Functions

All C++ programs are composed of functions. Even the smallest C++ program is composed of at least one function, called ***main***.

Modularity: We have already seen the use of program modules -- blocks of code -- integrated into a program, e.g.:

Compound statement -- { }
Selection -- if, if/else
Iteration -- for, while, do/while

Let's extend this concept to modularization, composing the entire program by putting together modules, or subprograms (functions / classes).

Modularization of a program results in:

- Increased logical clarity -- each module represents a specific, well-defined task
- Reduced debugging time
- Improved maintainability, evolvability
- Reduced programming time and smaller programs, since there is no need to repeat the programming of essentially the same set of instructions.

Functions are self-contained program structures. All C++ programs are composed of functions. Even **`main`** is a function.

A function is called (invoked) by use of its name followed by an optional argument list in parentheses. When a function call is executed, control is transferred to the beginning of the function, its statements executed sequentially and, when finished, control returns to the statement following the call.

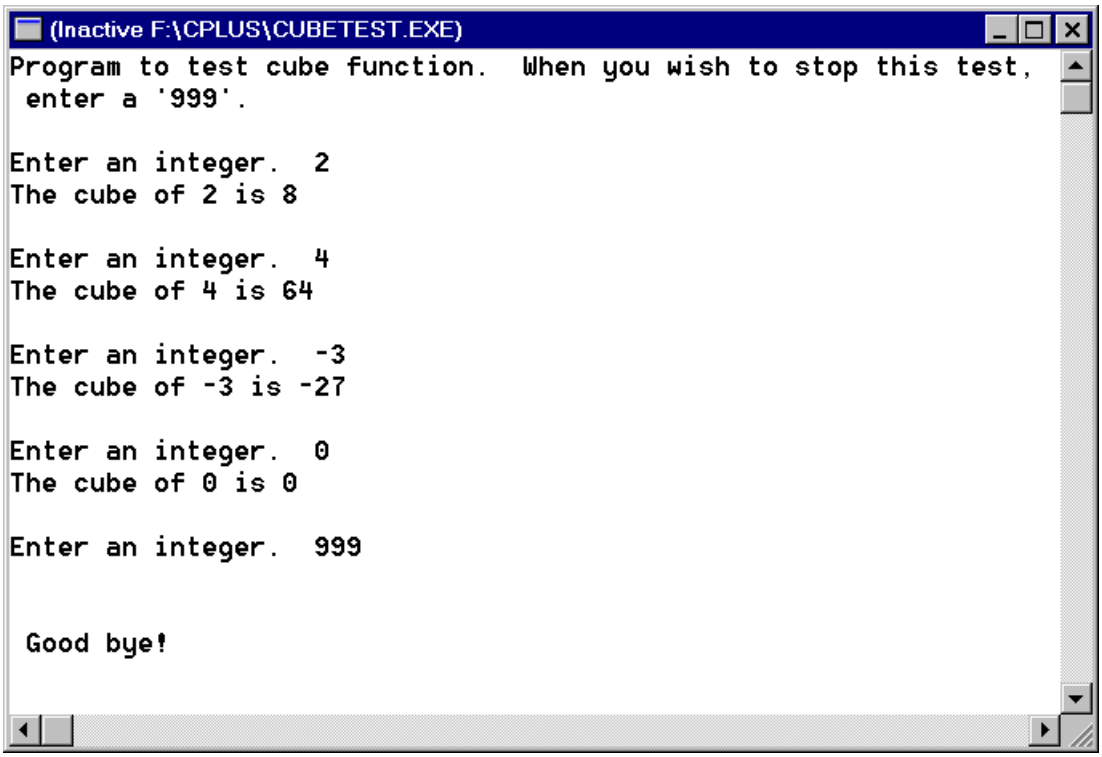
A function can be defined once and then accessed repeatedly at various points throughout the program. It can be compiled separately and stored in a library for repeated use by many different programs.

Every function we use in a C++ program must be declared before it is invoked.
Functions are declared using function prototypes:

```
type functionName (argument-list);
```

It is good practice to place all function prototypes before the `main` function of the program.

```
//function cube tested in program cubetest.cpp
//modified from Hubbard ex.4.4 p.93
#include <iostream>
using namespace std;
int cube(int);          //function declaration (fn prototype)
                        //functions must be declared
                        // before they are used
void main() {
   int n=1;
   cout << "Program to test cube function.  When you wish to stop this
      test,\n "
   << "enter a '999'.\n" << endl;
   cout << "Enter an integer.  ";
   cin >> n;
   while (n!=999) {
                        //function "call"
                        //n is an actual parameter - an argument
      cout << "The cube of " << n << " is " << cube(n) << endl;
      cout << "\nEnter an integer.  ";
      cin >> n;
   }//end while
   cout << "\n\n Good bye! ";
   return;
}//end main function

int cube (int x) {            //function definition
   return x * x * x;          //x is a formal parameter
} //end cube function
```



The prototypes for the built-in library of C++ functions are in the header files that we have been using.

Some header files in the Standard C / C++ library:

ctype.h	declares functions to test characters
float.h	declares constants relevant to floats
limits.h	defines the integer limits on your local system
math.h	declares mathematical functions sqrt(x), rand(), pow(x,y), …
stdio.h	declares functions for standard input and output
stdlib.h	declares utility functions
string.h	declares functions for processing strings
time.h	declares time and date functions
iostream.h	

Where are the definitions (the actual code) for these built-in functions? In the corresponding library files.

Built-in function - comes with the language / compiler
Programmer-defined function - added by the individual programmer - e.g., the cube function, above.

Example of program that uses functions without arguments:

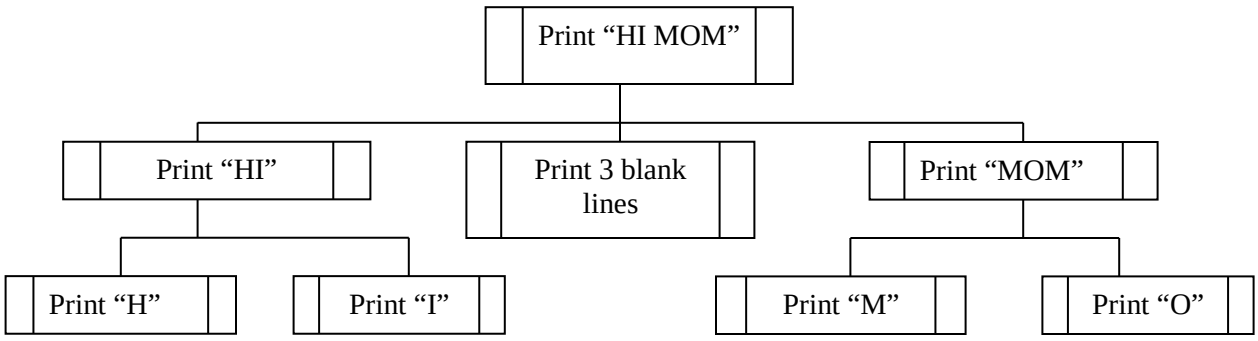
The Mother's Day Problem (a lá Friedman, Koffman, and Koffman):
Mother's Day is coming and you would like to do something special for your mother. Write a C++ program to print the message "HI, MOM" in large capital letters. Your mother will be very impressed, especially if she is paying for your tuition.

Algorithm with stepwise refinement:

Algorithm:
1. Print the word "HI" in large block letters.
2. Print 3 blank lines.
3. Print the word "MOM" in large block letters.

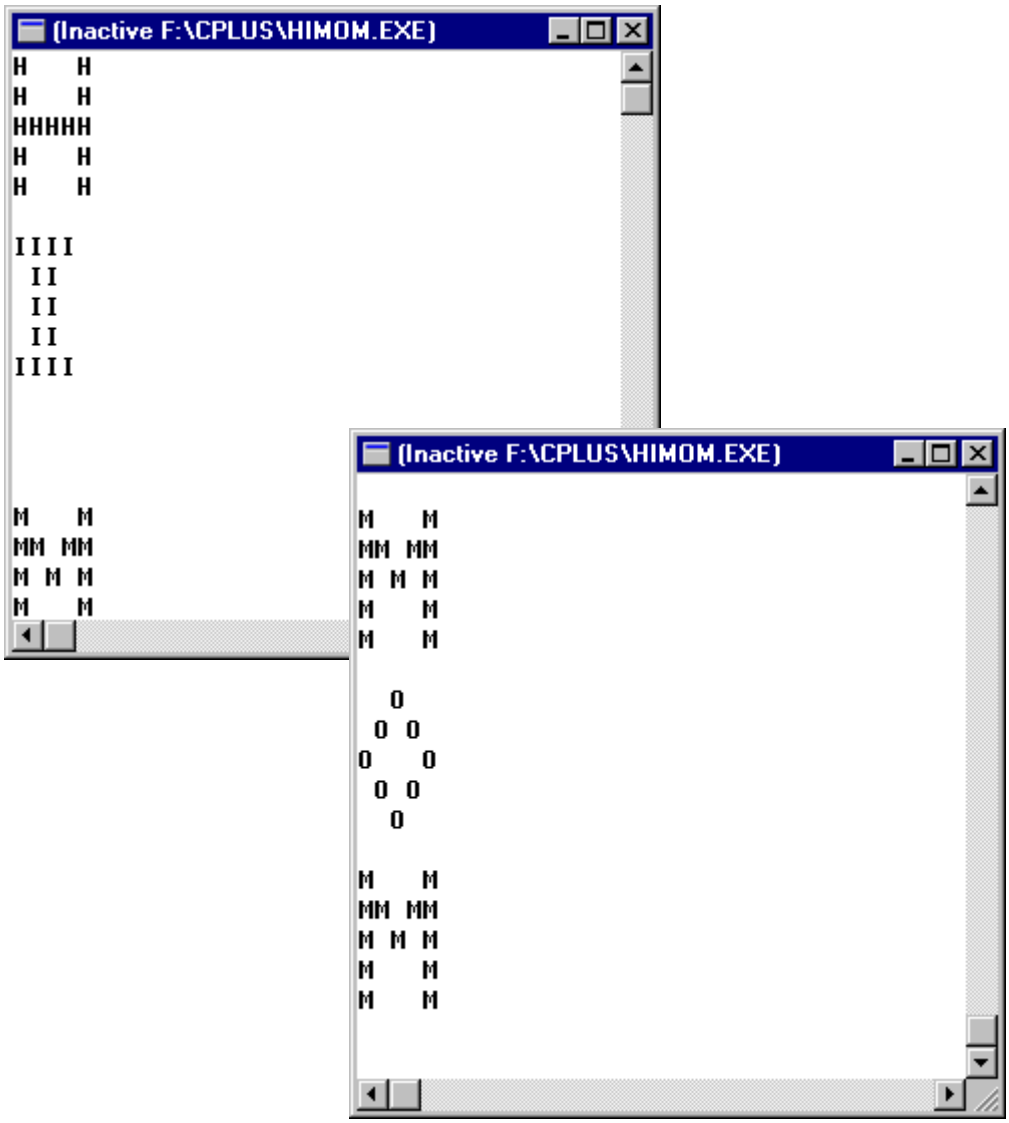
Refinements:
Step 1:
1.1. Print the letter "H".
1.2. Print the letter "I".

Step 3:
3.1 Print the letter "M".
3.2 Print the letter "O".
3.3 Print the letter "M".



```
//HiMom.cpp
// Mother's Day Printout
// Print a Mother's Day message for your Mom

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

// function prototypes
void printHI();         //function to print HI
void printMOM();        //function to print MOM
void printH();               //function to print H
void printI();               //function to print I
void printM();               //function to print M
void printO();               //function to print O

int main()
{
   printHI();
   cout << endl <<endl <<endl;
   printMOM();
   return 0;   //successful termination
} //end main

void printHI()          //function to print HI
{
   printH();
   printI();
} //end printH

void printMOM()         //function to print MOM
{
   printM();
   printO();
   printM();
} //end printMOM

void printH()                //function to print H
{
   cout << "H   H" <<endl;
   cout << "H   H" <<endl;
   cout << "HHHHH" <<endl;
   cout << "H   H" <<endl;
   cout << "H   H" <<endl << endl;
} //end printH

void printI()                //function to print I
{
   cout << "IIII  " <<endl;
   cout << " II   " <<endl;
   cout << " II   " <<endl;
   cout << " II   " <<endl;
   cout << "IIII  " <<endl << endl;
} //end printI
```

```
void printM()                 //function to print M
{
   cout << "M   M" <<endl;
   cout << "MM MM" <<endl;
   cout << "M M M" <<endl;
   cout << "M   M" <<endl;
   cout << "M   M" <<endl << endl;
} //end printM

void printO()                 //function to print O
{
   cout << "  O  " <<endl;
   cout << " O O " <<endl;
   cout << "O   O" <<endl;
   cout << " O O " <<endl;
   cout << "  O  " <<endl << endl;
} //end print
```



Inline function - if the function definition contains the inline specifier, e.g.,

```
inline int cube (int n) {
      return x * x * x;
}//end cube function
```

then the compiler simply expands the function at the point of call, e.g.,

```
cout <<  cube(n)  << endl;
```

becomes

```
cout <<  n * n * n  << endl;
```

in the executable program.

Why inline functions? There is a lot of overhead associated with invoking functions. The inline cube function is still more readable and understandable, but at the same time compiles to more efficient executable code.

Macro Pseudo-functions with #define

An alternative to inline functions.

```
#define cube(x) ((x) * (x) * (x))
#define min(n1,n2) (((n1) < (n2)) ? (n1) : (n2))
#define max(n1,n2) (((n1) > (n2)) ? (n1) : (n2))
```

Do we need so many parentheses? Ex:

```
double num1 = 50, num2 = 5, rslt;
rslt = min(num1 / 2, num2 * 2);
```

same as

```
rslt = (((50 / 2) < (5 * 2)) ? (50 / 2) : (5 * 2));
```

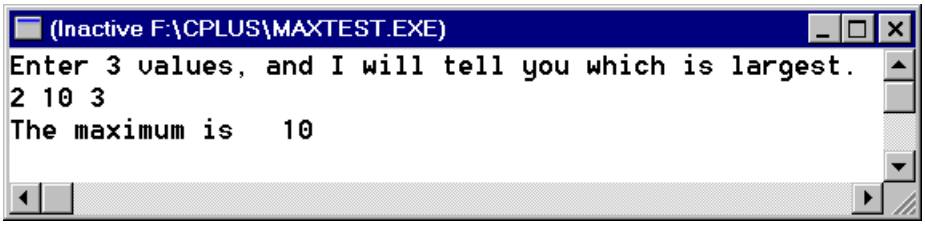
if parentheses had not been used in the macro, we would have:

```
rslt = 50 / 2 < 5 * 2 ? 50 / 2 : 5 * 2;
```

An inline function vs. a #define macro pseudo-function: inline functions do type checking, #define macros do simple text substitution; With inline functions, there is less chance of unintentional changes to critical variables.

Ex.
#define inc(n) ((n)+1) // later

```
//maxtest.cpp
// Use function to find the largest of 3 integer values
// The main function is used only to test max function

#include <iostream>
using namespace std;

int max (int, int, int);

void main(){
   int A, B, C;
   cout << "Enter 3 integers, and I will tell you which is largest.\n";
   cin >> A >> B >> C;
   cout << "The maximum is   " << max(A, B, C) << endl;
   return;
} //end main function

int max (int x, int y, int z){
   int tempmax;
   if (x > y) tempmax = x;
      else tempmax = y;
   if (z > tempmax) tempmax = z;
   return tempmax;
}//end max function
```



Why Functions?

1. Divide and conquer. Any well-designed program is designed in modules in some organized fashion (e.g., top down, object oriented).

- Increased logical clarity -- each module represents a specific, well-defined task -- what vs. how - abstraction - Black Box concept. You don't have to know how it works just what it does. This holds for functions that have already been written and tested (in the library) and also for functions that you will eventually write for your program; delayed coding.
- Reduced debugging time
- Improved maintainability, evolvability

2. Function library. Function libraries contain a wide variety of commonly required functions to perform various tasks. Can be reused over and over and shared among all users in an organization. Both built-in and programmer-defined. Why reinvent the wheel?

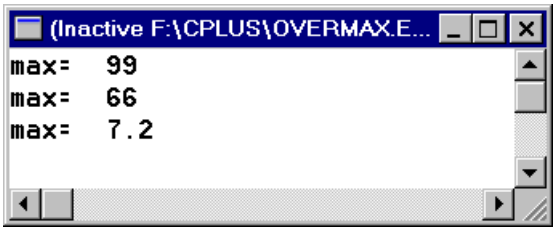
3. Teamwork. Makes it easier for a large group of people to cooperate on a large programming task.

Function Overloading - Function Polymorphism: C++ allows us to declare different functions with the same name; as long as the number or types of the parameters listed are different they are treated as completely different functions.

```
//overloading.cpp
//modified from SAMS 21 days, p.169
#include <iostream>
using namespace std;

int inc(int);
double inc(double);
char inc(char);

void main(){
char c = 'A';
int i = 10;
double x = 10.2;

cout << "c = " << c << endl
   << "i = " << i << endl
   << "x = " << x << endl;
cout << "overloading..." << endl;
cout << "c = " << inc(c) << endl
   << "i = " << inc(i) << endl
   << "x = " << inc(x) << endl;
return;
}

int inc (int n){
return n = n + 1;
}

double inc(double n) {
return n = n + 1;
}

char inc(char n) {
return n = n + 1;
}
```

"same" as #define inc(n) ((n)+1) // ????

```
//overmax.cpp
//modified from Hubbard Ex. 4.21 p.111
//function overloading

#include <iostream>
using namespace std;

//       Function Prototypes
//The following are all considered by the compiler
//to be different functions.
int max(int, int);
int max(int, int, int);
double max(double, double);
double max(double, double, double);

int main(){
      cout << "max=  " << max(99, 77) << endl;
      cout << "max=  " << max(55, 66, 33) << endl;
      cout << "max=  " << max(3.4, 7.2) << endl;
      return 0;
} //end main

int max (int x, int y){return (x > y ? x : y);}

double max (double x, double y){return (x > y ? x : y);}

int max (int x, int y, int z) {
      int m = (x > y ? x : y);
      return (z > m ? z : m);
}

double max (double x, double y, double z) {
      double m = (x > y ? x : y);
      return (z > m ? z : m);
}
```

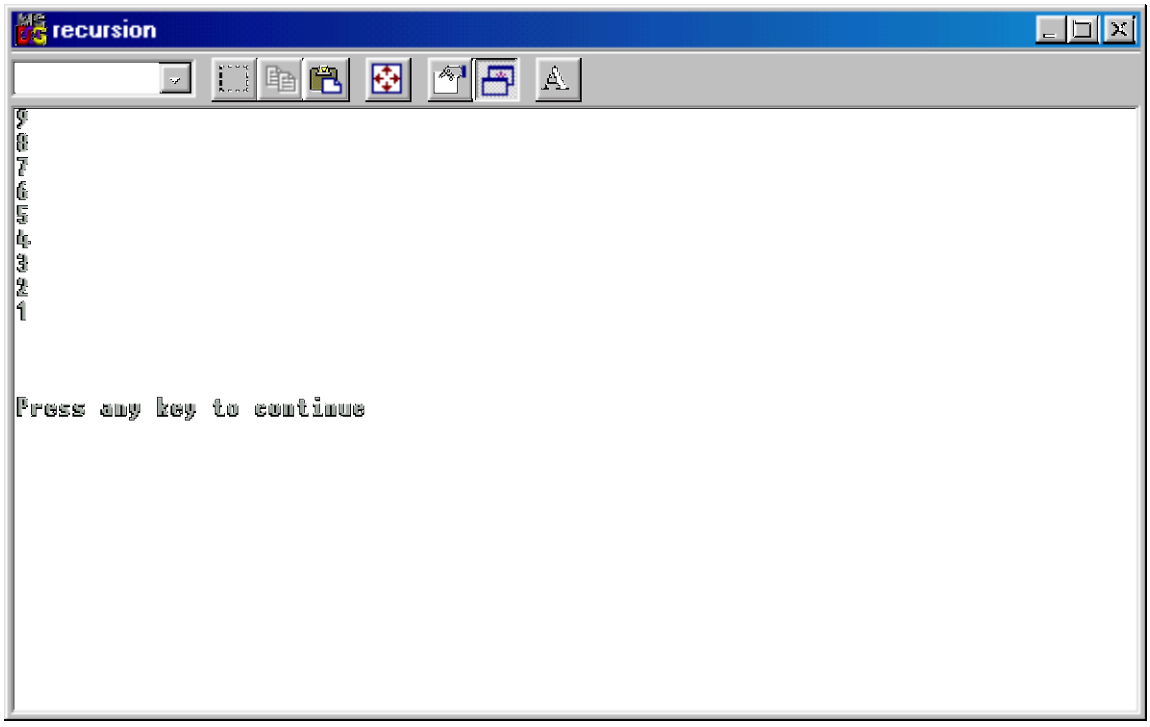


Recursive functions

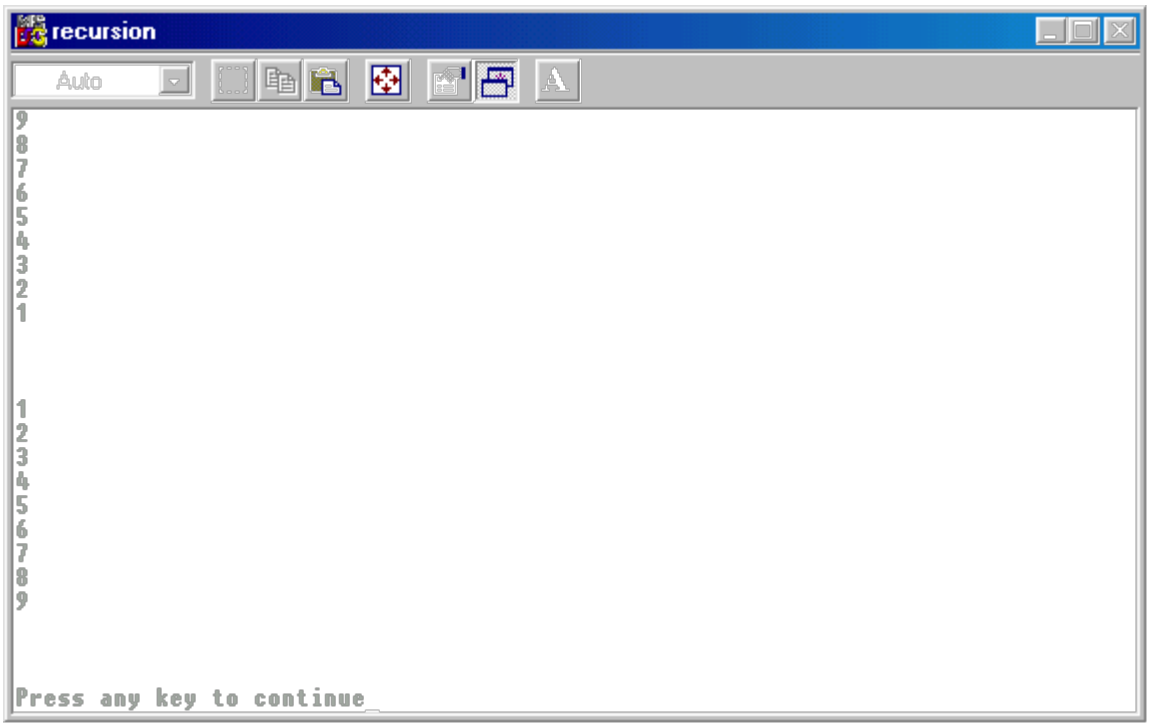
Many problems can best be solved by breaking them down into smaller, similar problems.

A recursive function is one that calls itself. Recursion can be direct or indirect.

```
//recursion.cpp
#include <iostream>
using namespace std;

void print1to (int);
void print (int);

main(){
   int n=9;
   print1to(n);
   cout << "\n\n\n";
   return 0;
}

void print1to(int a){
   if (a==1) print(1);
   else {
      print (a);
      print1to(a-1);
   }
   return;
}

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

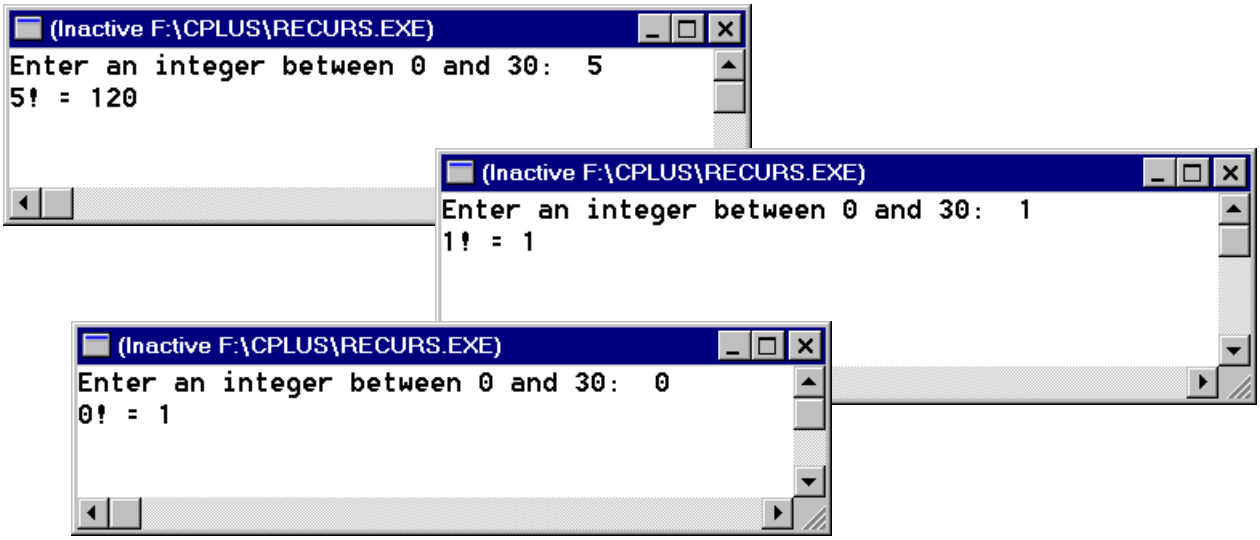


```
//recursion.cpp

#include <iostream>
using namespace std;

void print1to (int);
void print1upto (int);
void print (int);

main(){
   int n=9;
   print1to(n);
   cout << "\n\n\n";
   print1upto(n);
   cout << "\n\n\n";
   return 0;
}

void print1to(int a){
   if (a==1) print(1);
   else {
      print (a);
      print1to(a-1);
   }
   return;
}

void print1upto (int a){
   if (a==1) print(1);
   else {
      print1upto (a-1);
      print (a);
   }
   return;
}

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


recursion
Auto
9
8
7
6
5
4
3
2
1
1
2
3
4
5
6
7
8
9
Press any key to continue

Some problems are inherently, recursive, e.g., the definition of a factorial:
$n! = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (2) \cdot (1) = n \cdot (n-1)!$

```
//recursive factorial function
#include <iostream>
using namespace std;
double factorial (int);

const int MIN = 0;
const int MAX = 30;

void main(){
int n;
do{
      cout << "Enter an integer between " <<MIN<< " and " << MAX << ":
";
      cin >> n;
}while (n < MIN || n > MAX);
cout << n << "! = " << factorial(n) << endl;
return;
}

double factorial (int x){
if (x <=1) return 1;
      else return x * factorial (x-1);  //QUESTN: is the else needed?
}
```



Also, this one:

```
//recursive factorial function
//with some input validation

#include <iostream>
using namespace std;

double factorial (int);
int getInt();

const int MIN = 0;
const int MAX = 30;

void main(){
   int n;
   n = getInt();
   cout << n << "! = " << factorial(n) << endl;
   return;
}

int getInt(){
   int number;
   cout << "Enter an integer between " <<MIN<< " and " << MAX << ":  ";
   cin >> number;
   while (number < MIN || number > MAX) {       //Bad Data
      cout << "\n\tThat number was incorrect."
         <<"\n\tPlease try again.\n" ;
      cout << "\nEnter an integer between " <<MIN<< " and "
         << MAX << ":  ";
      cin >> number;
   }
   return number;
}

double factorial (int x){
if (x <=1) return 1;
   else return x * factorial (x-1);
}
```

Let's refine the input validation function:

Why do we insist that the user must continue to enter data indefinitely until the correct kind of data value is entered? Alternative: We can give the user options in a menu, either enter a data value, or quit. Another alternative, just accept data a certain number of times, the "three strikes and you're out approach." How to do that? Maybe something like this:

```
int getInt(){
   int number;
   for (int i=1; i<=3; i++){
      cout << "Enter an integer between " <<MIN<< " and "
         << MAX << ":  ";
      cin >> number;
      if (number < MIN || number > MAX) {
         cout << "\n\tThat number was incorrect."
            <<"\n\tPlease try again.\n" ;
      } //end 'then' block
      else break;
      } //end for loop
   return number;
}
```

Problem: This function will return number even if it proved to be invalid. Let's try again. Set up a flag that can be tested in main to determine whether the number is valid or not.

```
int getInt(){
   int number;
   for (int i=1; i<=3; i++){
      cout << "Enter an integer between " <<MIN<< " and "
         << MAX << ":  ";
      cin >> number;
      if (number < MIN || number > MAX) {
         cout << "\n\tThat number was incorrect."
            <<"\n\tPlease try again.\n" ;
         } //end 'then' block
      else break;
   } //end for loop
   if (i>3)                  // test to see if for loop "ran through"
      return 0;              // valid data not entered
   else
      return 1;              // data entered is valid
   }
```

Problem:

We now have a testable flag sent back to main() but we no longer have the number itself. It remains in the function's domain. Let's try to create a Boolean function with an output parameter, number.

```
bool getInt(int number){

   for (int i=1; i<=3; i++){
      cout << "Enter an integer between " <<MIN<< " and "
         << MAX << ":  ";
      cin >> number;
      if (number < MIN || number > MAX) {
         cout << "\n\tThat number was incorrect."
            <<"\n\tPlease try again.\n" ;
         } //end 'then' block
      else break;
   } //end for loop
   if (i>3)                    // test to see if for loop "ran through"
      return 0;                // valid data not entered
   else
      return 1;                // data entered is valid
   }
```

Then, in main() :

```
...
if (getint(n))                 // replaces n=getint() statement
cout << n << "! = " << factorial(n) << endl;
else
cout << "Sorry, we'll do this another day.\n\n"
...
```

Note: A Boolean function is one that returns a boolean value [true (1) or false (0)].

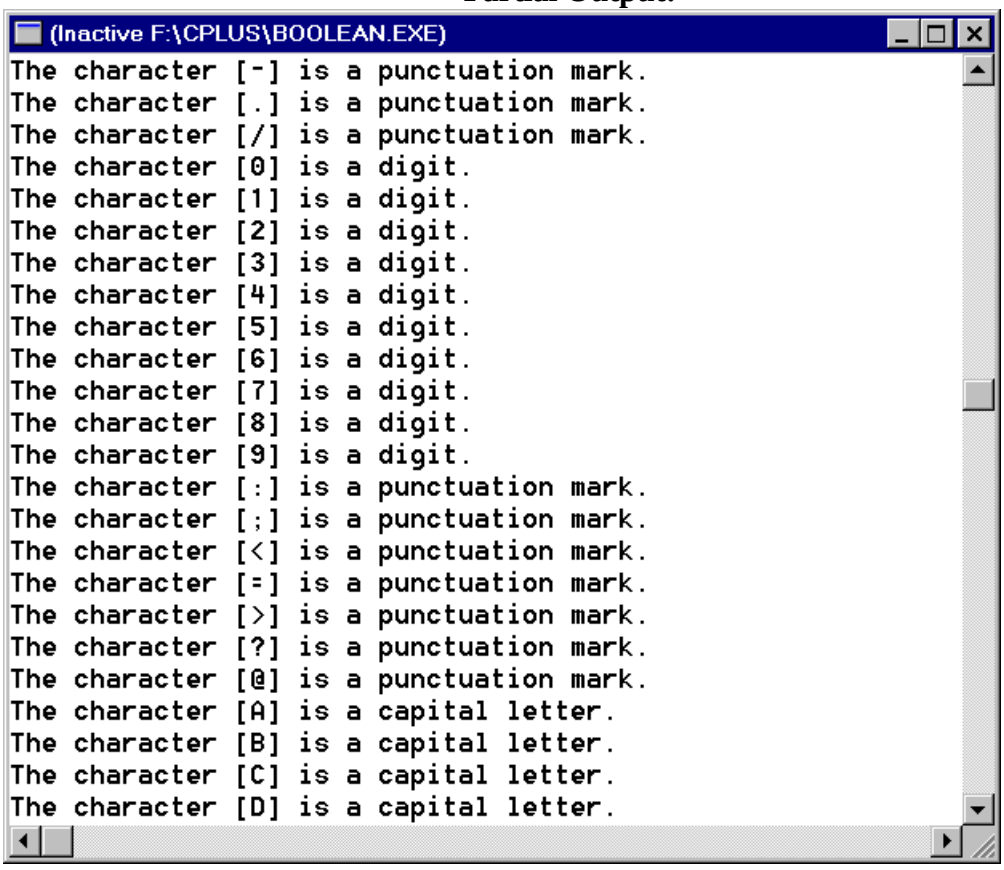
This ALMOST works. We will return to this problem shortly with a workable program.

More Boolean Functions

Return '1' for true, '0' for false. Example, using ctype library functions:

```
//boolean.cpp
//modified from Hubbard ex 4.11 p.100
#include <iostream>
#include <ctype>
using namespace std;
void printCharCategory (char); //function prototype

void main() {
   for (int c=0; c<128; c++) printCharCategory (c);
}//end main function

// Prints the category to which a given character belongs
void printCharCategory (char c) {
   cout << "The character [" << c << "] is a ";
   if (isdigit (c)) cout << "digit.\n";
   else if (islower (c)) cout << "lower case letter. \n";
   else if (isupper (c)) cout << "capital letter. \n";
   else if (isspace (c)) cout << "whitespace character. \n";
   else if (iscntrl (c)) cout << "control character. \n";
   else if (ispunct (c)) cout << "punctuation mark. \n";
   else cout << "Error.\n";
}//end printCharCategory function
```

Partial Output:



Parameter Passing

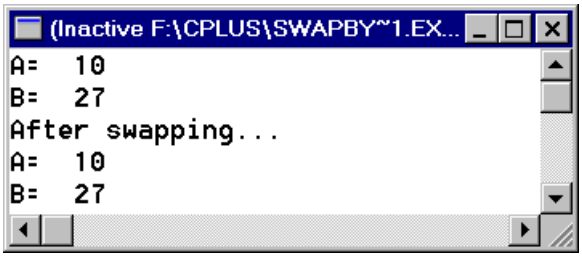
Passing by Value - This is the default.

Suppose we wish to write a function that will swap two numeric values. It might look something like this:

```
void swap (float x, float y){
   float temp = x;
   x = y;
   y = temp;
}//end swap
```

How does this work? Let's see (it doesn't). Here's a test driver program to test the swap function:

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

void swap (float, float);

int main(){
float a = 10;
float b = 27;
cout << "A=  " << a << endl
     <<"B=  " << b << endl;
swap(a,b);
cout << "After swapping..." << endl
     << "A=  " << a << endl
     <<"B=  " << b << endl;
return 0;
}//end main

void swap (float x, float y){
float temp = x;
x = y;
y = temp;
}//end swap
```



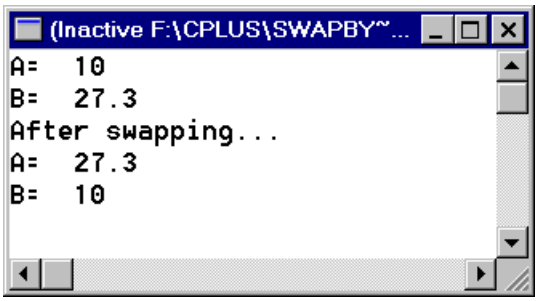
Why didn't the swap take place? After all, the function looks like it should work, but it doesn't work properly.

x and y, the formal parameters of swap are considered local to swap. Only the values of a and b (10 and 27) are passed to the swap function. Any changes that take place inside the function, stay there and do not get sent back to the actual parameters in the calling function. This is to protect our variables from unintentional modification. And is usually the best way to pass actual parameters (arguments) to formal parameters. Clearly, this doesn't work in all cases!

For the swap function to work as intended, we need to pass a reference to the location of the actual parameter so that it can be modified by the function. This is called **Passing by Reference**.

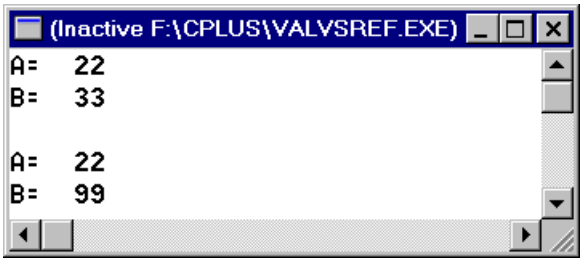
Passing by Reference:

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

void swap (float&, float&);

int main(){
   float a = 10;
   float b = 27.3;
   cout << "A=  " << a << endl
      <<"B=  " << b << endl;
   swap(a,b);
   cout << "After swapping..." << endl
      << "A=  " << a << endl
      <<"B=  " << b << endl;
   return 0;
}//end main

void swap (float &x, float &y){
   float temp = x;
   x = y;
   y = temp;
}//end swap
```

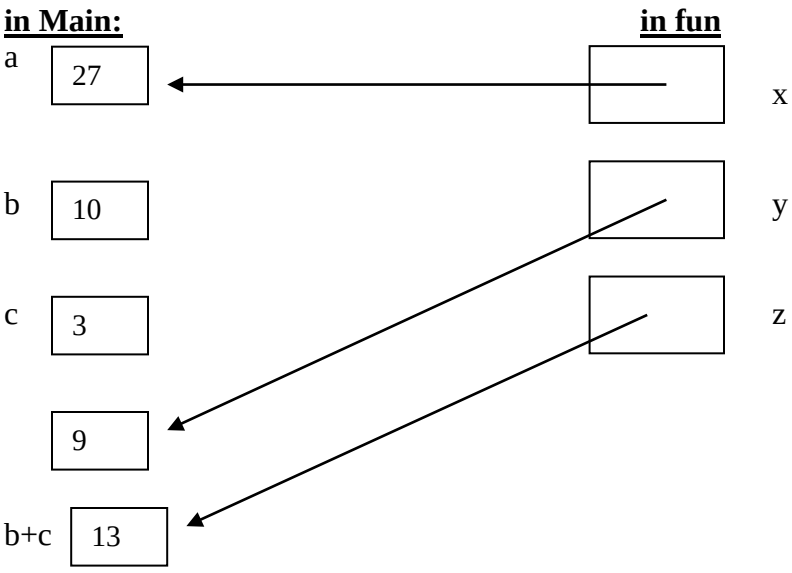


Example - What will output?

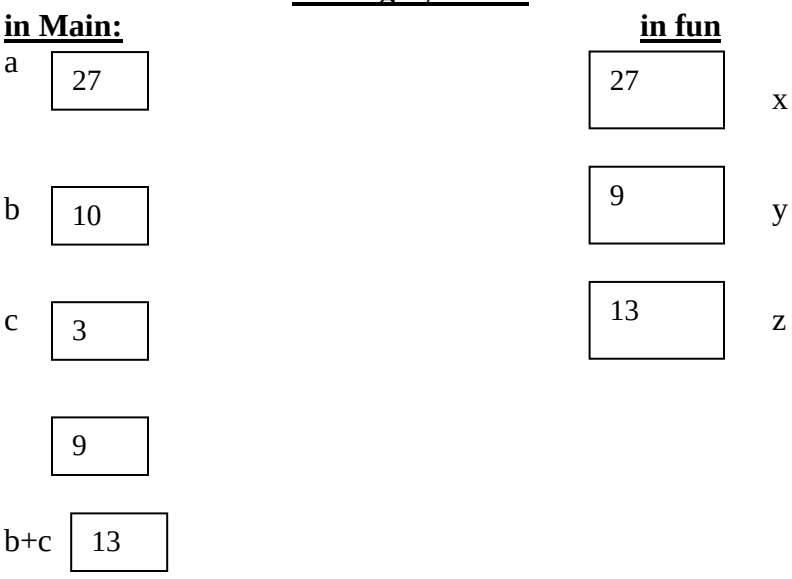
```
//from Hubbard ex.4.16, p. 106
#include <iostream>
using namespace std;

void testing(int, int&);

int main(){
   int a = 22, b = 33;
   cout << "A=  " << a << endl
      <<"B=  " << b << endl << endl;
   testing(a,b);
   cout << "A=  " << a << endl
      <<"B=  " << b << endl;
   return 0;
}

void testing (int x, int &y) {
   x = 88; y = 99;
}
```



```
#include <iostream>
void fun (int&, int&, int&);
int main(){
int a, b, c;
…
a = 27; b = 10; c = 3;
fun (a, 9, b+c);
cout << a;
…
return 0;
}
fun (int &x, int &y, int &z){
x = x + y + z;
}
```



Passing by value:



Passing by Value	***Passing by Reference***
int x;	int &x;
Formal parameter is a local variable	Formal parameter is a local reference
Formal parameter is a duplicate of the actual parameter	Formal parameter is a synonym for the actual parameter
Formal parameter cannot change the actual parameter	Formal parameter can change the actual parameter
Actual parameter may be constant, variable, or expression	Actual parameter must be a variable
Actual parameter is read-only	Actual parameter is read-write

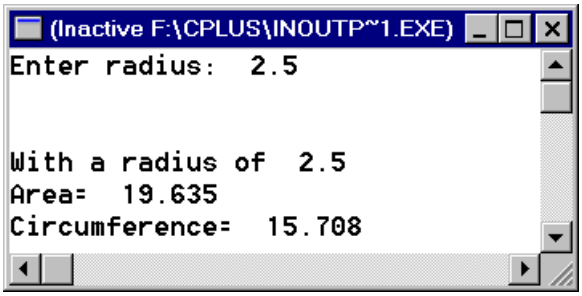
Input and Output Parameters

Input parameters send data to a function, e.g., in sqrt(x), x is an input parameter. The output from the function is the function itself which returns with the value of the square root.

Some functions do not return any value and so are of type void. Some functions must return more than one output value, and we cannot use the function itself to return more than one value. We do this using output parameters, passed by reference.

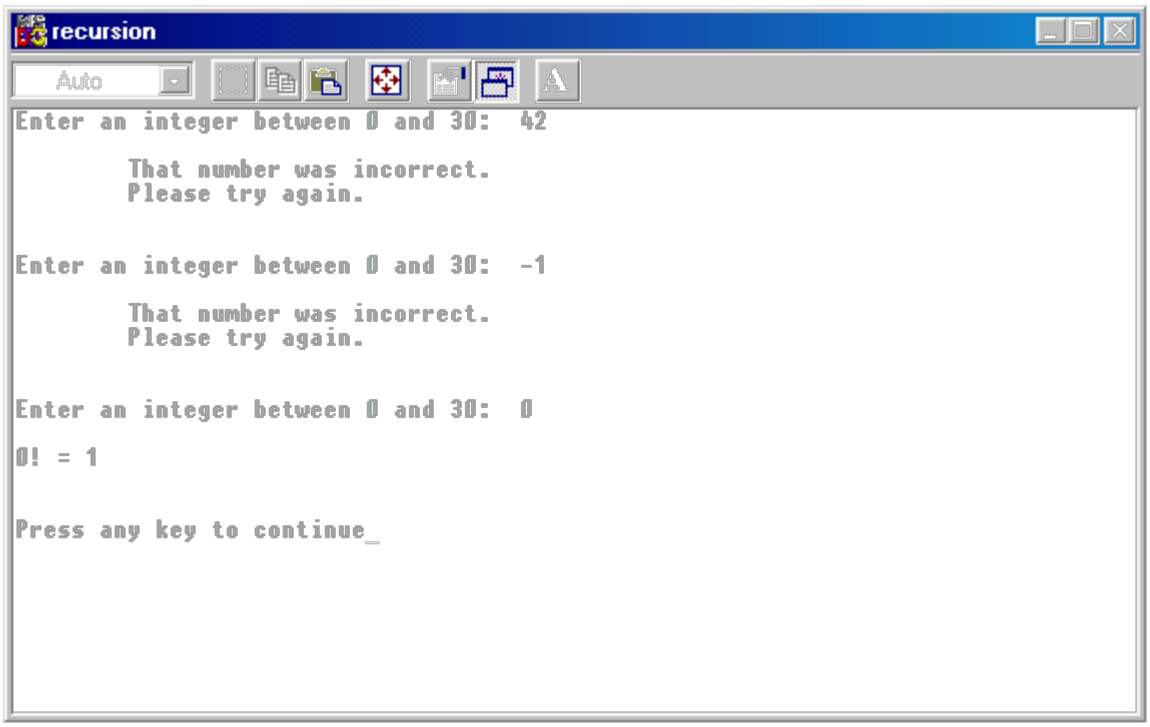
Example:

```
//inoutparms.cpp
//modified from Hubbard, ex. 4.17, p. 107
//computing the area and circumference of a circle

#include <iostream>
void ComputeCircle(double, double&, double&);

int main(){
   double radius, area, circumference;
   cout << "Enter radius:  "; cin >> radius;
   ComputeCircle(radius, area, circumference);
   cout << "\n\nWith a radius of  " << radius << endl
      << "Area=  " << area << endl
      << "Circumference=  " << circumference << endl;
   return 0;
} //end main

void ComputeCircle(double r, double &a, double &c) {
   const double PI = 3.141592653589793;
   a = PI * r * r;
   c = 2 * PI * r;
} //end ComputeCircle
```



Back to factorial w/ input validation example:

```
//recursive factorial function
//boolean input/validation function
//factorial.cpp

#include <iostream>
using namespace std;

double factorial (int);
bool getint (int&);

const int MIN = 0;
const int MAX = 30;

int main(){
   int n;
   if (getint(n))
      cout << n << "! = " << factorial(n) << endl;
   else
      cout << "Sorry, we'll do this another day.\n\n";
   cout << endl << endl;

   return 0;
}

bool getint (int &number){
   for (int i=1; i<=3; i++){
      cout <<"Enter an integer between " <<MIN<<" and " <<MAX << ":  ";
      cin >> number;
      if (number < MIN || number > MAX) {
         cout << "\n\tThat number was incorrect."
            <<"\n\tPlease try again.\n\n\n" ;
         } //end 'then' block
      else break;
   } //end for loop
   cout << endl;
   if (i>3)    // test to see if for loop "ran through"
      return 0;   // valid data not entered
   else
      return 1;   // data entered is valid
}

double factorial (int x){
if (x <=1) return 1;
   else return x * factorial (x-1);
}
```


recursion
Auto
Enter an integer between 0 and 30: 42
That number was incorrect.
Please try again.
Enter an integer between 0 and 30: -1
That number was incorrect.
Please try again.
Enter an integer between 0 and 30: 0
0! = 1
Press any key to continue_

Scope

[Lifetime, visibility – storage class].

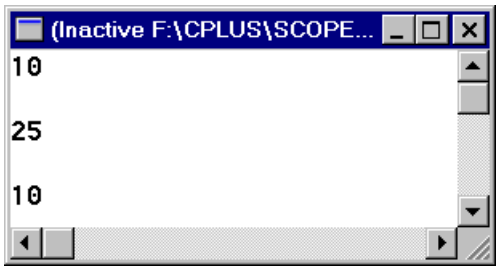
Every name in a C++ program must refer to a unique entity. This does not mean that a name can be used only once.

Sometimes context is important.

Scope – 3 forms: local scope,
namespace scope, class scope. LATER

local scope – function, compound statement

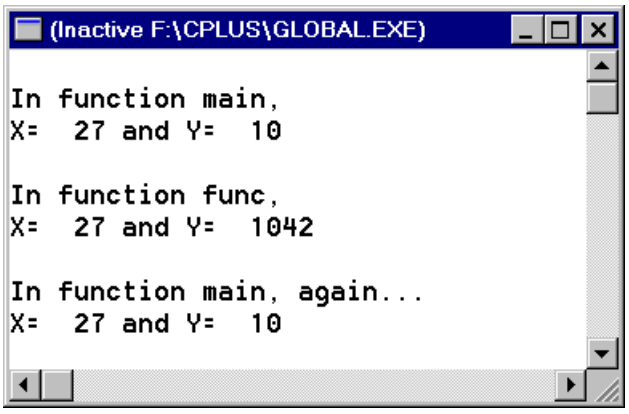
```
#include <iostream>
using namespace std;

int main(){
   int x=10;
   cout << x << endl << endl;
   {
      int x=25;
      cout << x << endl << endl;
   }
   cout << x << endl << endl;
   return 0;
}
```



Global and Local Variables

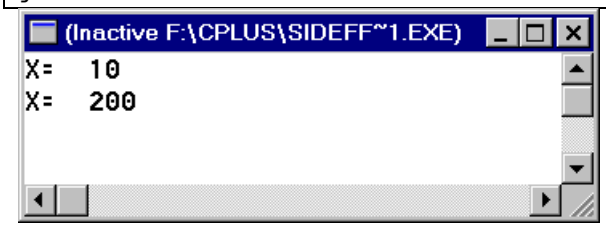
External variables = global variables
Automatic variables = local variables
Static variables

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

void func();                    //function prototype

int x=27, y=10;                 //global variables

int main(){
   cout << "\nIn function main, " << endl;
   cout << "X=  " << x << " and Y=  " << y << endl;
   func();
   cout << "\nIn function main, again..." << endl;
   cout << "X=  " << x << " and Y=  " << y << endl;
   return 0;
}

void func(){
   int y = 1042;                local variable
   cout << "\nIn function func, " << endl;
   cout << "X=  " << x << " and Y=  " << y << endl;
}
```



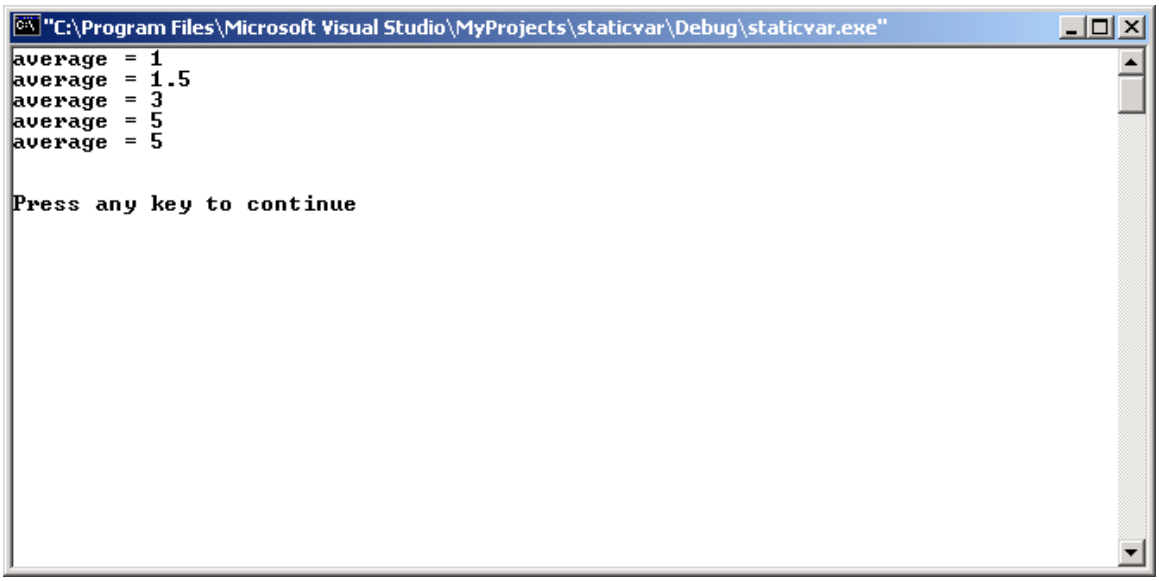
Side Effects - unintended changes to global variables from inside a function

```
#include <iostream>
using namespace std;
void SideEffects();
int x;

int main(){
   x=10;
   cout << "X=  " << x << endl;
   SideEffects();       // not clear here why the function call might
                        //change the value of x
   cout << "X=  " << x << endl;
   return 0;
}

void SideEffects(){
   x = 200;
}
```



Static variables

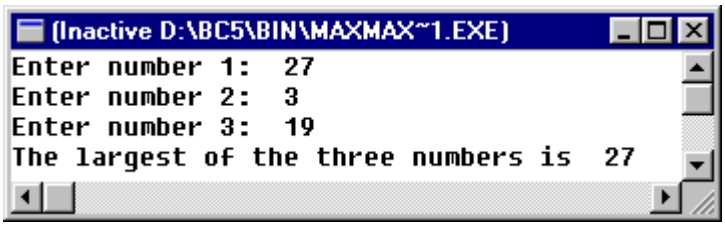
```
#include <iostream>
using namespace std;

double average (double x){
	static double count = 0;
	static double sum = 0;

	count++;
	sum += x;
	return sum / count;
}

int main(){
	cout << "average = " << average(1) << endl;
	cout << "average = " << average(2) << endl;
	cout << "average = " << average(6) << endl;
	cout << "average = " << average(11) << endl;
	cout << "average = " << average(5) << endl;
	cout << endl << endl;
	return 0;
}
```



Exercise:

Write a function, *max (int, int, int)*, to find the maximum of 3 integers. Have it make use of a *max(int, int)* function. One way is:

```
//maxmaxint.cpp
//from Hubbard, prb. 4.18
//Write a maximum function for three integers that uses
//a maximum function for two integers.
#include <iostream>
int max (int, int);
int max (int, int, int);

int main (){
   int a, b, c;
   cout << "Enter number 1:  " ;
   cin >> a;
   cout << "Enter number 2:  " ;
   cin >> b;
   cout << "Enter number 3:  " ;
   cin >> c;
   cout <<"The largest of the three numbers is  "
      << max (a,b,c) << endl;
   return 0;
}

int max (int x, int y){
   if (x > y) return x;
   else return y;
}

int max (int x, int y, int z) {
   return max(max(x,y),z);
}
```



OR:

A call to max (max(a,b),c)