Physics 131/231: Simulations in C++

Prof. Clare Yu
email: cyu@moses.ps.uci.edu
phone: 949-824-6216
Office: 2168 FRH

Winter 2000

These notes draw material from the notes of Prof. Steve White, from Barton and Nackman's book *Scientific and Engineering C++*, and from *C++ for Dummies* by Stephen R. Davis.

LECTURE 1

Discuss Syllabus.

Project possibilities: Matrices of complex numbers, arbitrary precision arithmetic, data analysis, experiment control, finite elements, solve differential equations, plotting library (see web pages listed in syllabus for ideas and projects done by previous classes).

Misc: To obtain an account on Linux machines in computer lab (1002 FRH), see Tory Graziano in 480 Rowland Hall, pscsg@mail.ps.uci.edu, 824-6377. You can do the homework on any machine. If you use a Linux workstation, use the g++ compiler. Get key card to the computer lab from Jim Geier in Physics Department office. The machines are called frh1002x.ps.uci.edu where x=1,2,3,4,5. For example, frh10021.ps.uci.edu is the name of a machine. You can access them remotely using ssh; telnet will not work. Web pages:In your account, there will be a directory called WWW. Anything you put in this directory is accessible on the internet.

Reasons for learning C++

1. We can write better, bigger and more sophisticated programs.
   (a) Better encapsulation–"Object oriented approach"
   (b) Dramatic reduction in run–time bugs
   (c) Code reuse
2. Efficiency:
   (a) Memory efficiency is much better.
   (b) Speed: Can be as good as Fortran (and much better than other things)

3. Good for getting jobs outside of physics

Problems with C++

1. It's harder to learn.
2. A *fast* program takes effort.
3. C++'s most elegant features tend to be rather inefficient.
4. Not specifically designed for physics–need to make your own libraries

Basic Elements of C++ (Fortran Perspective)

Simplest program: hello.cc

```
#include <iostream.h>
int main()
{
  cout<<"hello world"<<endl;
return 0;
}
```

Or we could have written

```
#include <iostream.h>
void main()
        {
        cout<<"hello world"<<endl;
        }
```

Here `void main` means nothing is returned, even though something is done.

To compile this program:

```
g++ hello.cc
```

This gives the executable `a.out`. If you want the executable to be `hello`, then type

```
g++ hello.cc -o hello
```

To get more information on g++, type `man g++`.

Everything in C and C++ are functions. The main program `main` is called like a function.

Syntax: Brackets {} group lines. Each line ends with a semicolon ";". Usually there is no semicolon after brackets {} (except for a class). In fortran each line ends when you hit `<return>` and you continue lines with a character in the 6th column. In fortran

every new line must be indented by at least 7 columns. In C and C++, you don't have to indent lines.

`#include <iostream>` means read in the file iostream here. The `#` sign indicates that the statement is a preprocessor command. Conceptually, the preprocessor is the first step in compilation. We will discuss preprocessor commands as they arise. `< >` means look for the file in the standard places.

`cout` is defined in the file iostream.h as an iostream. It represents standard output and writes to the screen. `<<` is read as an arrow pointing out. `endl` means make a new line and flush the file from the buffer. `return 0` means standard return when everything is fine.

Input and output of numbers:

```
double a,b; //or could write float
cin >> a >> b;
cout << "a=" << a << ", b=" << b << endl;
```

`cin` is a way of reading numbers in that have been entered from the keyboard.

Things which are similar in Fortran, C, and C++

Types

Fortran	C, C++
integer	int (16 or 32 bits)
integer*8 (8 bytes) (1 byte=8 bits)	long or long int (32 or 64 bits)
integer*4	short or short int (16 or 32 bits)
integer*2	
character*1	char (8 bits)
real	float (32 bits)
real*8	double (64 bits)
real*16 (not on many machines)	long double (128 bits)
complex	defined in library: Complex
character*20	char a[20]; or use "string" library
logical	int (or 1 byte) (0 or 1)
Arrays	
Real A(10) has elements	float a[10]; has elements from
from A(1) to A(10)	a[0] to a[9]
Real A(10,10) is 2D array	float a[10][10]; // [row][col]

To initialize arrays in C:

```
int a[5]={0,2,18,12,5};

float b[2][4]={{23.0, 1.3, 22.2, 0.2},
               {0.24, 4.1, 12.0, 9.7}};
```

Comparisons

x.lt.y	x < y
x.le.y	x <= y
x.eq.y	x == y
x.gt.y	x > y
x.ge.y	x >= y
x.ne.y	x != y

Logical expressions

.false.	0
.true.	nonzero, 1
.not.x	!x
x.and.y	x && y
x.or.y	x || y (inclusive or)

Arithmetic operations: `+, -, *, /` mean the same in fortran, C and C++. But in fortran x^y is written as `x**y`, while in C and C++, `x**y` means nothing. One often uses `pow(x,y)` from the math.h library.

Conditional Statements

```
if(   ) then
......
endif
```

```
if(x<0)
  { .....
  }
```

Style statement: No {} are needed if there is only one statement.

```
if(x<0.0)
  x=0.0;
```

{} make a group of statements look like a single statement.

Else Statements

```
if(    )
          {...
          }
        else if(...)
```

```
    {
    }
  else
    {
    }
```

Nested `if` *statements*

```
  if(    )
    {
     if(    )
       {    }
     else {    }
    }
```

As an example, let's implement the `fabs()` function:

```
float fabs(float x)          //A function has () after its name.
  {
   float y;
   if(x<0.0)
     y = -x;
   else
     y = x;
   return y;
  }
```

Another way to write this is

```
float x,y;
y = (x<0.0)? -x : x;
return y;
```

In `y = expr1 ? expr2 : expr3`, expression `expr1` is evaluated first. If it is true (nonzero), `y` is set equal to `expr2`. If it is false (`expr1 = 0`), `y = expr3`.

Switch

The `switch` statement is a multi–way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly. The syntax is as follows:

```
switch (expression) {
 case const-expression: statements
 case const-expression: statements
 default: statements
 }
```

Each case is labeled by one or more integer–valued constants or constant expressions. If a case matches the expression value, the execution starts at that case. All case expressions must be different. The case labeled `default` is executed if none of the other cases are satisfied. A `default` is optional; if it isn't there and if none of the cases match, no action at all takes place. Cases and the default clause can occur in any order. For example

```
int i, j;
cin >> i;
switch(i) {
 case 1:
   j=i;
 case 2:          //if i=2 or 3, then j=i+5
 case 3:
   j=i+5;
 default:
   break;
}
```

The `break` statement causes an intermediate exit from the `switch`. After the code for one case is done, execution falls through to the next case unless you take explicit action to escape. `break` and `return` are the most common ways to leave a `switch`.

While loops

```
while(expression)
  statement
```

The *expression* is evaluated. If it is nonzero, *statement* is executed and *expression* is re–evaluated. This cycle continues until *expression* becomes zero, at which point execution resumes after *statement*.

Do–While loops

```
do
  statement
while (expression);
```

The *statement* is executed, then the expression is evaluated. If it is true, the *statement* is evaluated again, and so on. When the expression becomes false, the loop terminates.

Do Statement/ For loop

In fortran, the do loop has the form

```
do i=1,10
 .....
enddo
```

In C and C++, we would write

```
for(i=1;i<=10;i++)  // i++ is the same as i=i+1
.....
```

(Comment follows //. "//" is equivalent to "!" in Fortran or "/*...*/" in C.) The `for` statement is equivalent to

```
i=1;
while(i<=10)
 {
 ....
 i++;            // i++ means i=i+1
 }
```

Common Blocks vs. external variables

In fortran we denote a common block by

```
common /x/ a,b, ...
```

In C and C++, anything defined outside a function is global, unless declared otherwise.

```
int a;
void func1()
  {
   a=2;
  }
void func2()
  {
   cout << a << endl;
  }
```

Here these are all the same `a`.

A variable defined inside a function is local to the function, and no other function can have direct access to it. Each local variable in a function comes into existence when a function is called, and disappears when the function is exited. External variables are globally accessible by all functions, and they exist permanently, rather than appearing and disappearing as functions are called and exited. Suppose our program consists of several files with some functions in filea.cc, other functions in fileb.cc, and still other functions in filec.cc. How can we have variables that are global to the functions in filea.cc and fileb.cc but not in filec.cc? That's what the `extern` declaration is for. Suppose we want `aa` to be global to filea.cc and fileb.cc. Then we would define `aa` outside of any function in one file, say filea.cc, and then put an `extern` declaration in fileb.cc. One can either put the `extern` declaration at the top of fileb.cc or in each function that requires

it in fileb.cc. Note that a variable can only be *defined* once in a program but it can be *declared* more than once. A *definition* refers to the place where the variable is created or assigned storage. A *declaration* refers to places where the nature of the variable is stated but no storage is allocated. Our example would look something like this:

```
//*****************************filea.cc*****************************

  float aa;              //global definition of aa is outside of any function
  int func1(){

   aa=2;
   ...
   }

  float func2(){

   aa=aa*3.0;
   ...
   }
//*****************************fileb.cc*****************************

  extern float aa;     // declaration of aa that indicates the
                       // definition of aa is elsewhere.
  void func3(){
   cout<<aa<<endl;
   ....
   }

  void func4(){
   ...
   }
```

Or if we only want `func3` to know about `aa`, we could write

```
//*****************************fileb.cc*****************************

  void func3(){
  extern float aa;     // declaration of aa that indicates the
                       // definition of aa is elsewhere.

  }
  void func4(){
     float aa;         // here aa is local to func4() and is not the
```

```
                    // same as the global variable aa.
...
}
```

Built–in types that are not in Fortran:

Enumerations

```
enum Color {red, pink, blue};
Color c = blue;
```

Each color actually denotes an integer. By default, the count starts at 0.

```
enum Color {red=0, pink=1, blue=2}    //default
```

But you can specify some other numbering scheme:

```
enum Color {red=5, pink=7, blue}     //In this case, blue=8.
```

It is legal to go from `enum` to `int`. `int i = pink;` is legal. But it is not legal to go from `int` to `enum`. `Color c = blue + 1;` is not legal because there are more integers than Colors.

(By the way, C and C++ are sensitive to upper and lower case, while Fortran is not case sensitive.)

New arithmetic operations in C and C++:

There is a nice shorthand involving `+=, -=, *=, /=`. For example, `a=a+b` becomes `a += b` and `a=a*c` becomes `a *= c`.

Increment:

`i++` is the same as `i=i+1`

`i--` is the same as `i=i-1`.

`++` and `--` may be used either as prefix operators (before the variable, as in `++n`), or postfix operators (after the variable: `n++`). In both cases, the effect is to increment `n` by 1. But the expression `++n` increments `n` *before* its value is used, while `n++` increments `n` *after* its value has been used. For example,

```
n=5;
x=n++;
```

sets `x` to 5 and `n` to 6. But

```
n=5;
x=++n;
```

sets both `x` and `n` to 6. Increment and decrement operators can only be applied to variables; and an expression like `(i+j)++` is illegal.

Assignments using `=` signs return values, namely a new left side: `a=(i=4)+2` sets `a` to 6.

Modulo `i % j` = modulus = remainder. For example, `5 % 3 = 2`, `20 % 7 = 6`, etc.

Bit Operators

```
  ~i              bit by bit "not", switches 0 to 1 and 1 to 0
  i & j           "AND"
  i ^ j           exclusive "OR"  1^1 yields 0
  i | j           inclusive "OR"  1 | 1 yields 1

 operator        i            j          i operator j
-----------------------------------------------------

   ~           001                       110
   &           011          110          010 (Both have to be 1 to get 1)
   ^           011          110          101
   |           011          110          111
```

<< left shifts bits. For example, `1 << 2 = 4` since 4 is written `100` in binary. The `2` tells how far to shift the 1 bit.

>> right shifts bits. For example, `4 >> 2 = 1`.

Octal constants are preceded by `0` (zero): `030 = 24` since `3 x 8 = 24`. Hexadecimal constants are a sequence of digits preceded by `0x`. For example, `0x10 = 16`.

Unsigned type such as `unsigned int` means that all the bits go into numbers, and none of the bits is used to designate a sign. This gives more bits to be used for numbers.

Characters and Strings

A single quoted letter is a constant for one character. `'a'` means one character, the letter `a`.

```
char c = 'X';         //defines c and initializes it.
c='y';                //reassigns c.
```

Special characters:

```
'\n' = newline
'\t' = tab
 etc.
```

Literally a char is a byte–size integer. `'a'` means "ascii code for `a` as a byte–sized integer."

```
cout << 'a' << (int)('a') << endl;
```

yields `a97` because 97 is the code for `a`. Here's how you can print out the ascii code for the alphabet:

```
for(char c = 'A'; c <= 'z'; c ++)
  cout << c << "  " << (int)c << endl;
```

(Try it!).

Cast In the above example `(int)c` is an example of using the *cast* operator. Even though `c` is of type `char`, writing `(int)c` forces a type conversion into an `int`. `c` isn't converted into an `int`, but the expression `(int)c` delivers the integer value of `c`. In general, a cast construction has the form

```
(type-name) expression
```

Here the value of `expression` is converted to the named type, even though `expression` itself is not actually altered.

A **string** is an array of chars.

```
char s[20];
```

A string constant looks like this:

```
"A string"            A   s t r i n g \0
                      - - - - - - - - -
                      0   2   4   6   8
```

The array elements occupy slots 0 through 8. The array size usually is larger than the string length. The last array element is a `\0`.

```
char s[]="A string"; //sets the length automatically
char s[20]="A string"; //sets s to  A   s t r i n g \0 \0 \0 \0 ...
```

To copy strings, treat them like arrays. So don't say

```
char s[20];
char t[20]="A string";
s=t;
```

(This makes `s` and `t` point to the same point in memory. We want to make 2 copies of the same array that will occupy 2 different places in memory.) Rather say

```
char s[20];
char t[20]="A string";
for(i=0;i<20;i++)
  s[i]=t[i];
```

If we don't want to waste time copying all the `\0's`, we can write

```
char s[20];
char t[20]="A string";
i=0;
while (1)
```

```
{
 s[i]=t[i];
 i++;
 if(t[i]=='\0')
   break;
}
```

or

```
char s[20],t[20];
i=0;
while((s[i]=t[i]) != '\0')
  i++;
```

"`break;`" tells the program to exit the innermost loop that it is inside of. If the program doesn't break, then it continues with the next iteration.

Typedef

The `typedef` declaration gives an additional name to an existing type. For example,

```
typedef float distance;
```

Now distance is a synonym for `float`. We can write

```
distance d;
```

This can make the variables in a program easier to understand. Another use for `typedef` is if you want to change the variable types in a program. You may want to do this to suit machine–dependent data types, since different machines can have different numbers of bytes for `long`, `short`, and `int`. So rather than change every line, you declare your variables with a synonym and then just change the `typedef` statement. For example, let's say you write

```
typedef long number;
number a;
number b;
number c;
```

Later, if you want to change all the `number` variables to be `short`, you just change the `typedef` statement:

```
typedef short number;
number a;
number b;
number c;
```

This is much easier to change than if you had originally written

```
long a;
long b;
long c;
```

and then had to change each line to

```
short a;
short b;
short c;
```