

```
#include <iostream.h>
class Vehicle { int weight;
   public:
      int getweight() { return(weight); }
      void setweight(int i) { weight=i; } };
class Land: public Vehicle { float speed;
   public:
      float getspeed() { return(speed); }
      void setspeed(float x) { speed=x; } };
void check_wt(Vehicle& v) {
   if (v.getweight()>2000) cout << "Vehicle is too heavy\n"; }
main() {
   Vehicle blob; blob.setweight(100);
   check_wt(blob);
   Land car; car.setweight(5000);
   check_wt(car); }
```

```
Your vehicle is too heavy
```

Constructors and Destructors of Derived Classes

- When an object of derived class is declared, the constructor for the base class is invoked first, then the constructor for the derived class. It is the opposite order for destruction. When a derived-class object goes out of scope, the destructor for the derived class is invoked first, then the destructor for the base class.

- Actual arguments to a derived-class constructor are passed in the usual way. To specify actual arguments to the base class constructor, insert a colon after the derived class constructor's formal argument list, followed by the name of the base class and a parenthesized list of actual arguments for it.

- This technique is demonstrated in the code on the following page.

Constructors and Destructors of Derived Classes Example

```
#include <iostream.h>
class Widget {
      int wx;
   public:
      Widget(int a) { wx=a; }
      void display() { cout << "wx=" << wx << endl; }
};
class Gidget: public Widget {
      int gx;
   public:
      Gidget(int i, int j): Widget(i) { gx=j; }
      void display() { cout << "gx=" << gx << endl; }
};
main() {
   Gidget sally(5,33);
   sally.display();
   sally.Widget::display();
}
```

```
gx=33
wx=5
```

Mathematical Example: Averaging Data -- Data Class

- In this C++ code, the base class **`Data`** is responsible simply for inputting and outputting two arrays of data.

```
class Data {
   protected:
      double *x, *y;
      int N;
   public:
      Data(int i=1) {
         N=i;
         x=new double[N];
         y=new double[N]; }
      ~Data() { delete[]x; delete[]y; }
      void getdata();
      void showdata();
};
void Data::showdata() {
   for(int i=0; i<N; ++i)
      cout << "i="<<i<<" x="<<x[i]<<" y="<<y[i]<<endl; }
void Data::getdata() {
   for (int i=0; i<N; ++i)
      cin >> x[i] >> y[i]; }
```

Mathematical Example : Averaging Data -- Average Class

- The derived class -- **`Average`** -- extends and reuses the **`Data`** class. In **`Average`**, data is read in and averages of the data are also calculated.

```
class Average: public Data {
      double mux, muy;
   public:
      Average(int i): Data(i) {}
      void averagex() {
         double sum=0.0;
         for(int i=0; i<N; ++i)
            sum += x[i];
         mux=sum/N; }
      void averagey() {
         double sum=0.0;
         for(int i=0; i<N; ++i)
            sum += y[i];
         muy=sum/N; }
      void display () {
         cout << "Average x: " << mux << endl;
         cout << "Average y: " << muy << endl;
};
```

Mathematical Example : Averaging Data -- Main & Output

```
main() {
   Average temp(9);
   temp.getdata();
   temp.averagex();
   temp.averagey();
   temp.display();
}
```

```
Average x: 5
Average y: 51.8556
```

Virtual Function Terminology

- In more traditional programming languages, the act of invoking a function is static, which means that the compiler has sufficient information at compile-time to generate all the code necessary to implement the function call. This is called **early binding** of the function name to its code.

- A principal feature of C++ is that the **same function name** with the **same argument list** can perform different actions depending on the class of the object on which it is invoked. This property is referred to as **polymorphism**. In order to implement polymorphism, sometimes you must defer until run-time the association of a specific function name with the code that implements it. When this run-time connection occurs it is referred to as **late binding**. Finally, functions for which late binding is used are called **virtual functions**.

Virtual Functions: Syntax and Use

- To declare a member function to be virtual, precede the function declaration with the keyword `virtual`. Say that a base class has a virtual function and that function is overridden in a derived class. When the function name is invoked by an object of the derived class, the derived class version of the function is **always used**, **regardless of how the derived-class object is referred to**. In other words, when using virtual functions the meaning selected for the function depends on the ***class*** of the object rather than the way you ***refer*** to the object.

- On the next two pages we have resurrected the Golfer class hierarchy to demonstrate the difference between the actions of non-virtual and virtual overridden functions.

Virtual Functions Example: Golfer revisited

```
#include <iostream.h>
class Golfer {
  public:
    int rounds_per_mo;
    void swing() {cout << "#&%*@!\n";} };
class Serious_Golfer: public Golfer {
  public:
    int handicap;
    void swing() {cout << "Ahhhh!\n";} };
class Pro_Golfer: public Serious_Golfer {
  public:
    float income;
    void swing() {cout << "It's in the hole!\n";} };
main() {
   Pro_Golfer jc;
   Golfer *gp;
   gp=&jc;
   gp->swing();
}
```

```
#&%*@!
```

Virtual Functions Example: modified Golfer revisited

```
#include <iostream.h>
class Golfer {
  public:
    int rounds_per_mo;
    virtual void swing() {cout << "#&%*@!\n";} };
class Serious_Golfer: public Golfer {
  public:
    int handicap;
    void swing() {cout << "Ahhhh!\n";} };
class Pro_Golfer: public Serious_Golfer {
  public:
    float income;
    void swing() {cout << "It's in the hole!\n";} };
main() {
   Pro_Golfer jc;
   Golfer *gp;
   gp=&jc;
   gp->swing();
}
```

```
It's in the hole!
```

Pure Virtual Functions

- The virtual function **`Golfer::swing()`** shown on the previous page is an example of a **simple virtual function** because it is both **declared and defined** in the base class. A **pure virtual function**, on the other hand, is ***only declared*** in the base class. The (rather odd!) syntax for a pure virtual function is to append = 0 to the function declaration:

```
virtual return-type func-name(args) = 0;
```

- Pure virtual functions in base classes have the following characteristics:
  - **No objects of the base class can be declared**. Base class can only be used for the derivation of other classes.
  - **Must** be overridden in each derived class
  - A pure virtual function establishes a ***function interface*** inherited by all derived classes
  - A class that declares one or more pure virtual functions is termed an **abstract base class**

Pure Virtual Functions Example

- This program makes the **`Golfer::swing`** function a pure virtual function and thus makes **`Golfer`** an abstract base class.

```
#include <iostream.h>
class Golfer {
  public:
    int rounds_per_mo;     virtual void swing()=0; };
class Serious_Golfer: public Golfer {
  public:
    int handicap;    void swing() {cout << "Ahhhh!\n";} };
class Pro_Golfer: public Serious_Golfer {
  public:
    float income;
    void swing() {cout << "It's in the hole!\n";} };
main() {
   Pro_Golfer jc;
// Golfer mike;Illegal! Can't declare variable of an abstract base class
   Golfer *gp;
   gp=&jc;
   gp->swing();
}
```

```
It's in the hole!
```

C++ Templates and The Standard Template Library (STL)

Introduction to C++ Templates

- Say you wrote a function to sort an array of integers and you wanted to write a function that sorted an array of doubles as well. The algorithm you would be using would stay the same, but you would have to change all the type definitions in your new function.
- Function templates allow the C++ programmer to **write a single function to do the sorting and essentially pass the type of element to be sorted** to the function. Specifically, a function template contains a dummy argument corresponding to the type of data used in the function; the actual argument in the template function call determines what type of data is worked on. The programmer writes just one function template which will work for all types of data (**both standard types and derived types such as classes**). This general style of coding is called **generic programming**.
- Similarly, C++ programmers can create **class templates** which represent a family of classes determined by the type passed to the class when an object of the class is instantiated. Thus, in one situation the class data and member functions could be working on integers and in another on a more advanced data type. But again, only one class template definition is written.

Function Template Syntax

- A function template definition is similar to an ordinary function definition except that it is proceeded by a **template specification** header.
- A template specification consists of the keyword `template` followed by an angle-bracket-enclosed list of **template arguments**.
- Each template argument has the keyword `class` in front of an identifier. The identifier acts as a dummy argument for the type of data the actual template function will be working on when it is invoked.
- Here is an example of a function template definition:

```
template <class T>
void worker(T x, …) { function body }
```

Function Template Use

- To use the worker function just type its name as with normal C++ functions.
- Based on the actual type of the first argument **x**, the compiler will know what type **T** is for the particular invocation and generate the appropriate code. Due to this mechanism **C++ requires that each template argument appear as a type in the function argument list**.
- In addition, to insure that template functions work at all, implicit conversions are never performed on template arguments.
- There are some examples of using the template function worker:

```
int i;
worker(i,…);
Bookshelf library;
worker(library,…)
```

Type holder **T** is set to `int`

Type holder **T** is set to `class Bookshelf`

Demonstration Program

```
#include <iostream.h>
class A {
    int datum;
  public:
    A(int i) { datum=i; }
    void display() { cout << "(A)datum=" << datum << endl; } };
class B {
    double datum;
  public:
    B(double d) { datum=d; }
    void display() { cout << "(B)datum=" << datum << endl; } };
template <class T>
  void repeat(T tt, int n) {
    for (int i=1; i<=n; ++i) { tt.display(); } }
void main() {
   A anAobject(1);
   B aBobject(2.5);
   repeat (anAobject,2);
   repeat (aBobject,3);
}
```

```
(A)datum=1
(A)datum=1
(B)datum=2.5
(B)datum=2.5
(B)datum=2.5
```

Sample Function Template Program

```
#include <iostream.h>
template<class kind>
kind max(kind d1, kind d2) {
  if (d1 > d2)
    return(d1);
  return(d2);
}
void main() {
  cout << "The max of 3.5 and 8.7 is " << max(3.5,8.7) << endl;
  cout << "The max of 100 and 567 is " << max(100,567) << endl;
  cout << "The max of 'A' and 'a' is " << max('A','a') << endl;
}
```

```
The max of 3.5 and 8.7 is 8.7
The max of 100 and 567 is 567
The max of 'A' and 'a' is a
```

Templates and Related C++ Entities

Macros

- Function templates act as a sort of sophisticated macro expansion since the template function code is created at compile time using the "passed" type. Template Functions are preferred over macros for several reasons:
  - Macros prone to subtle syntactic context errors
  - Template functions allow for easier debugging and diagnostic information
  - Since template functions almost look like normal functions they are easier to understand

Overloaded Functions

- Like function templates, overloaded functions can be used to construct a family of related functions that are associated with their argument profiles. However, overloaded functions are typically used when **the behavior of the function differs depending on the function arguments**. On the other hand **template functions are essentially applying identical logic**, just with different types.

Function Template Specialization

- Function templates are designed to work correctly for any data type passed to it. What if this is not so? What if the programmer wants to do something special -- slightly modify the algorithm -- for certain special types?

- The user can create a **separate special function** to handle this case. This function can have the same name and argument profile as the function template. The new function will be **used instead of the function template**: its logic will be applied.

Function Template Specialization Example

```
#include <iostream.h>
#include <string.h>
template<class kind>
kind max(kind d1, kind d2) {
  if (d1 > d2)
    return(d1);
  return(d2);
}
char* max(char* d1,char* d2) {
  if (strcmp(d1,d2)>0)
    return(d1);
  return(d2);
}
void main() {
  cout << "The max of 3.5 and 8.7 is " << max(3.5,8.7) << endl;
  cout << "The max of 100 and 567 is " << max(100,567) << endl;
  cout << "The max of 'A' and 'a' is " << max('A','a') << endl;
  cout << "The max of apple and apply is " << max("apple","apply")
       << endl; }
```

```
The max of 3.5 and 8.7 is 8.7
The max of 100 and 567 is 567
The max of 'A' and 'a' is a
The max of apple and apply is apply
```

Class Templates

- A class template defines a **family of related classes which differ only in the type of the data stored and used by them**. A class template definition is preceded by the keyword `template` followed by an angle-bracket-enclosed list of class template arguments. These arguments consist of the keyword `class` followed by an identifier that will act as a dummy argument for the actual type the class will use. A class template argument can also be just a normal type name followed by an identifier. Here is an example:

```
template<class T, int N>
class Array {
   T array_element[N];
 public:
   T first_element() { return array_element[0]; }
};
```

- Each class in this template can be thought of as “an N-element array of Ts”.

Class Templates Use

- To create an object of template class, the class template arguments must be specified during the object's definition, as follows:

```
Array<int,10> wave;
```

- As with traditional classes, member functions for class templates must be declared in the body of the class template definition but may be defined either inside or outside the body. If defined outside the class template definition a special syntax must be used. The definition must be preceded by the class template specifier and the class template name and argument list (without type specifications). Here is an "outside" definition of the function `first_element`:

```
template<class T, int N>
T Array<T,N>::first_element() {
   return array_element[0];
}
```

Class Template Example

```
#include <iostream.h>
class Demo {
  public:
    Demo() { cout<<"Demo Constructor called"<<endl; } };
template<class T, int N>
class Tuple {
  public:
    T data[N];
    int getsize() { return(N); }
    T element(int i); };
template<class T, int N>
T Tuple<T,N>::element(int i) { return data[i]; }
void main() {
  Tuple<Demo,2> show; Tuple<int,3> point;
  point.data[0]=2; point.data[1]=78; point.data[2]=5;
  cout << "point is a "<<point.getsize()<<"-tuple\n";
  cout << "2nd element of point is "<<point.element(1)<< endl;
  Tuple<float,2> grid;
  grid.data[0]=3.4; grid.data[1]=45.6;
  cout << "grid has "<<grid.getsize()<<" elements\n";
  cout << "1st element of grid is "<<grid.element(0)<<endl; }
```

```
Demo Constructor called
Demo Constructor called
point is a 3-tuple
2nd element of point is 78
grid has 2 elements
1st element of grid is 3.4
```

Template Classes vs. Derived Classes

- It is natural to compare these two C++ constructs because both are used to represent families of related classes. To choose between these two options, there is a simple philosophy to follow:
  - Use **template classes** when classes in the family are **similar in behavior** but different in the type of data used
  - Use **inheritance** when classes in the family are **different in behavior** but similar in terms of some base class properties.
- Once instantiated, template classes act like ordinary classes in terms of inheritance. A template class can be a base class with no special syntax required.

Template Classes and Friend Functions

- A function declared to be a friend by a class template is a friend function of every template class that can be made from the template. That is, the friend function will work with all the template classes regardless of the type that got assigned to the class. As shown in the following sample program, friend functions for template classes are often template functions.

```
#include <iostream.h>
template<class T>
class A {
      T data;
      friend void show(A<T> ff);
   public:
      A(T x) { data=x; } };
template<class T>
void show(A<T> ff) { cout << ff.data << endl; }
void main() {
   A<int> ai(23);
   A<double> ad(45.678);
   show(ai);
   show(ad); }
```

```
23
45.678
```

Introduction to STL

- The Standard Template Library is a collection of extremely useful class templates and function templates allowing the programmer a wide variety of capabilities.
- There are literally hundreds of classes and functions the programmer can use with a variety of **simple and derived data types**.
- It provides many of the **data structures ("container classes")** and **algorithms** useful in computer science thereby preventing the programmer from "reinventing the wheel".

STL Features

- Here are some of the salient Features of STL:
  - It is an ANSI **standard and integral** part of C++ (beware competitors…)
  - STL is **very efficient**: container classes use no inheritance or virtual functions
  - STL algorithms are **stand-alone** functions which can work on virtually all types of simple data, normal classes, and STL containers
  - In order to accommodate machine-dependent, varying mechanisms for memory allocation and management, STL containers use special objects called ***allocators*** to control storage. This enables the portability required of an ANSI standard.

STL References

- Because of the sheer complexity of the STL, it is difficult to list and impossible to cover all the classes and functions within it.
- We will attempt to teach the library material most useful to research scientists and engineers.
- For a complete description of the library, see these recommended Web Sites

  http://www.sgi.com/Technoloy/STL/
  http://www.dinkumware.com/htm_cpl/

STL Quick Overview: Containers

- In the next few pages are small sample programs demonstrating the most useful STL libraries. This is meant to give the reader a taste of the detailed descriptions that will follow.
- **Containers**: Containers are classes designed to hold data objects. There are ten major container classes in STL and each is written as a class template. Thus, containers can hold data of virtually any type. Here is a program using the `vector` container template class.

```
#include <vector>
#include <iostream.h>
void main() {
   using namespace std;
   vector<int> v;
   v.push_back(42);
   v.push_back(1);
   cout << "vector size is " << v.size() << endl;
   cout << "v[0]="<<v[0] << endl;
}
```

```
vector size is 2
v[0]=42
```

STL Quick Overview: Iterators

- **Iterators**: STL makes heavy use of iterators which can be thought of as generalized pointers to the objects in a container. Iterators allow the programmer to move through the container and access data. There are several types of STL iterators which vary in the manner in which they scan container objects. The following program compares STL iterators and traditional C++ pointers:

```
#include <vector>
#include <iostream.h>
int array[]={1,42,3};
vector<int> v;
void main() {
   int* p1;
   for (p1=array;p1!=array+3;++p1)
      cout << "array has "<<*p1<<endl;
   v.push_back(1);
   v.push_back(42);
   v.push_back(3);
   vector<int>::iterator p2;
   for (p2=v.begin();p2!=v.end();++p2);
      cout << "vector has "<<*p2<< endl;
}
```

```
array has 1
array has 42
array has 3
vector has 1
vector has 42
vector has 3
```

STL Quick Overview: Algorithms

- **Algorithms**: The algorithms library contains common tasks programmers typically perform on containers of objects. Algorithms are stand-alone template functions which can operate on all the types of containers **and** regular C++ arrays. The program below illustrates the use of the `sort` STL function.

```
#include <vector>
#include <iostream.h>
#include <algorithm>
int array[]={1,42,3};
vector<int> v;
void main() {
   int* p1;
   sort(array,array+3);
   for (p1=array;p1!=array+3;++p1)
      cout << "array has "<<*p1<<endl;
   v.push_back(1);
   v.push_back(42);
   v.push_back(3);
   vector<int>::iterator p2;
   sort(v.begin(),v.end());
   for (p2=v.begin();p2!=v.end();++p2);
      cout << "vector has "<<*p2<< endl; }
```

```
array has 1
array has 3
array has 42
vector has 1
vector has 3
vector has 42
```

STL Quick Overview: Advanced I/O Stream

- **Advanced I/O Stream**: In addition to the basic capabilities presented earlier in terms of using the I/O stream method with stdin and stdout, there exists more sophisticated capabilities (i.e., string I/O, line I/O, file I/O, etc.) in the STL classes and functions. The sample program below shows a common file I/O application:

```
#include <string>
#include <fstream.h>
#include <list>
void main() {
  ifstream in("list.in");
  list<string> todo;
  while (in.good()) {
    string buf;
    getline(in,buf);
    todo.push_back(buf);
  }
  list<string>::iterator iter;
  for(iter=todo.begin();iter!=todo.end();++iter)
    cout << *iter << endl;
}
```

Output is exactly what is in the file `list.in`

```
Mow lawn
Teach C++
See movie
```

Containers: Basic types

- In this section, the various types of STL containers will be explained as well as the basic operations common to **all** containers. Here are ten STL containers with a brief description of their structure:

Container Type	*Brief Description*
`vector`	Linear, contiguous storage, fast inserts at end only
`deque`	Linear, non-contiguous storage, fast inserts at both ends
`list`	Doubly-linked list, fast inserts anywhere
`set`	Set of items, fast associative lookup
`multiset`	Like `set` but duplicate objects allowed
`map`	Collection of one-to-one mappings
`multimap`	Collection of one-to-many mappings
`stack`	First-in, last-out data structure
`queue`	First-in, first-out data structure
`priority_queue`	Maintains objects in sorted order

Containers: Functions

- Although containers have different properties, they all share a **set of common functions** which do basic bookkeeping and creation/comparison/destruction tasks. Here are functions related to basic container existence (using a vector container):

Function	*Example*	*Description*
Constructor	`vector()`	Construct the container
Destructor	`~vector()`	Destroy a container
Empty	`bool empty()`	Returns true if container is empty
Max_size	`size_type max_size()`	Returns maximum number of objects the container can hold
Size	`size_type size()`	Returns the number of objects in the container

Containers: Functions -- Example

- The following program illustrates the use of the common “container existence” functions with a vector container:

```
#include <vector>
#include <iostream.h>
void main() {
   vector<double> v;
   cout << "empty=" << v.empty() << endl;
   cout << "size=" << v.size() << endl;
   cout << "max_size=" << v.max_size() << endl;
   v.push_back(42);
   cout << "empty=" << v.empty() << endl;
   cout << "size=" << v.size() << endl;
}
```

```
empty=1
size=0
max_size=536870911
empty=0
size=1
```

Containers: More Functions

- The next two common functions allow assigning one container to another and swapping the objects of two containers

=

```
vector<T>& operator=(const vector<T>& x)
```

replaces calling object's vector with a copy of the vector **x**

swap

```
void swap(vector<T>& y)
```

swap the calling object's vector with the vector **y**

Containers: More Functions -- Example

- In the following program = and swap are used:

```
#include <vector>
#include <iostream.h>
void print(vector<double>& x) {
   for (int i=0; i<x.size(); ++i)
      cout << x[i] << " ";
   cout << endl; }
void main() {
   vector<double> v1; v1.push_back(12.1); v1.push_back(45.6);
   vector<double> v2; v2.push_back(2.893);
   cout << "v1="; print(v1);
   cout << "v2="; print(v2);
   v1.swap(v2);
   cout << "v1="; print(v1);
   cout << "v2="; print(v2);
   v2=v1;
   cout << "v2="; print(v2); }
```

```
v1=12.1 45.6
v2=2.893
v1=2.893
v2=12.1 45.6
v2=2.893
```

Containers: Even More Functions

- The last common function are related to comparing and copying containers:

Copy Constructor

```
vector<T>(const vector<T>& x)
```

construct a container object to be a copy of **x**

==

```
bool operator==(const vector<T>& y)
```

returns true if container object contains same items in same order as **y**

<

```
bool operator<(const vector<T>& z)
```

returns true if container object is “less than” **z** (***by lexicographical order***)

Containers: Even More Functions -- Example

TIP: If you include the STL library called utility, the other relational operators (!=,>, <=, >=) can also be used since they can be built from == and <.

- The following program shows the use of several of the relational operators:

```
#include <vector>
#include <iostream.h>
#include <utility>
void main() {
   vector<char> v;
   v.push_back('h');
   v.push_back('i');
   cout << "v=" << v[0] << v[1] << endl;
   vector<char> w(v);
   w[1]='o';
   cout << "w=" << w[0] << w[1] << endl;
   cout << "(v==w)=" << (v==w) << endl;
   cout << "(v<w)=" << (v<w) << endl;
   cout << "(v!=w)=" << (v!=w) << endl;
   cout << "(v>w)=" << (v>w) << endl; }
```

```
v=hi
w=ho
(v==w)=0
(v<w)=1
(v!=w)=1
(v>w)=0
```

Iterators

- We have already been introduced to the idea of an STL iterator as a "generalized pointer". Before we can explore more sophisticated containers than vector, the various types of iterators need to be discussed. Regardless of the type of iterator one thing is always true: **At any point in time, an iterator is positioned at exactly one place in a container until it is repositioned**.
- There are three major types of iterators:

  forward

  – can work with object only in the forward direction

  bidirectional

  – can move forwards and backwards

  random access

  – can jump an arbitrary distance

Iterator Use

- Iterators can be used with both input and output streams, as well as with containers (as we have seen). Basically the same set of basic arithmetic, logical, and de-referencing operations that can be performed with C++ pointers also work with STL iterators.

- Each STL container works with a certain type iterator. The **`vector`** and **`deque`** containers use **random access** iterators. The **`list`**, **`multiset`**, **`set`**, **`multimap`**, and **`map`** containers use **bidirectional**. In addition, each STL container has a set of typedefs that describe its iterators. For a container iterator there are the member functions begin and end to set the iterator to extreme positions in the container:

  `iterator begin()` -- returns an iterator positioned at the first object

  `iterator end()` -- returns an iterator **positioned immediately after the last object**

Reverse Iterators

- As you might expect, a reverse iterator travels a container of objects backwards instead of forwards. As with regular iterators there are container member functions used to place the iterator at extreme positions

  `reverse_iterator rbegin()` -- returns a reverse iterator positioned at the last object

  `reverse_iterator rend()` -- returns a reverse iterator **positioned immediately before the first object**

Reverse Iterator Example

- The following program uses a reverse iterator.

```
#include <string>
#include <iostream.h>
#include <list>
void main() {
   list<string> niece;
   niece.push_back("Madelyn");
   niece.push_back("Claire");
   niece.push_back("Ennis");
   list<string>::reverse_iterator r;
   for (r=niece.rbegin(); r!=niece.rend(); ++r)
      cout << *r << endl;
}
```

```
Ennis
Claire
Madelyn
```

Random Access Iterators

- Random access iterators can do it all: more forward, move backward, and jump from one position to another. Recall that the default iterator for a **vector** container is **random access** while you study the following program:

```
#include <vector>
#include <iostream.h>
void main() {
   vector<int> v;
   v.push_back(11);
   v.push_back(12);
   v.push_back(13);
   vector<int>::iterator i=v.end();
   cout << "last object is " << *--i << endl;
   i-=2;
   cout << "first object is " << *i << endl;
}
```

```
last object is 13
first object is 11
```

Algorithms

- The designers of the STL wished to implement **generic algorithms**: those that can be applied to a variety of data structures from normal integers and doubles to advanced classes. This design was realized through the use of function templates to define the algorithm functions in the library. There are **over 65 algorithms** in the algorithm library, the code for which actually makes up the bulk of the entire STL.

- To allow the STL algorithms to work on a variety of data structures from C++ arrays to STL containers themselves, the STL algorithms only access data indirectly by using iterators. In addition, some algorithms have several implementations optimized for the kind of iterator involved.

- In the following pages, sample programs using several of the more popular algorithms are presented. In a separate handout, a categorized listing of the STL algorithms and a brief description of their operation is presented.

Min/Max Algorithm

```
#include <algorithm>
#include <iostream.h>
void main() {
   double x=min(444.90,6.5);
   cout << "minimum of 444.90 and 6.5 is " << x << endl;
   char c=max('=','3');
   cout << "maximum (by ASCII value) of = and 3 is "
        << c << endl;
}
```

```
minimum of 444.90 and 6.5 is 6.5
maximum (by ASCII value) of = and 3 is =
```

Count Algorithm: (C++ array and vector container)

```
#include <algorithm>
#include <iostream.h>
#include <vector>
int num[]={1,4,2,8,9,12,2,13,8,2,2,1,4,2,33,1};
void main() {
   int n=0;   // must initialize, count increments n
   int asize=sizeof(num)/4;   // 4 bytes per element
   count(num,num+asize,2,n);
   cout << "There are " << n << " twos in the array\n";

   vector<int> v;
   v.push_back(1);v.push_back(4);v.push_back(2);v.push_back(8);
   v.push_back(9);v.push_back(12);v.push_back(2);v.push_back(13);
   v.push_back(8);v.push_back(2);v.push_back(2);v.push_back(1);
   v.push_back(4);v.push_back(2);v.push_back(33);v.push_back(1);
   n=0;
   count(v.begin(),v.end(),1,n);
   cout << "There are " << n << " ones in the vector\n";
}
```

```
There are 5 twos in the array
There are 3 ones in the vector
```

Accumulate Algorithm: Summing

```
#include <algorithm>  // may be <numeric> on some systems
#include <iostream.h>
void main() {
   const int N=8;
   int a[N]={4,12,3,6,10,7,8,5 };
   sum=accumulate(a,a+N,0);
   cout << "Sum of all elements:" << sum << endl;
   fun_sum=accumulate(a+2,a+5,1000);
   cout << "1000+a[2]+a[3]+a[4] = " << fun_sum << endl;
}
```

```
Sum of all elements:55
1000+a[2]+a[3]+a[4] = 1019
```

Accumulate Algorithm: General Binary Operation

```
#include <algorithm>
#include <iostream.h>
#include <vector>

int mult(int i, int j) {
   return (i*j);
}

void main() {
   vector<int> v(6);
   int prod;
   for (int i=0; i<v.size(); ++i)
      v[i]=i+1;
   prod=accumulate(v.begin(), v.end(), 1, mult);
   cout << "The factorial of 6 is " << prod << endl;
}
```

```
The factorial of 6 is 720
```

Inner Product Algorithm

```
#include <algorithm>
#include <iostream.h>
void main() {
   int a[3]={2,20,4};
   int b[3]={5,2,10};
   int inprod;
   inprod=inner_product(a,a+3,b,0);
   cout << "<a,b>=" << inprod << endl;
}
```

```
<a,b>=90
```

Inner Product Algorithm: General Operations

```
#include <algorithm>
#include <iostream.h>

int mult(int x, int y) {
   return(x*y);
}

int power(int x, int n) {
   int y=1;
   for (int k; k<n; ++k) y*=x;
   return(y);
}

void main() {
   int a[3]={2,3,5};
   int b[3]={4,1,2};
   int strange;
   strange=inner_product(a,a+3,b,1,mult,power);
//     power(2,4)*power(3,1)*power(5,2);
   cout << "This strange calculation equals "<<strange<<endl;
}
```

```
This strange calculation equals 1200
```

Find Algorithm

```
#include <algorithm>
#include <iostream.h>
#include <vector>
void main() {
   vector<int> v(5,8);
   for (int k=0; k<v.size(); ++k)
      cout << v[k] << " ";
   cout << endl;
   v[2]=33;
   vector<int>::iterator pos;
   pos=find(v.begin(), v.end(), 33);
   cout << "The value 33 was found at position "
        << pos-v.begin() << endl;
}
```

```
8 8 8 8 8
The value 33 was found at position 2
```

Merge Algorithm: Mixing Containers

```
#include <algorithm>
#include <iostream.h>
#include <vector>
#include <list>
void main() {
   vector<int> a(5);
   a[0]=2; a[1]=3; a[2]=8; a[3]=20; a[4]=25;
   int b[6]={7,9,23,28,30,33};
   list<int> c(11);
   merge(a.begin(), a.end(), b, b+6, c.begin());
   list<int>::iterator k;
   for (k=c.begin(); k!=c.end(); ++k)
      cout << *k << " ";
   cout << endl;
}
```

```
 2 3 7 8 9 20 23 25 28 30 33
```

Merge Algorithm: User-defined C++ Structures

```
#include <algorithm>
#include <iostream.h>
struct entry {
   int age;
   char name[30];
   bool operator<(const entry&b) {
      return(age < b.age); }
void main() {
   entry a[3]={ {10,"Andy"}, {45,"David"},{114,"Takii"} };
   entry b[2]={ {16,"Julie"}, {72,"Dorothy"} };
   entry c[5],*p;
   merge(a, a+3, b, b+2, c);
   for (p=c;p!=c+5; ++p)
      cout << p->age << " " << p->name << endl;
}
```

```
10 Andy
16 Julie
45 David
72 Dorothy
114 Takii
```

The Functional Library

- As was shown in some of the previous sample programs, one can customize (or radically modify) the behavior of certain algorithms by **using your own C++ functions and providing them as arguments in the algorithm call**. In the STL library **`functional`**, the designers of STL have provided the user with a set of built-in, useful functions that the programmer can use to do this same customization. STL actually allows these built-in functions to be encapsulated and treated as data objects and they are thus called **function objects**.

- Specifically, recall the sample program which used the accumulate algorithm to actually perform a factorial operation. In that previous program, the user had to define and use the function **`mult`**. On the next page, is the identical program but using the **`times`** function object that is a part of the **`functional`** library.

- In a separate handout, a table of all the function objects comprising the functional library is shown.

Accumulate Algorithm: `times` Function Object

```
#include <algorithm>
#include <iostream.h>
#include <vector>
#include <functional>
void main() {
   vector<int> v(6);
   int prod;
   for (int i=0; i<v.size(); ++i)
      v[i]=i+1;
   prod=accumulate(v.begin(), v.end(), 1, times<int>());
// for some compilers times has been replaced by multiplies
   cout << "The factorial of 6 is " << prod << endl;
}
```

```
The factorial of 6 is 720
```

Function Objects

- A function object may be created, stored, and destroyed just like any other kind of C++ objects. On the other hand, unlike an ordinary C function, a function object can have associated data. The key requirement of a function object is that **the () operator is defined** so that the use of the function object can look like a normal function reference. In addition, many of function objects in the STL **`functional`** library are written as templates in order to work with a variety of data types.

- For example, here is the structure template definition for the **`times`** function object we just used: (obtained from the **`functional`** include file)

```
template <class T>
struct times: binary_function<T,T,T> {
   T operator()(const T& x, const T& y) const {
      return x*y; }
};
```

Function Objects Example

- Here is a small program in which a “times” structure is created:

```
#include <iostream.h>
#include <functional>
void main() {
   time<double> prod; // create an instance of times object
   cout << prod.operator()(3.4,2.0) << endl;
   cout << prod(3.4,2.0) << endl;
}
```

```
6.8
6.8
```

Categories of STL Function Objects

- There are three kinds of function objects used in the STL:
  - **Predicates**: Boolean function objects used for setting the conditions for the action of algorithm to take place.
  - **Comparitors**: Boolean function objects used for ordering objects in a sequence (container or C++ array)
  - **General Functions**: Perform an arbitrary (but usually mathematical) operation on its arguments.
- The following pages contain sample programs in which each of these function object categories are used.

Predicate Sample Program

```
#include <iostream.h>
#include <functional>
#include <algorithm>
void main() {
   int tf[7]={1,0,0,1,1,0,1};
   int n=0;
   count_if(tf, tf+7, logical_not<int>(), n);
   cout << "The count of falses was " << n << endl;
}
```

```
The count of falses was 3
```

Comparitor Sample Program

```
#include <iostream.h>
#include <functional>
#include <algorithm>
#include <vector>
void main() {
   vector<int> v(3);
   v[0]=4;
   v[1]=1;
   v[2]=9;
   sort(v.begin(), v.end(), greater<int>());
   vector<int>::iterator k=v.begin;
   while (k != v.end())
      cout << *k++ << endl;
}
```

```
9
4
1
```

General Function Sample Program

```
#include <iostream.h>
#include <functional>
#include <algorithm>

void main() {
   int original[6]={2,3,-8,45,9,-3};
   int changed[6];
   transform(original,original+6,changed,negate<int>());
   for (int k=0; k<6; ++k)
      cout << changed[k] << " ";
   cout << endl;
}
```

```
-2 -3 8 -45 -9 3
```

Problem Set

1) Use accumulate to subtract instead of add. Output whatever is necessary to show that your program is working.

2) Write the code for a function called **`palindrome`** which takes a vector as its argument and returns a Boolean that is true only if the vector is equal to the reverse of itself. **Use iterators to perform the comparison**. In your **`main`** program, test your **`palindrome`** function with several vectors and output the results.

3) Use the **`remove`** algorithm to remove some objects from a vector. Does the size of the vector decrease? If not, modify your program so that it actually does remove the desired objects.

   TIP: Use the **`fill`** algorithm to fill up your vector to begin with.

Problem Set Continued

4) Write and use a function object **`choose()`** that takes two arguments and randomly returns one of them each time it is called. In your **`main`** program make enough uses of the **`choose`** object to verify that, statistically, the returns are random.

5) Write a program that uses the vector member functions **`pop_back`** and **`insert`** to erase parts of a vector and insert new objects in the middle.

6) Write a program that deals a set of hands for the card game Euchre.

HINT: You should use one of the containers we did not cover in detail in the class.