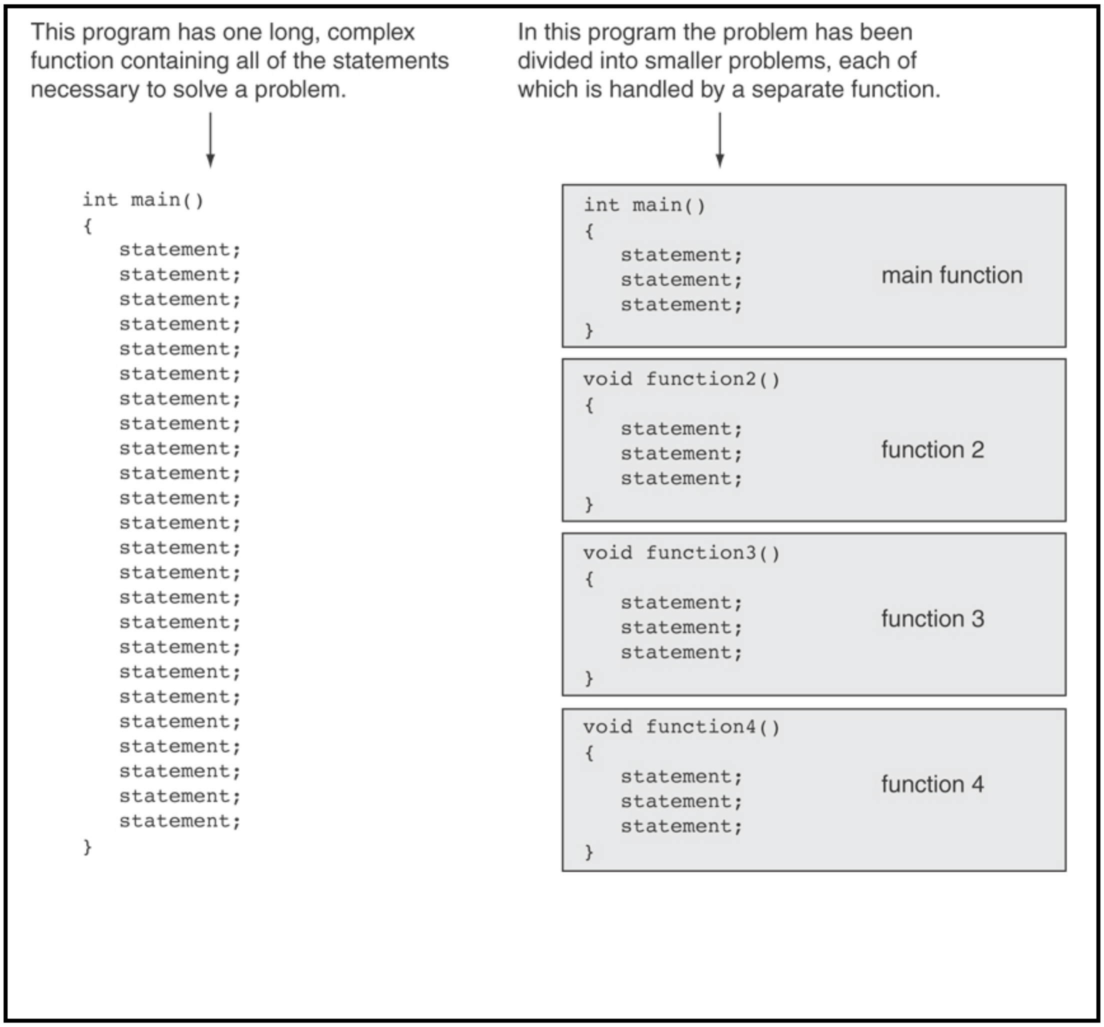
Functions

Modular Programming

- Modular programming: breaking a program up into smaller, manageable functions or modules

- Function: a collection of statements to perform a specific task

- Motivation for modular programming:
  - Improves maintainability of the programs
  - Simplifies the process of writing programs



This program has one long, complex function containing all of the statements necessary to solve a problem.

```
int main()
{
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
   statement;
}
```

In this program the problem has been divided into smaller problems, each of which is handled by a separate function.

```
int main()
{
   statement;
   statement;
   statement;
}
```

main function

```
void function2()
{
   statement;
   statement;
   statement;
}
```

function 2

```
void function3()
{
   statement;
   statement;
   statement;
}
```

function 3

```
void function4()
{
   statement;
   statement;
   statement;
}
```

function 4

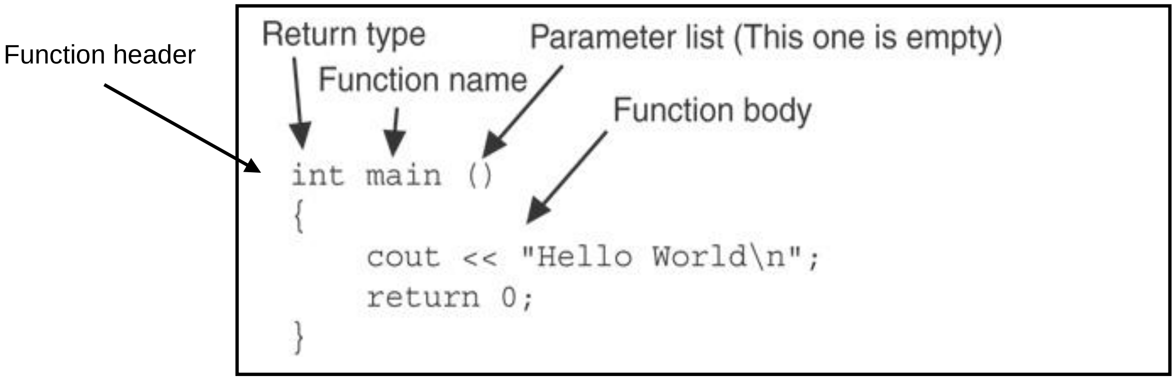
Defining and Calling Functions

Function call: statement that causes a function to execute
Function definition: statements that make up a function

Definition includes:

- Return type: data type of the value that the function returns to the part of the program that called it
- Name: name of the function. Function names follow the same rules as variables
- Parameter list: variables containing values passed to the function
- Body: statements that perform the function's task, enclosed in { }

```
return-type function-name (parameter declarations - if any)
{
  Statement / s
}
```



```
int main ()
{
    cout << "Hello World\n";
    return 0;
}
```

Function Return Type

- If a function returns a value, the type of the value must be indicated:

```
int main()
```

- If a function does not return a value, its return type is **void**:

```
void printHeading()
{
     cout << "Monthly Sales\n";
}
```

Calling a Function

- To call a function, use the function name followed by ()

```
printHeading();
```

- When called, the program executes the body of the called function
- After the function terminates, execution resumes in the calling function after the function call.

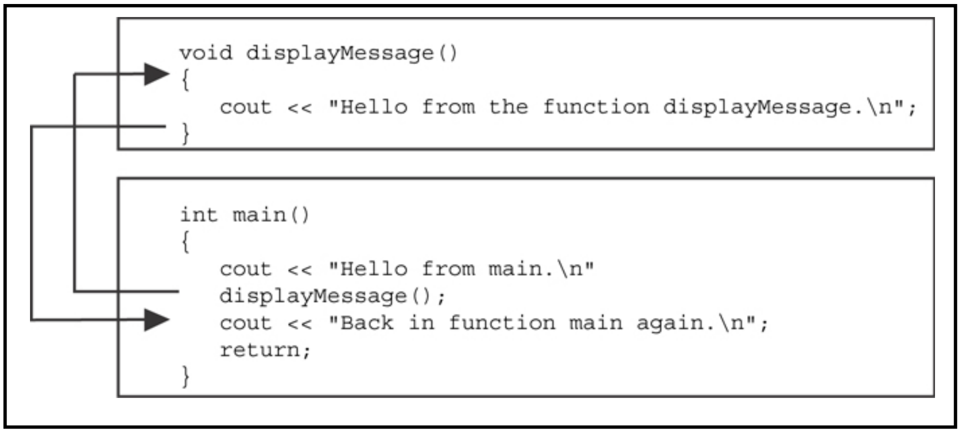
Example

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

void displayMessage()

{
     cout << "Hello from the function playMessage.\n ";
}


int main()
{
     cout << "Hello from Main.\n ";
     displayMessage();
     cout << "Back in function Main again.\n ";
     return 0;
}
```

Output

```
Hello from Main.
Hello from the function displayMessage.
Back in function Main again.
```

Flow of Control:



```
void displayMessage()
{
   cout << "Hello from the function displayMessage.\n";
}

int main()
{
   cout << "Hello from main.\n"
   displayMessage();
   cout << "Back in function main again.\n";
   return;
}
```

Control always starts from main

Calling Functions

- Main program can call any number of functions
- Functions can call other functions
- Compiler must know the following about a function before it is called:
  - name
  - return type
  - number of parameters
  - data type of each parameter

Function Prototypes

Two ways to notify the compiler about a function before it encounters a call to the function:

- Place the function definition before all calls to that function.

- Place a function prototype (function declaration) before all calls to that function
  - ✴ Prototype looks like the function header
  - ✴ Example: `void printHeading();`

Prototypes in a program

Example:

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

// function prototypes

void first();
void second();

int main()
{
      cout << "I am starting in function main.\n";
      first();
      second();
      cout << "Back in function main again.\n";
      return 0;
}

// function definition

void first()
{
      cout << "I am now inside the function first.\n";
}

void second()
{
      cout << "I am now inside the function second.\n";
}
```

Output

```
I am starting in function main.
I am now inside the function first.
I am now inside the function second.
Back in function main again.
```

Prototype Notes

- Place prototypes near the top of the program (before any other function definitions) - good programming style

- Program must include **either** a **prototype** **or** **full function definition** before any call to the function

    - Otherwise: compiler error

- With prototypes, you can place function definitions in any order in the source file

- Common style: all function prototypes at beginning, followed by definition of main, followed by other function definitions.

Example

```
/*
 * BF.cpp
 *
 *  Author: Husain Gholoom
 */


#include <iostream>

using namespace std;
// Function prototype
void Addition() ;
void Subtraction() ;
void Division() ;
void Multiplication() ;

int main()
{
      cout<<"The function of this program is to Simulate a basic "<<endl;
      cout<<"calculator. The operations are (  / * +  and - ) "<<endl;
      cout<<"Enter The + - * or /   ";
      char   op;
      cin>>  op;

      switch (op){
      case '*' :    Multiplication();
                    break;
      case '/' :    Division();
                    break;
      case '+' :    Addition();
                    break;
      case '-' :    Subtraction();
                    break;
      default : cout<<" Your Entered a Wrong Operation Symbol";
}

    return 0;

}
// function definition
void Addition() {
             int firstNumber, secondNumber, opResult;
             cout<<endl;
             cout<<"Enter Two Numbers to be Added    ";
             cin>> firstNumber>>secondNumber;
             opResult = firstNumber + secondNumber;
             cout<<endl;
             cout<<" The result of Adding  "<<firstNumber<<
                      "  and  "<<secondNumber<<"   is =   "<< opResult;
}
```

```
void Subtraction() {

             int firstNumber, secondNumber, opResult;
             cout<<endl;
             cout<<"Enter Two Numbers to be Subtracted    ";
             cin>> firstNumber>>secondNumber;
             opResult = firstNumber - secondNumber;
             cout<<endl;
             cout<<" The result of Subtracting  "<<firstNumber<<
                      "  from  "<<secondNumber<<"   is =   "<< opResult;
             cout<<endl;
}

void Division() {
             int firstNumber, secondNumber, opResult;
             cout<<endl;
             cout<<"Enter Two Numbers to be Divided    ";
             cin>> firstNumber>>secondNumber;
             opResult = firstNumber / secondNumber;
             cout<<endl;
             cout<<" The result of Dividing  "<<firstNumber<<
                      "  by  "<<secondNumber<<"   is =   "<< opResult;
}


void Multiplication() {
             int firstNumber, secondNumber, opResult;
             cout<<endl;
             cout<<"Enter Two Numbers to be Multiplied    ";
             cin>> firstNumber>>secondNumber;
             opResult = firstNumber * secondNumber;
             cout<<endl;
             cout<<" The result of Multiplying  "<<firstNumber<<
                      "  and  "<<secondNumber<<"   is =   "<< opResult;
}
```

Sending Data into a Function

- You can pass values to a function through the function call:

c = pow(a, 2);

- Expressions (or values) passed to a function are called **arguments**
- Variables in a function that accept the values passed as arguments are called **parameters**

A Function with a Parameter

```
void displayValue(int num)
{
cout << "The value is " << num << endl;
}
```

- num is the **parameter**. It accepts int arguments.
- Calls to this function must have an **argument** of type int.

```
displayValue(5);
```

Parameters, Prototypes, and Function Headers

- The prototype must include the data type of each parameter inside its parentheses

- The header must include a declaration for each parameter in its () (data type + param name)

- The call must include an expression for each parameter, inside its parentheses.

Example:

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

// Function Prototype

void displayValue(int);

// Beginning of the program

int main() {
      cout << "I am passing the argument 5 to displayValue.\n";
      displayValue(5);
      cout << "Back in function main again.\n";
        return 0;
}

// Function definition

void displayValue(int num) {
      cout << "The value is " << num << endl;
}
```

```
Output:
I am passing 5 to displayValue.
The value is 5
Back in function main again.
```

Passing Multiple Arguments to Functions

- A function can have multiple parameters
- When calling a function and passing multiple arguments:
  - The number of arguments in the call must match the prototype and definition
  - The value of the argument expression is copied into the parameter (using initialization) when the function is called
  - There must be a data type listed in the prototype and a parameter declaration in the function header for each parameter
  - The first argument will be used to initialize the first parameter, the second argument to initialize the second parameter, etc.
  - A parameter’s scope is the function which uses it

Calculator Example Revisited

```
/*
 *  BasicFunctions.cpp
 *
 *  Author: husaingholoom
 *  The function of this program is to Simulate a basic
 *  calculator. The operations are (  / * + - )
 *
 */


void Addition(int firstNumber, int secondNumber) ;
void Multiplication(int firstNumber, int secondNumber) ;
void Division(int firstNumber, int secondNumber)  ;
void Subtraction(int firstNumber, int secondNumber) ;

#include<iostream>
using namespace std;


int main()
{
      int firstNumber, secondNumber;
      char op;
      cout<<"Enter Two Numbers     ";
      cin>>firstNumber>>secondNumber;
      cout<<endl;
      cout<<"Enter The + - * / \t\t     ";
      cin>>op;

      switch (op){
      case '*' :  Multiplication(firstNumber , secondNumber);
                  break;
      case '/' :  Division(firstNumber , secondNumber);
                  break;
      case '+' :  Addition(firstNumber , secondNumber);
                  break;
      case '-' :  Subtraction(firstNumber , secondNumber);
                  break;
      default : cout<<" Your Entered a Wrong Operation Symbol";
      }

    return 0;
}

void Multiplication(int firstNumber, int secondNumber) {
            int    opResult;
            cout<<endl;
            opResult = firstNumber * secondNumber;
            cout<<endl;
            cout<<" The result of Multiplying  "<<firstNumber<<
                    "  and  "<<secondNumber<<"   is =   "<< opResult;
}
```

```
void Division(int firstNumber, int secondNumber) {
            int   opResult;
            cout<<endl;
            opResult = firstNumber / secondNumber;
            cout<<endl;
            cout<<" The result of Dividing  "<<firstNumber<<
                    "  by  "<<secondNumber<<"   is =   "<< opResult;
}

void Addition(int firstNumber, int secondNumber) {
            int   opResult;
            opResult = firstNumber + secondNumber;
            cout<<endl;
            cout<<" The result of Adding  "<<firstNumber<<
                    "  and  "<<secondNumber<<"   is =   "<<opResult;
}

void Subtraction(int firstNumber, int secondNumber) {

            int    opResult;
            cout<<endl;
            opResult = firstNumber - secondNumber;
            cout<<endl;
            cout<<" The result of Subtracting  "<<firstNumber<<
                    "  from  "<<secondNumber<<"   is =   "<< opResult;
            cout<<endl;
}
```

Sample Run

```
Enter Two Numbers    3 5

Enter The + - * /                *


 The result of Multiplying  3  and  5   is =   15
```

What is the output of the following ?

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

void function(double , int ) ;   //  Function Prototype

int main()
{
        int x = 60;
        double y = 1.5;
        cout << x  << " "  <<  y  << endl  << endl  ;
        function( y , x ) ;
        cout << x  << " "  <<  y  << endl  << endl  ;
        function( x , y ) ;
  return 0;

}


void function( double a , int b )           // Function Definition
{
        cout << a  << " "  <<  b  << endl ;
        a = 50.50 ;
        b = 10;
        cout << a  << " "  <<  b  << endl  << endl  ;

}
```

The **return** statement and Returning a value from a function

- You can use the return statement to send a value back to the function call.

```
return expr;
```

- The value of the expr will be sent back.
- The data type of the value the function is returning is required in the function header:

Return type:

```
int doubleIt(int x) {
     return x*2;
}
```

- If the function returns void, the function call is a Statement
- If the function returns a value, the function call is an Expression

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

int doubleIt(int x) ;


int main()
{      cout<<  " Double it    "<< doubleIt (2)<< endl;

  return 0;

}



int doubleIt(int x) {
      return x*2;
}
```

Calculator Example Revisited

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


int Addition( int , int ) ;
int Subtraction(int , int ) ;
int Division( int , int ) ;
int Multiplication( int , int ) ;


int Addition(int firstNumber, int secondNumber) {
            int   opResult;
            opResult = firstNumber + secondNumber;
            return  opResult;
}

int Subtraction(int firstNumber, int secondNumber) {

            int    opResult;
            opResult = firstNumber - secondNumber;
            return  opResult;
}

int Division(int firstNumber, int secondNumber) {
            int   opResult;
            opResult = firstNumber / secondNumber;
            return  opResult;
}

int Multiplication(int firstNumber, int secondNumber) {
            int    opResult;

            opResult = firstNumber * secondNumber;
            return  opResult;
}

int main()
{
      int firstNumber, secondNumber, opResult;
      char op;
      cout<<"Enter Two Numbers     ";
      cin>>firstNumber>>secondNumber;
      cout<<endl;
      cout<<"Enter The + - * / ";
      cin>>op;

      switch (op){
      case '*' :  opResult = Multiplication(firstNumber , secondNumber);
                  cout<<endl;
                  cout<<" The result of Multiplication is    "<<opResult;
                  break;
      case '/' :  opResult = Division(firstNumber , secondNumber);
                  cout<<endl;
                  cout<<" The result of Division is    "<<opResult;
                  break;
      case '+' :  opResult = Addition(firstNumber , secondNumber);
                  cout<<endl;
                  cout<<" The result of Addition  is    "<<opResult;
                  break;
      case '-' :  opResult = Subtraction(firstNumber , secondNumber);
                  cout<<" The result of Subtraction   is    "<<opResult;
                  cout<<endl;
                  break;
      default : cout<<" Your Entered a Wrong Operation Symbol";
      }

    return 0;        }
```

Returning a Boolean value

```
bool isValid(int number)    {
       bool status;
       if (number >=1 && number <= 100)
       status = true;
       else
       status = false;
       return status;
 }
```

The above function is equivalent to this one:

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

 bool isValid( int );  // Function Prototype

 int main() {

       int val;
       cout << "Enter a value between 1 and 100: ";
       cin >> val;
       while (!isValid(val)) {
             cout << "That value was not in range.\n";
             cout << "Enter a value between 1 and 100: ";
             cin >> val;
       }

       cout << "You Entered   "<<  val;

 return 0;     }



// Function Definition

 bool isValid (int number) {
       return (number >=1 && number <= 100);           }
```

Sample Run

```
Enter a value between 1 and 100: 300
That value was not in range.
Enter a value between 1 and 100: -200
That value was not in range.
Enter a value between 1 and 100: 20
You Entered   20
Process returned 0 (0x0)   execution time : 7.715 s
Press any key to continue.
```

Example: calling a function more than once

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

void pluses( double ) ;   //  Function Prototype

int main() {
      int x = 2;
            pluses(4);
            pluses(x);
            pluses(x+5);
            pluses(pow(x,3.0));

      return 0;
}



void pluses(double count) {   // Function Definition
      for (int i = 0; i < count; i++)
                   cout << "+";
      cout << endl;
}
```

Output

```
++++
++
+++++++
++++++++
```

Example: function calls another function

```
void deeper() {

         cout << "I am now in function deeper.\n"; }

void deep() {
            cout << "Hello from the function deep.\n";
            deeper();
            cout << "Back in function deep.\n";    }

int main() {

      cout << "Hello from Main.\n";
      deep();
      cout << "Back in function Main again.\n";

return 0;
}
```

Output:

```
Hello from Main.
Hello from the function deep.
I am now in function deeper.
Back in function deep.
Back in function Main again.
```

Passing Arguments by Value

- Pass by value: when an argument is passed to a function, its value is ***copied*** into the parameter.

- Parameter passing is implemented using variable initialization:

```
int param = argument;
```

- Changes to the parameter in the function **do not** affect the value of the argument

Example:

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

// Function   Prototype

void changeMe(int);

int main() {
      int number = 12;
      cout << "The variable number in main  is   " << number << endl;
      changeMe(number);
      cout << "Back in main, the variable number is " << number << endl;
return 0;
}

// Function Definition

void changeMe(int myValue) {
myValue = 200;
cout << "myValue is " << myValue << endl;
}
```

Initialize myValue with number **>>>** same as
Value of number is copied into myValue **>>>** same as
int myValue = number;

Output

```
The variable number in main  is  12
myValue is 200
Back in main, the variable number is 12
```

- Parameter is initialized to a copy of the argument's value.
- Even if the body of the function changes the parameter, the argument in the calling function is unchanged.
- The parameter and the argument are stored in **separate** variables, separate locations in memory.

What is the output of the following ?

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

void foo( int ) ;

int main()
{
   int x = 5;
   cout << "x = " << x << endl;

   foo(x);

   cout << "x = " << x << endl;
   return 0;
}

void foo( int y)
{
   cout << "y = " << y << endl;

   y = 6;

   cout << "y = " << y << endl;
}
```

Passing Arguments by Reference

- Pass by reference: when an argument is passed to a function, the function **has direct access** to the original argument.

- Pass by reference in C++ is implemented using a reference parameter, which has an **ampersand (&)** in front of it:

```
void changeMe (int &myValue);
```

- A reference parameter acts as an alias to its argument.

- Changes to the parameter in the function **DO** affect the value of the argument

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

void changeMe(int &);

int main() {
      int number = 12;
            cout << "number is " << number << endl;
            changeMe(number);
            cout << "Back in main, number is " << number << endl;
      return 0;
}

void changeMe(int &myValue) {
      myValue = 200;
      cout << "myValue is " << myValue << endl;
}
```

Output

```
number in main  is 12
myValue is 200
Back in main, number is 200
```

Another Example : Using Pass by Reference for input

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

double square(double number ) ;    // square Function  Prototype
void getRadius(double &rad) ;    // getRadius Function Prototype

int main() {
      const double PI = 3.14159;
      double radius;
      double area;
      cout << fixed << setprecision(2);
      getRadius(radius);
      area = PI * square(radius);
      cout << "The area is " << area << endl;
return 0;
}

// getRadius Function Definition

void getRadius(double &rad) {
 cout << "Enter the radius of the circle: ";
 cin >> rad;
}

// square Function Definition

double square(double number ) {
      return number * number;
}
```

During the function execution, **rad** is an **alias** (alternative name) to **radius** in the main program.

Pass by Reference

- Changes to a reference parameter are actually made to its argument
- The **&** must be in the function header AND the function prototype.
- The argument passed to a reference parameter must be a variable – it cannot be an expression or constant
- Use when appropriate – don't use when
    - argument should not be changed by function
    - function needs to return only 1 value

Return multiple values from a function :-

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

 // Function  Prototype

 void tester( double & , double & );

 // main function

 int main()
 {
     double x = 0.0, y = 0.0;

     // init variables

     x = 5.6;
     y = 10.25;

     // print original values

     cout<<"Original Values of x and y  "<< x<< "       "<<y<<endl;

     // call tester passing the address rather than the values

     tester( x, y );

     // print new values gotten from function

     cout<< "New Values of x and y       "<<x<<"    "<<y<<endl;

     return 0;

 } // end of main


 // notice the function is void as it returns nothing
 // you passed in the address of the variables


void tester( double &x, double &y )  {

   // pointers are a pointer to an address
   // these statements are basically saying that the address contains the
   // following values

     x = 900.23;
     y = x * 50.3 + 100.00;   }
```

Output

```
Original Values of x and y  5.6       10.25
New Values of x and y       900.23    45381.6
```

Overloading Functions

- **Overloaded functions** have the same name but different parameter lists.
- Used to create functions that perform the same task over different sets of arguments.
- The parameter lists of each overloaded function must have different types and/or number of parameters.
- The compiler will determine which version of the function to call based on arguments and parameter lists

Example: Overloaded function prototypes

- Different number of arguments:

```
double sum (int exam1, int exam 2);
double sum (int exam1, int exam 2  , int exam 3);
double sum (int exam1, int exam 2 , int exam 3 , int exam 4);
```

```
/*
 *      FunOverLoad.cpp
 *
 *      Author: Husain Gholoom
 */

#include<iostream>
using namespace std;

void calc(int num1);                      // Function Prototype
void calc(int num1, int num2 );           // Function Prototype
void calc(double  num1);                  // Function Prototype
void calc(double  num1, double num2 );    // Function Prototype

int main()          //begin of main function
{
          calc(5);
          calc(5.2);
          calc(6,7);
          calc('A');
          calc(12.0,5.0);

      return 0;     }
// Function Definition

void calc(int num1)

      {   cout<<"Square of a given number: " <<num1*num1 <<endl;   }


void calc(int num1, int num2 )

      {   cout<<"Product of two whole numbers: " <<num1*num2 <<endl;    }

void calc(double  num1)

      {   cout<<"Square of a given number: " <<num1*num1 <<endl;   }


void calc(double num1, double num2 )

      {   cout<<"Quotient   of two whole numbers: " <<num1/num2 <<endl; }
```

Sample Run

```
Square of a given number: 25
Square of a given number: 27.04
Product of two whole numbers: 42
Square of a given number: 4225
Quotient   of two whole numbers: 2.4
```

Default Arguments

- A default argument is a value passed to the parameter when the argument is left out of the function call.

- The default argument is usually listed in the function prototype:

```
int showArea (double = 20.0, double = 10.0);
```

- Default arguments are literals (or constants) with an = in front of them, occurring after the data types listed in a function prototype

```
void showArea (double = 20.0, double = 10.0);
...
void showArea (double length, double width) {
      double area = length * width;
      cout << “The area is ‟ << area << endl;
}
```

This function can be called as follows:

```
showArea(); ==> uses 20.0 and 10.0
The area is 200

showArea(5.5,2.0); ==> uses 5.5 and 2.0
The area is 11

showArea(12.0); ==> uses 12.0 and 10.0
The area is 120
```

Example: Default Arguments

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

void displayStars(int = 10, int = 1);

int main () {

      displayStars(); // uses 10 x 1
      cout << endl;
      displayStars(5); // uses 5 x 1
      cout << endl;
      displayStars(7, 3); // uses 7 x 3

return 0;

}

void displayStars(int cols, int rows) {
      for (int down = 0; down < rows; down++) {
            for (int across = 0; across < cols; across++)
                  cout << "*";
            cout << endl;      }
}
```

Sample Run

```
**********

*****

*******
*******
*******
```

Note :

- When an argument is left out of **a function call**, all arguments that come after it must be left out as well.

```
displayStars(5);     // uses 5 x 1
displayStars( ,7);  // NO, won't work for 10 x 7
```

- If not all parameters to a function have default values in the **prototype**, the parameters with defaults must come last:

```
int showArea (double = 20.0, double); // NO
int showArea (double, double = 20.0); // OK
```

Default Arguments

- Default arguments are like overloaded functions

```
void displayStars(int = 10, int = 1);
```

- Is like declaring 3 overloaded functions:

```
void displayStars();          // uses 10 and 1
void displayStars(int);       // uses arg and 1
void displayStars(int, int);  // uses arg1 and arg2
```

The exit() Function

The eixt() function causes a program to terminate , regardless of which function or control mechanism is executing.

#include<cstdlib> might be required in some IDE's.

Example

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

void function( ) ;   //  Function Prototype

int main()
{
        function( ) ;

        cout << "This message will also never be displayed  ." << endl ;
        cout <<"because the program has already terminated.\n";

   return 0;   }

void function()                 // Function Definition
{
        cout << "This program terminates with the exit function ." << endl ;
        cout <<"Byte!!!\n";

        exit(0);

        cout << "This message will never be displayed  ." << endl ;
        cout <<"because the program has already terminated.\n";   }
```

Sample Run

```
This program terminates with the exit function .
Byte!!!
```

Variable Definitions and Scope

- The scope of a variable is the part of the program where the variable may be used.
- For a variable defined inside a function, its scope is the function, from the point of definition to the end of the function.
- For a variable defined inside of a block, its scope is the innermost block in which it is defined, from the point of definition to the end of that block.

Variables in functions and blocks

```
 *   FunAndVarScop.cpp
 *
 *   Author: Husain Gholoom
 */


#include <iostream>
#include<iomanip>
using namespace std;
int main()
{

      double income; //scope of income is red + blue
      cout << "What is your annual income? ";
      cin >> income;
      if (income >= 35000) {
            int years; //scope of years is blue;
            cout << "How many years at current job? ";
            cin >> years;
            if (years > 5)
                  cout << "You qualify.\n";
            else
                  cout << "You do not qualify.\n";
            }
      else
      cout << "You do not qualify.\n";
      cout << "Thanks for applying.\n";

      return 0;

}
```

Cannot Access **years** down here

Variables with the same name

- In an inner block, a variable can have the same name as a variable in the outer block.
- When in the inner block, the outer definition is not available (it is hidden).
- Not good style: difficult to trace code and find bugs

```
/*   FunAndVarScop.cpp
 *
 *   Author: Husain Gholoom
 */

#include <iostream>
#include<iomanip>
using namespace std;
int main()
{
      int number;
      cout << "Enter a number greater than 0: ";
      cin >> number;
      if (number > 0) {
            int number; // another variable named number
            cout << "Now enter another number ";
            cin >> number;
            cout << "The second number you entered was ";
            cout << number << endl;
      }
      cout << "Your first number was " << number << endl;

return 0;

}
```

Sample Run

```
Enter a number greater than 0: 10
Now enter another number 15
The second number you entered was 15
Your first number was 10
```

Local and Global Variables

- Variables defined inside a function are **local** to that function.
  - They are hidden from the statements in **other** functions, which cannot access them.
- Because the variables defined in a function are hidden, other functions may have separate, distinct variables with the same name.

  - This is not bad style. These are easy to keep straight.

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

void anotherFunction();    // Function Prototype

int main() {
int num = 1;
cout << "In main, num is " << num << endl;

anotherFunction();

cout << "Back in main, num is " << num << endl;

return 0;

}

//    Function Definition


void anotherFunction() {

int num = 20;

cout << "In anotherFunction, num is " << num << endl;
     }
```

Sample Run

```
In main, num is 1
In anotherFunction, num is 20
Back in main, num is 1
```

- When the program is executing main, the num variable defined in main is visible.
- When anotherFunction is called, only variables defined inside that function are visible, so the num variable in main is hidden.

Local Variable Lifetime

- Parameters have the same scope as local variables in the function.
- When the function begins, its parameters and local variables (as their definitions are encountered) are created in memory, and when the function ends, the parameters and local variables are destroyed.

Global Variables

- A global variable is any variable defined outside all the functions in a program.
- The scope of a global variable is the portion of the program from the variable definition to the end.
- This means that **a global variable can be accessed by all functions** that are defined after the global variable is defined

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

void anotherFunction();    // Function Prototype


int num = 2;

int main() {

   cout << "In main, num is " << num << endl;
   anotherFunction();
   cout << "Back in main, num is " << num << endl;
return 0;

}

//    Function Definition

void anotherFunction() {

    cout << "In anotherFunction, num is " << num << endl;
    num = 100;
    cout << "Still in anotherFunction, num is " << num << endl;


     }
```

Sample Run

```
In main, num is 2
In anotherFunction, num is 2
Still in anotherFunction, num is 100
Back in main, num is 100
```

Notes :

- You should avoid using global variables because:
  - They make programs difficult to debug.
  - If the wrong value is stored in a global var, you have to find every place in the whole program where the value is changed
- Functions that access globals are not self-contained
  - cannot easily reuse the function in another program.
  - cannot understand the function without understanding how the global is used everywhere

Global Constants

- **It is ok to use global constants because their values do not change.**

Example

```
double getArea(double);       // Function Prototype
double getPerimeter(double);  // Function Prototype

const double PI = 3.14159;     // PI is Global Constant

int main() {
double radius;
cout << fixed << setprecision(2);
cout << "Enter the radius of the circle: ";
cin >> radius;
cout << "The area is " << getArea(radius) << endl;
cout << "The perimeter is " << getPerimeter(radius) << endl;
return 0;
}

// Function Definition

double getArea(double number) {
     return PI * number * number;
}
double getPerimeter(double number) {
     return PI * 2 * number;
}
```

Sample Run

```
Enter the radius of the circle: 2
The area is 12.57
The perimeter is 12.57
```

Accessing a global variable

Example of accessing a global variable using scope resolution operator **::** when there is a local variable with same name

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

// Global x

int x = 0;

int main() {
        // Local x
        int x = 10;
        cout << "Value of global x is " << ::x;
        cout << "\nValue of local x is " << x;
        return 0;
}
```

Sample run

```
Value of global x is 0
Value of local x is 10
```

What is the output of the following?

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

void myFunction();          // prototype

int x = 5, y = 7;

int main() {

        cout << "x from main: " << x << "\n";
        cout << "y from main: " << y << "\n\n";
        myFunction();
        cout << "Back from myFunction!\n\n";
        cout << "x from main: " << x << "\n";
        cout << "y from main: " << y << "\n";
        return 0;
}

void myFunction() {
        int y = 10;

        cout << "x from myFunction: " << x << "\n";
        cout << "y from myFunction: " << y << "\n";
        cout << "again y from myFunction: " << ::y << "\n\n";


}
```

Scope Rules Summary

- Variable scope: to end of the block it's defined in.
- Variables cannot have same name in same exact scope.
    - A variable defined in inner block can hide a variable with the same name from outer block.
- Variables defined in one function cannot be seen from another.
- Parameter scope: the body of the function
    - cannot have function variable same name as parameter
- Variable lifetime: variables are destroyed at the end of their scope
- Global variable/constant scope: to end of entire program
    - variables defined inside a function are called Local

File Streams & Functions :

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


//  The function  is to read integers  from a file, Output the integers  to an output file and screen ,
//  Add  the numbers to an integer variable Sum .
//  At the end , write the Sum  to an output file and screen

void read(ifstream &T , ofstream &O)    // pass the file stream to the function
{
        int  No; int sum = 0;
        T >> No;

        while(T)
        {
                cout << No << "    ";
                O <<  No << "   ";
                sum += No;
                T >> No;
        }

        cout << "\n\n----------------" << endl;
        O       << "\n\n--------------" << endl;

        cout << "\n\nSum   =   " << sum << endl;
        O << "\n\nSum   =    " <<   sum << endl;
}

int main()

{
        ifstream T;
        T.open ("file1");
        if ( !T )
        {
                cout << endl << endl
                  << "***Program Terminated.***" << endl << endl
                  << "Input file failed to open." << endl;

                T.close();

          return 1;

        }  // Quit, but don't return a 0; send back a non-zero value.


        ofstream  O;
        O.open ("out");

        read(T , O);

        T.close();
        O.close();

        return 0;

}
```

Practice

What is the output of the following code?

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

int showVolume(int length, int  = 1, int  = 1);

int main()
{
	int vola,volb,volc;
	vola=showVolume(4, 6, 2);
	cout<<vola<<endl;
	volb=showVolume(4, 6);
	cout<<volb<<endl;
	volc=showVolume(4);
	cout<<volc<<endl;

	return 0;
}

int showVolume(int length, int width, int height)
{
	int volume;
	volume=length*width*height;
	return volume;
}
```

What is the output of the following program ?

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


void test(int = 2 , int = 4 , int = 6);

int main() {
        test();
        test(6);
        test(3, 9);
        test(1, 5, 7);
        return 0;
}
void test(int first, int second, int third) {
        first += 3;
        second += 6;
        third += 9;
        cout << first << " " << second << " " << third << endl;
}
```

What is the output of the following program ?

```
#include <iostream>
using namespace std;
int manip(int);
int manip(int, int);
int manip(int, double);
int main() {
        int x = 2, y = 4, z;
        double a = 3.1;
        z = manip(x) + manip(x, y) + manip(y, a);
        cout << z << endl;
        return 0;
}
int manip(int val) {
        return val + val * 2;
}
int manip(int val1, int val2) {
        return (val1 + val2) * 2;
}
int manip(int val1, double val2) {
        return val1 * static_cast<int>(val2);
}
```

What is the output of the following program ?

```
#include <iostream>
using namespace std;
void func1(double, int); // Function prototype
int main() {
        int x = 0;
        double y = 1.5;
        cout << x << "   " << y << endl;
        func1(y, x);
        cout << x << "   " << y << endl;
        return 0;
}
void func1(double a, int b) {
        cout << a << "   " << b << endl;
        a = 0.0;
        b = 10;
        cout << a << "   " << b << endl;
}
```

What is the output of the following program ?

```
#include <iostream>
using namespace std;
void func1(double,   int & ); // Function prototype
int main() {
        int x = 0;
        double y = 1.5;
        cout << x << "\t" << y << endl;
        func1(y, x);
        cout << x << "\t" << y << endl;
        return 0;
}
void func1(double  a,  int  & b) {
        cout << a << "\t" << b << endl;
        a = 0.0;
        b = 10;
        cout << a << "\t" << b << endl;
}
```