

# C++ Seminar

Joe Boudreau  
June 15-19 1998

What is C++?

- \* C++ is a programming language supporting the Object-Oriented Paradigm.
- \* C++ grew out of the C programming language, a simple and popular programming language developed at AT&T in 1970.
- \* C++ is a superset of the C language ==> C++ compilers will compile C code, but not the other way around.
- \* Many of the features of C++ are incremental improvements to the C programming language.
- \* However the most important features require a completely new way of thinking about programming (Object-oriented programming).

What is the big difference, then?

**Procedural** programs start by defining **data** and **functions** that manipulate these data. The data is passive and the functions act on the data. This is the paradigm of languages such as FORTRAN, C, PASCAL, COBOL.

**Object-oriented** programs consist of autonomous packages of data + functions called **objects**. Programs are built out of objects, which exchange messages through specific, programmer-defined protocols. This is the paradigm of languages such as JAVA, Smalltalk, Eiffel, and C++.

C++ supports both procedural and Object-oriented programming, but I will try to teach Object-oriented programming.

## Why C++?

-Many people feel that the Object-oriented paradigm is better.

- More reusable code
- Excellent match to large-scale projects.
- Imposes more engineered & designed code.

-But also:

- Harder to learn.
- Easier to screw up.
- Requires more discipline.

-And

- Built-in data types are extensible.
- Programmer extensions are also extensible

-C++ is a powerful language because it supports the OO paradigm, by providing the following features:

- Encapsulation
- Inheritance
- Polymorphism
- Templates

-C++ has a powerful run-time library:

- String classes.
- Container classes.
- I/O classes.

=> So, let's start learning about C++.

## How to create and run C++ programs:

---

\* Create the text of the program. For this I suggest the editor "emacs" on unix platforms.

```
emacs helloWorld.cc
```

\* Your first C++ program will look like this:

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

\* The program must then be **compiled** and **linked**. This can be done in one step by typing (from the unix command-line):

```
CC    helloWorld.cc    -o helloWorld
```

\* An alternate way to build the program is to type

```
make helloWorld
```

The unix utility "make" knows how to build programs from files with standard extensions and applies the set of rules it knows to build programs. This is very powerful, but we will not discuss it further.

\* Now, to run your program, you may type:

```
helloWorld
```

\* Now, try doing this from your account.

We will necessarily have to use certain constructions before we explain them fully, especially for I/O.

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

header file  
inclusion

```
int main() {
```

main program

```
cout << "Hello world!" << endl;
```

next-line

```
return 0;
```

character string

```
}
```

output class variable

shift operator

\* In order to see if your test programs work, you need to get them to print something.

\* The `cout` output class variable is used for printing text to the screen.

\* The `cout` output class variable can also be used to print the value of `constants` and `variables` to the screen.

\* The `shift operator` "sends things" to `cout` (note that it suggests a direction of "flow").

\* The `endl` object means "end line", (`endl` is called an input/output manipulator).

## Data types.

---

C++ has built-in data types.

Integer types:

```
int
long int
short int
char (or very short int)
```

```
unsigned int
unsigned long int
unsigned short int
unsigned char
```

Floating point (e.g real number) types:

```
float (don't use this)
double
```

A data type for true or false:

```
bool
```

A string data type (strictly speaking, not part of the language):

```
string
```

(To get this, you need to include a header file, on the SGI,)

```
#include <mstring.h>
```

C++ code consists of comments, declarations and executable expressions

Comments:

```
// A comment like this is not processed by the
// compiler. It also doesn't slow down the
// code..So use plenty of 'em! Not even authors
// can read a piece of code without comments.

/* Here is another kind of comment, which is
   the C-style of writing comments. It's still
   understood in C++, but the "double slash" form
   is preferred! */
```

Declarations:

```
int nDimensions;    // number of dimensions
double x;           // x-value of vector.
```

\* all variables must be declared

\* a declaration consists of a type, a name, (and a comment).

\* the following syntax is also allowed:

```
int i, j;
```

\* variables typically "come up" uninitialized.. that is, their value is unpredictable. You can explicitly initialize them in C++;

```
int i=0, j=0;
int k(4);
```

Executable expressions:

```
i = j+2;
j = j+2;
x = 2*y;
```

## Notes on C++ syntax:

\* variable names may consist of any combination of letters, numbers, and underscores (`_`) but may not:

- be identical to certain C++ reserved words (for, which, int..)
- contain spaces.
- start with a number.
- \_contain nonalphanumeric characters (except underscore)

\* C++ is case sensitive: (nPhotons is not the same variable as NPHOTONS).

\* all declarations and statements must end with a semicolon.

\* all declarations and statement may continue onto other lines.

\* more than one declaration or statement may appear on any line, provided that they are separated by a semicolon.

\* your emacs editor will beep, indent, or otherwise bother you if it detects a syntax error in C++ code.

\* Declarations and executable statements may be mixed, unlike in many other languages:

```
int i;  
i = 4;  
int j;  
j=2*i;
```

\* all variables must be used before they are used.

\*variables are not to be re-declared (exceptions to this rule will be discussed later but are not recommended anyway).

\*a comment may follow any declaration or executable statement:

\*For now we will put all variable declarations within the body of "main", later; within the body of main or subroutines.

## Notes on C++ variable naming conventions

The following are not enforced by the compiler, but will result in more conventional and readable code:

- \* local and global variable names lower case or mixed case:

```
myCoordinateAxis  
axis  
colour  
quarkColor
```

- \* constants in upper case:

```
M_PI  
C_LIGHT  
RAND_MAX
```

- \* class variables start with an underscore:

```
_xValue;  
_yValue;  
_zValue;
```

- \* function names are lower or mixed case

- \* private member functions start with an underscore.

Operators:

\* Executable statements involve variables & operators:

\* Here is a list of the arithmetic operators and their function:

Operator	Meaning	E.G
*	multiply	
/	divide	
+	add	
-	subtract	
%	modulus	4%3 (returns 3)

\* The assignment operator is denoted by "=".

\* The assignment operator returns a value,

```
x = y = z = 0;
```

```
x = 4 % (z=2+1);
```

\* There is a difference between assignment and initialization. The symbol "=" however can appear in both of them.

```
double hBar=6.582E-20; //(MeV s) (initialization)
double kineticEnergy;
kineticEnergy = 1.1; //(GeV) (assignment)
```

=> initialization initializes the state of a variable.

=> assignment changes the state of a variable.

\* \*, /, and % have precedence over + and -.

## \* Constants.

A number without a decimal point is an integer constant, eg.

4040  
-10  
0

Integers constants may be given "base-8" (octal) by providing a leading zero:

019 (equal to 89 in normal base-10)

=> be careful with leading zeros!

Integers may (more usefully) be given "base-16" (hexadecimal) by providing a leading 0x

0x10 (equal to 160)

A number which includes a decimal point is a double precision constant, e.g.

10.3  
4040.0  
0.003  
0.0

Double precision constants may also be written in scientific notation:

4.04E3  
6.582E-20

char's may be given either as integers of a limited range or as keyboard characters between single quotes:

'c'  
'd'  
'F'  
'@'

"Special" characters also exist:

'\b'	backspace	move the cursor to the left one character
'\f'	formfeed	go to the top of a new page
'\n'	newline	go to the next line (like endl);
'\r'	return	go to the beginning of the current line
'\t'	tab	go to the next tab stop
'\''	apostrophe	'
'\"'	quote	"
'\nnn'	char number	the ascii character # nnn in octal
'\NN'	char number	the ascii character # NN in hexadecimal

Finally, string constants, which are arrays of characters in a contiguous block of memory.

"Hello, world"  
"Hello, world\n"  
"Hello \t world"

Single characters go between single quotes.

Arithmetic operations on numeric types:

Mostly self-explanatory, however the operator / is different because it gives different results on integer types as on floating point types.

Integer arithmetic...

$4/5 \Rightarrow 0$

$5/4 \Rightarrow 1$

Integer division gives an integer result; that result is obtained by throwing away the remainder

floating point arithmetic gives a floating point result:

$4.0/5.0 \Rightarrow 0.8$

$5.0/4.0 \Rightarrow 1.25$

Then what is  $4/5.0$ ???

The compiler **promotes** the integer to a floating point number.

$4 \Rightarrow 4.0$

The result is computed as 0.8. The compiler will promote variables--either built-in or programmer-defined--whenever it can.

Floating point numbers may be converted to integers, too, this conversion is done by truncation (throwing away the fractional part)

```
int i;  
i= 1.2; // i gets a value of 1
```

\*Integer types:

int  
long int  
short int

Are integer data types. They each have their own size, with usually

long int 32 bits (4 bytes)  
int 32 bits (4 bytes)  
short int 16 bits (2 bytes)

\* one bit is used for the sign (+/-), so the int data type can store integers with a value

$-2,147,483,647 \leq \text{value} \leq 2,147,483,647$

\* unsigned integers can only represent positive integer quantities, but the positive range is doubled.

$0 \leq \text{value} \leq 4,294,967,295$

they are written as

unsigned long int  
unsigned int  
unsigned short int

float (32 bits)  
double (64 bits)

Are floating point data types.

Shorter data types save memory. They used to save execution time. With modern compilers & processors, this is no longer clear. I recommend using int and double, unless you have good reason to do otherwise.

More on output:

-----

\* Use the cout class variable to write out variables as well as strings, eg,

```
int hour=11;
double temperature = 76.2;
cout << "The temperature was " << temperature
     << " at hour " << hour << endl;
```

produces this output:

The temperature was 76.2 at hour 11

\* There is a class variable called cin that reads from the terminal.

\* We defer detailed discussion till later.

\* A few examples will get you by:

```
#include <iostream.h>
int main() {
    int tempFahrenheit;
    cin >> tempFahrenheit;
    int tempCelsius = (tempFahrenheit-32)*(5.0/9.0);
    cout << tempFahrenheit << " Fahrenheit= "
         << tempCelsius
         << " Celsius" << endl;
    return 0;
}
```

\*This program does the following. User types:

90

Program echos:

90 Fahrenheit= 32 Celsius

\* Several variables can be read in at a time:

```
cin >> variable 1 >> variable 2;
```

\* Input should be two numbers separated by white space, e.g.

```
40 20
```

(on same line) or

```
40  
20
```

(on different lines).

\* We will shortly show a way to test whether the input has reached the end or failed.

\* For C language `scanf` and `printf` continue to work, but you shouldn't use them (c.f. Scott Meyers, *Effective C++*)

## Exercices.

1) Write a program that:

- reads two integers from the terminal
- calculates the number of times the second integer divides the first
- calculates the remainder
- prints the results in the format:

$n$  divided by  $m = p \text{ } q/m,$

e.g,

12 divided by 7 is 1 5/7.

2) What do you think the following program will print out?  
When you think you know, try it and see.

```
#include <iostream.h>
int main() {
    int integer;
    double floating;

    cout << (floating = (1.0/2.0)) << endl;
    cout << (integer = 1/3) << endl;
    cout << (floating = (1/2) + (1/2)) << endl;
    cout << (floating = 3.0 / 2.0) << endl;
    cout << (integer = floating) << endl;
    cout << (floating = integer) << endl;
    return 0;
}
```

Shortcuts:

\*The construction

```
i = i + 1;  
i = i + 2;
```

is often used. There is a more compact way to write this that is also allowed:

```
i++ or ++i;  
i+=2
```

(Note `i++` and `++i` both increment `i` by one. The only difference comes in statements like `j = ++i;` or `j=i++;` In the first case, `i` is incremented before it is assigned; in the second case `i` is assigned before it is incremented. I recommend **never** incrementing variables in conjunction with other operations because it is confusing.)

The following are also allowed:

```
--  
*=  
/=
```

and

```
--
```

E.G.

```
int j= 6;  
j /= 3;    // j will have a value of 2.
```

## Decision and control statements:

- \* **Branching statements** control whether or not to execute some piece of code.
- \* **Looping statements** control the iteration of some piece of code.
- \* Conditionals are used in both.
- \* Conditionals are statements that evaluate to true or false.
- \* Simple examples:

```
if (0) cout << "This will never be executed" << endl;  
if (1) cout << "This will always be executed" << endl;  
if (j<4) cout << "This will sometimes be executed" << endl;
```

- \* Conditionals are usually made up of relational operators. Relational operators are:

<=	less than or equal to
<	less than
>	greater than
>=	greater than or equal to
==	equal to.
!=	not equal to.

- \* They work on both floating point and integers. However, be careful testing equality of floating point numbers, because of precision.

- \* Be careful not to confuse = and ==.

= assignment.

== equality relation.

if (a=1) is legal, but almost certainly wrong! (some compilers will warn, others might not!).

The syntax of C++ supports boolean operations on all conditional expressions:

Logical Operators:

---

|| OR  
&& AND  
! NOT

( Warning: The operators

| Bitwise OR

& Bitwise AND

also exist in C++, but mean something quite different! They take the bit pattern used to represent the integer and perform the AND or OR operation Bit-by-bit. We won't discuss these further)

E.G., you can write:

```
if ((j>10) && (j<15)) cout << j << endl;  
if ((j>0) && !((j>10) && (j<15))) cout << j  
                                     << endl;
```

The syntax of the if statement:

-----  
if (condition) statement;

E.G.

```
if (j>0) cout << j << endl;
```

-----  
if (condition)  
 statement;

E.G.

```
if (j>0)  
    cout << j << endl;
```

*no semicolon!*



-----  
if (condition) {  
 statement1;  
 statement2;  
 statement3;  
}

*This is known  
as a "block"*



E.G.

```
if (j<0) {  
    cout << "The value of j was: ";  
    cout << j;  
    cout << endl;  
}
```

And in full generality:

```
if (condition1) {  
    block 1  
}  
else if (condition2) {  
    block 2  
}  
else if (condition3) {  
    block 3  
}  
else {  
    block 4  
}
```

The blocks may mix variable declarations and executable statements.

- \* Since C++ is not a line oriented language, it is not too sensitive to how the blocks are indented, what lines the braces go on.
- \* It is extremely sensitive to incomplete parenthesis.
- \* A consistent style of indentation makes your code readable.
- \* There are several styles in common use, including the one above.
- \* The emacs editor does automatic indentation for you when you are editing a C++ program.
- \* Parenthesis highlighting (in the Options/Parenthesis menu) can be useful in detecting parenthesis errors.

Loop statements:

```
while (condition) {  
    block.  
}
```

Example: Compute the sum of numbers from zero to 100:

```
#include <iostream.h>  
main () {  
    int i(0),result(0);  
    while (i<=100) {  
        result +=i;  
        i++;  
    }  
    cout << result << endl;  
}
```

Example: compute the sum of all numbers typed by a user:

```
#include <iostream.h>  
main () {  
    int i(0),result(0);  
    while (cin >> i) {  
        result +=i;  
    }  
    cout << result << endl;  
}
```

means "while the read was successful"...just remember this even though the syntax is strange.

\* The keywords "break" and "continue" are used within looping statements like "while"

\* Break means exit the loop.

\* Continue means proceed to the next iteration of the loop.

For example:

```
#include <iostream.h>
main () {
    int i(0),result(0);
    while (1) {
        result +=i;
        i++;
        if (i==101) break;
    }
    cout << result << endl;
}
```

Gives same result as our previous example.

The most commonly used looping statement is the for statement. Our program to sum integers from 0 to 100 could be written like this:

```
#include <iostream.h>
main () {
    int result(0);
    for (int i=0;i<101;i++) {
        result +=i;
    }
    cout << result << endl;
}
```

executed once before loop

tested at start of iteration.  
if condition fails, loop exits.

performed at end of iteration, before test.

## Syntax of the "for" statement:

```
for ( initial statement ; condition ; iteration-statement) {  
    statement1;  
    statement2;  
    statement3;  
}
```

The for statement is most often used to perform operations a preset number of times, but that is not necessary.

```
//-----  
// A short blackjack program.  Enter cards one at  
// a time.  The program keeps track of your total.  
// Program exits when you get Blackjack (total of 21)  
// or when you go above 21.  
//-----  
#include <iostream.h>  
main () {  
    int i(0);  
    cout << "First card is:";  
    for (int result=0;(cin >> i) ; cout << "Next card is:" ) {  
        result += i;  
        if (result==21) {  
            cout << "Blackjack" <<endl;  
            break;  
        }  
        else if (result > 21) {  
            cout << "You are dead" << endl;  
            break;  
        }  
        else {  
            cout << "Your total is " << result <<endl;  
        }  
    }  
    return 0;  
}
```

## Running the program, you get:

```
First card is:13  
Your total is 13  
Next card is:7  
Your total is 20  
Next card is:1  
Blackjack
```

Some other branching and looping statements are useful. You may look into these statements at home or when you have time.

**switch statements** (for switching between multiple blocks of code, similar to a chain of if-else statements).

**do statements.** (similar to for loops, except the conditional is evaluated at the end of the loop).

Exercises:

1) Make a program to compute the value of  $N!$  ( $N$  factorial) for any value of  $N$ . Design your program so that you can input  $N$  from the terminal.

2) Make a program to compute the value of  $2^N$ , and use it to determine the length of an int, a long int, and a short int on an SGI machine like cactus. (Hint: if you compute  $2^N$  on a hand calculator, you overflow the calculator for some  $N!$  Which value of  $N$  causes your program to overflow the range of an integer? How can you tell?

## Constants.

\* In a C++ program data can be either **variable** or **constant**. A constant cannot ever change its value.

\* A constant is declared much like a variable, except with the keyword `const`, e.g.

```
const int NMAX=100;  
const unsigned int NGENERATIONS=3;  
const double HBAR=6.582E-20;
```

\* constants should always be initialized because they are impossible to change.

\* Note: The qualifier `const` is also used in many other ways to mean many other things.

\* Using upper case letters in the names of constants is customary, but not required.

## Pointers and arrays.

\*All variables "live" in some piece of memory. The location of that piece of memory is called the address of the variable. C++ (like C) has the notion of a **pointer** which is a something that holds the address of a variable, as well as the type of the variable:

```
int i;           // is an integer variable
int *iPtr       // is a integer pointer.
```

\* In C++ you can take the address of a variable using the operator &:

```
iPtr = & i;    // assigns iPtr to the address of i.
```

\*Note that iPtr must be a pointer-to-integer to contain the address of an integer.

\*Then, one can fetch the value of the variable using the \* operator.

```
*iPtr = 4;     // iPtr is an integer-pointer.
               // *iPtr is an integer.
               // that integer is assigned the value "4".
```

Brain teaser: what does the following program print:

```
#include <iostream.h>
int main() {
    int i=0;
    int *iPtr=&i;
    *iPtr = 4;
    cout << i << endl;
}
```

=> Variables can be "handled" directly or through their pointers.

\*C++ (unlike C) also has a thing called "references", which are just aliases for other variables, e.g. another name for the same variable:

```
int i;  
int & another_name_for_i = i;  
  
i = 5; // another_name_for_i changes, too.  
another_name_for_i = 5; // i changes, too.
```

\*Why one would want such a thing will become clear pretty soon.

\*Arrays are sets of objects that occupy contiguous memory locations: They are referenced by index. e.g.

Declaration:

```
int socialSecurityNumber[10];
```

Access the first social security number in the array:

```
int firstSSN = socialSecurityNumber[0];  
int secondSSN = socialSecurityNumber[1];
```

\*Note: Arrays use zero-based counting (0,1,2...) rather than (1,2,3..)

\*Multidimensional arrays also exist in C++. I won't discuss them. For people with prior C experience...they are very much the same.

You can declare arrays like this:

---

Declare an array. Elements will be zeroed (?).

```
int ssN[10];
```

---

Declare and initialize an array:

```
int ssN[10]={119562830, 2391738713};
```

Other elements are left zero. Extra values (if any) generate compile time errors.

---

Another way of declaring an array:

```
int ssN[]={119562830, 2391738713};
```

Makes an array with only two elements and initializes them as specified.

---

You can set the size of the array with a constant integer:

```
const int N=5;  
int ssN[N];
```

---

You can make arrays out of any kind of variable, even the ones that you define yourself.

Pointer arithmetic:

Suppose I have an array

```
int intArray[] = {1, 4, 88, 53, 2, 7, 8};
```

Consider:

```
intArray
```

and

```
&intArray[0]
```

In C++ (as in C) these are the same thing! They mean "the place where the array begins", specifically, the address of the first object.

The type of both is `int *`

It is legal therefore to write

```
int *anotherPointer = intArray;
```

`anotherPointer+1` means "the next integer in the array"

it is equivalent to `&intArray[1]`;

I can also write `anotherPointer[4]`. What value does it have?

\* Summary: pointers can be treated like arrays, and vice versa.

\* C Programmers will know about this.

\* Everybody else will think it's **really weird!**

Here is a program that prints out values of an array.

```
#include <iostream.h>
main() {
    double sequence[]={32, 16, 8, 4, 2, 1, .5, .25, .125, .0625};
    double *handle = sequence;
    for (int i=0;i<10;i++) {
        cout << handle[i] << endl;
    }
}
```

The C++ compiler treats this program in exactly the same way.  
Why?

```
#include <iostream.h>
main() {
    double sequence[]={32, 16, 8, 4, 2, 1, .5, .25, .125, .0625};
    double *handle = sequence;
    for (int i=0;i<10;i++) {
        cout << *(handle+i) << endl;
    }
}
```

The following would be illegal:

```
int handle * = sequence;
```

Pointers are very similar to references. Both allow programmers to read or write one variable **through** another. Pointers are more natural when dealing with arrays.

Both pointers and references can be made **read-only** through the use of the qualifier **const**:

```
double x=10.5;
const double *handle = sequence;
x = handle[i] ; // is allowed
handle[i]=x ; // is forbidden, because handle is const.
```

## Functions:

A block of code that is frequently executed can be put into a function, (C++ equivalent of a subroutine).

\*A functions has a name, parameters. and return value.

\*Here's an example. The following program has become too complicated to put in a single main function:

```
// Program to calculate eulers E. Calculates
// sqrt(e) through the series expansion
// 1 + x + x^2/2 + ..., for x=1.5, then squares
// the result.
#include <iostream.h>
main() {
    double x = 0.5;
    double sqrtE=0;
    const int NTERMS=10;
    for (int i=0;i<NTERMS;i++) {
        //
        // calculate x^i
        //
        double xToTheIth=1.0;
        for (int j=0;j<i;j++) xToTheIth *= x;
        //
        // calculate i!
        //
        int N=i;
        int iFactorial=1;
        while (N>0) {
            iFactorial *= N;
            N--;
        }
        //
        // calculate a term in the Taylor series
        //
        double term = xToTheIth/iFactorial;
        //
        // add the term to the sequence
        //
        sqrtE += term;
    }
    cout << sqrtE * sqrtE << endl;
}
```

Bits in red should be moved to functions.

Functions, like variables need **declarations**. The declaration formalizes the **calling sequence**. C++ is type-safe, which means that a function that expects a parameter with a specific type will not accept another, unless a conversion exists.

Sample declarations:

```
int factorial( int N);  
double pow(double x, int y);
```

\*The declaration must precede the first use of the function because the compiler does **type checking**.

Functions also need **definitions**. This is code where the function is actually implemented.

```
#include <iostream.h>  
int factorial (int N);  
double pow(double x, int N);  
  
main() {  
    double x = 0.5;  
    double sqrtE=0;  
    const int NTERMS=10;  
    for (int i=0;i<NTERMS;i++) {  
        sqrtE += pow(x,i)/factorial(i);  
    }  
    cout << sqrtE * sqrtE << endl;  
}  
  
// calculates x raised to the Nth power.  
double pow(double x, int N) {  
    double xToTheIth=1.0;  
    for (int j=0;j<N;j++) xToTheIth *= x;  
    return xToTheIth;  
}  
  
// calculates the factorial  
int factorial (int N) {  
    int iFactorial=1;  
    while (N>0) {  
        iFactorial *= N;  
        N--;  
    }  
    return iFactorial;  
}
```

## Functions (continued)

\*Some variables in the previous example are used in more than one function, e.g. "N". That is all right. Variable that are declared within functions are called local variables. They cannot be seen outside the function in which they are declared.

\* A functions parameters are copied to the function when the function is called. Their value at the end of the function is not copied back to the **calling routine**. That means there is only one way for the data to flow into the function, IN.

\* The result comes back to the calling routine through the **return value** of the function, i.e. the constant or variable following the **return statement**.

\* A function may have one or many arguments.

\* A function may be declared and defined in the same place. This happens when the definition of the function appears before the calling routine. .

\* Function overloading.

It may be advantageous to give two separate functions the same name. For example:

```
double pow (double x, int N);  
double pow (double x, double y);
```

This can be done in C++ and is called function overloading.

\*However, you cannot overload functions on return type alone:

```
double factorial(int N);  
int factorial(int N);    // compiler will barf.
```

\* A function need not return a value. In case it doesn't its return type is void.

Suppose you want a function to modify the value of a function from a calling routine? What do you do?

\* Solution #1 Pass a pointer

```
#include <iostream.h>
// Zero N elements starting with address Array
void zero(int *array, int N) {
    for (int i=0; i<N; i++) {
        array[i]=0;
    }
}
```

```
main() {
    int i = 44;
    int iarr[]={2,4,8,16,32};
    //
    // Print i and iarr before zeroing
    //
    cout << "Before calling zero" << endl;
    cout << i << endl;
    cout << iarr[0] << " "
         << iarr[1] << " "
         << iarr[2] << " "
         << iarr[3] << " "
         << iarr[4] << endl;
    zero(iarr, 5);
    zero(&i,1);
    //
    // Print i and iarr after zeroing
    //
    cout << "After calling zero" << endl;
    cout << i << endl;
    cout << iarr[0] << " "
         << iarr[1] << " "
         << iarr[2] << " "
         << iarr[3] << " "
         << iarr[4] << endl;

}
```

```
Before calling zero
44
2 4 8 16 32
After calling zero
0
0 0 0 0 0
```

Solution #2 Pass by reference Consider the program:

```
#include <iostream.h>
// Zero N elements starting with address Array
void twice(double x) {
    x = x*2.0;
}

main() {
    double X=1;
    for (int n=1;n<10;n++) {
        cout << X << endl;
        twice(X);
    }
}
```

Its output is:

```
1
1
1
1
1
1
1
1
1
1
1
```

On the other hand, change function "twice" like this:

```
#include <iostream.h>
// Zero N elements starting with address Array
void twice(double & x) {
    x = x*2.0;
}
```

and the output is:

```
1
2
4
8
16
32
64
128
256
```

- \* Passing by reference (or pointer) not only allows you to modify variables in the calling routine.....
- \* It is also (sometimes) more efficient than passing-by-value.
- \* Double precision, integer, and character data is small and efficient to pass, anyway
- \* But user-defined types can be arbitrarily large. Memory-to-memory copy can be expensive both in terms of memory and in terms of CPU.
- \* Sometimes, pass-by-reference is chosen for this reason, and not in order to allow modification of values.
- \* In that circumstance one can make it impossible to modify data in the calling sequence by using the `const` qualifier, e.g.

```
void myFunction (const type & name);  
void myFunction (const type * name);
```

"type" could be `int`, `double`, or user-defined types.

## Exercises:

1) Write a function to calculate the square root of a double precision number. The function should work for all values positive of the input parameter. Test the function.

2) Write a program that reads two vectors from the standard input (i.e., the terminal), and calculates:

- The magnitude of each vector.
- Their dot product.
- Their cross product
- The sine and cosine of the angle between the two vectors.

Represent each vector as a three-element array.

## Code reuse in C++

-----

\*In C and C++ functions can be reused if their declaration and implementation are available to others, and if they are documented.

\*Two of the routines we've been using are available as part of the standard C library: `pow` and `sqrt`. Many others are available as well.

\* Most of the standard C++ library involves reusable objects rather than reusable functions. We defer discussion of these for now. But we point out that our input and output so far has been based on these:

\*Documentation:

to read the documentation on the `sqrt` function, type from the unix terminal: `xman`

from the part labelled "Sections 1 of 2" select "3 (subroutines)"

click and read.

\* Accessing the declaration.

close to the top of the the documentation you'll see the include directive `#include <math.h>` This is the name of the header file. By default, the compiler searches for this file in the director `/usr/include`. You may wish to type the header file. The important part of the header file is just the function prototypes, like the prototypes you have created for `pow` and `sqrt`.

## \*Accessing the implementation.

The implementation of the `sqrt` function is kept in a separate file called a library. This code is compiled already; that is, it consists of executable instructions in a form that the computer understands. You need only to link to it.

To link to a library, you need to tell the compiler what file to link to. This is done with an option to the compiler:

The `sqrt` and `pow` functions "live" in the C math library which is a file called `libm.a`. To link to `libm.c`, you need to add the qualifier

`-lm`

To the link line, eg.

```
CC -g myProgram.C -o myProgram -lm
```

If you are using the `make` utility, you can achieve the same effect on unix by creating a file called `Makefile` that overrides the usual rules for building executables:

Your `makefile` should include just two lines: Don't try to understand this for now, just copy! The white space at the beginning of the second line must be a space. Copy all punctuation.

```
.C:
    $(CXX) $(CXXFLAGS) $@.C $(LDFLAGS) -o $@
```

\*Note that the header file contains important information for **clients** of the library, namely, it contains the calling sequence for all of the functions. By contrast, the actual implementation of the `sqrt` and `pow` functions are hidden. They are not important for the client.

\*It is possible to create your own library of code (.a) files. Ask me for details if you're interested.

Exercise:

1) In the previous exercise, use the standard C math routines in place of your own routine for pow and sqrt.