Chapter 6 - Functions

- return type: void or a valid data type (int, double, char, etc)
- name
- parameter list: void or a list of parameters separated by commas
- body
- return keyword: required if function returns a value; optional if it does not

```c
int square ( int x )
{
    return x * x;
}
```

```c
void ErrorMsg ( )
{
    cout << "An error occurred. The program is exiting...
    exit( EXIT_FAILURE );
}
```

- parameters are the names of the local variables defined in the function header.
- arguments are the names of the variables/expressions used in the function call (invocation).

```c
int square ( int );
```

- calling a function - function-name(argument list);

```c
void main ( )
{
    ...
    y = square ( x );
    ...
}
```

- you may optionally include parameter variable names in the prototype

```c
void ShowSum ( int x, int y, int z);
```

- you must place either the function definition or prototype ahead of all calls to the function

- call by value - a copy of an argument is passed to the calling function

```c
z = 10;
y = square ( z );
```

- return statement

```c
int Square ( int x )
{
    return x * x; // return the square of x.
}
```
local and global variables

- a **local variable** is declared inside a function and is local to that function
- a **global variable** is declared outside of all functions. The scope of a global variable is the point of declaration to the end of the program

  - local variables have the **automatic** storage class
  - automatically created and destroyed – not automatically initialized
  - global variables have the **static** storage class
  - they exist for the life of the program – if not explicitly initialized, they are implicitly initialized to 0

local and global variables with the same name - unary scope resolution operator ( ::)

In the absence of the unary scope resolution operator, a local variable with the same name as a global variable “hides” the global variable.

```cpp
int Number = 100;
void main ( )
{
    int Number = 10;
    cout << “Number in main: “ << Number << “, global Number: “ << ::Number;
}
Displays: Number in main: 10, global Number: 100
```

**static local variables** [ variables declared static within a function ]

- use the storage class qualifier **static**
- static int x;

  - static storage class
  - if not explicitly initialized, implicitly initialized to 0
  - initialized only once
  - persistent between function calls ( retain their values )

default arguments

- passed to parameters automatically if no argument is provided in the function call
- constant values with an = operator in front of them
- specified in the function prototype if present, otherwise in the function header
- when an argument is left out of a function call, all arguments that come after it must be left out as well

```cpp
double area ( double=0, double=0 );
void main ( )
{
    double length = 5.5, width = 7.5;
    cout << “ the area of the rectangle is: “ << area ( );
    cout << “ the area of the rectangle is: “ << area(length);
    cout << “ the area of the rectangle is: “ << area(length, width);
}
```

- in the function definition, default arguments must be specified after all required parameters

```cpp
int Strange ( int, int, int=0, int=0);
function has 2 required parameters and 3 optional parameters
```
reference parameters
• reference parameters allow a function to access the parameter’s original argument. Changes made to the parameter are also made to the original argument
• the & operator is used to declare a reference parameter

```c
void square ( int &x ); // call by reference using a reference

void main ()
{
    int x = 5;
    cout << “Print the value of x is: “ << x << endl;
    square ( x );
    cout << “Print the square of x is: “ << x << endl;
}
```

do not allow this overload

```c
void square ( int &y )
{
    y *= y;
}
```

overloaded functions
• functions with the same name that have different parameter lists
  ```c
  int square ( int );
  float square ( float );
  ```
• function with the same name may have the same return type but MUST have different parameter lists

```c
exit ( ) function
• causes a program to terminate regardless of which function is currently executing
• to use, you must include the header file cstdlib
• requires an integer value as an argument
• can be used with the symbolic constants EXIT_SUCCESS and EXIT_FAILURE

exit ( EXIT_SUCCESS ); exit ( EXIT_FAILURE );

Chapter 7 - Arrays
An array is an aggregate data type. It can store a group of values, all of the same type. Arrays are declared similar to other variables, except that they must have a size declarator or an initializer list that is used to determine how many elements to allocate space for. The size declarator must be a constant integer expression (integer literal or integer named constant) with a value greater than 0.

Example declarations:
```c
int Scores[50];  // define an array of 50 integers using a constant literal

const int size = 50;  // define a named integer constant
int Scores[size];  // use named integer constant as a size declarator
```

Arrays are like any automatic variables. If they are not initialized, the elements contain garbage.

When referencing array elements, you use a subscript enclosed in square brackets. Subscript numbers are in the range of 0 to array size – 1 because in C++ you start counting at 0! (zero-based addressing).

```c
int X[10];  // 10 element array of integers; subscripts are 0 .. 9
```

|------|------|------|------|------|------|------|------|------|------|

To work with a single array element, use the array name and a subscript value. The value can be an integer variable or expression.

```cpp
int temps[5];
int k=3;

temps[3] = 81;  or  temps[k] = 81;
```

C++ does not do any bounds checking. If array Count is declared to have 10 elements, you can refer to Count[10] or Count[-1] - both of which are outside of the bounds of the array.

Arrays can be initialized when they are declared using an initializer list.

```cpp
int Odds[5] = { 1, 3, 5, 7, 9 };  
```

The initializer list may specify fewer than the number of elements in the array. You may not skip elements in the initializer list. Uninitialized elements are initialized to 0.

```cpp
int Odds[5] = { 1, 3 };  // elements 2, 3, and 4 are initialized to 0.
```

Arrays can be implicitly sized by leaving out the array size declarator and specifying an initialization list.

```cpp
float ratings[] = {1.5, 2.0, 2.5, 3.0};
char name[] = "Gadden";  // allocates an array of 7 character elements (one element is for the NULL terminator).
```

You cannot use the assignment operator to copy one array’s contents to another.

```cpp
int x[3] = { 1, 3, 5 };
int y[3] = { 2, 4, 6 };  

x = y;  // NO!, can’t do this…

instead, use:
for (int n = 0 ; n < 3 ; n++ )
    x[n] = y[n];
```

The contents of an array may be printed. To do this, use a loop. The only exception is printing a character array that holds a C-string. You can do this using cout;

```cpp
int x[3] = { 1, 3, 5 };
int y[3] = { 2, 4, 6 };

x = y;  // NO!, can’t do this…

instead, use:
for (int n = 0 ; n < 3 ; n++ )
    x[n] = y[n];
```

A single element in an array can be passed to a function, or an entire array may be passed to a function.

The default passing mechanism for passing an element is by value. Entire arrays are not actually passed, only the address of the array. Array names are passed by reference.

```cpp
// function definition:
int square ( int x )
{
    return x * x;
}

// function call:
y = square ( NumArray[2] );
```

A two-dimensional array is like several identical arrays put together. It is useful for storing multiple sets of data.

```cpp
float Array[2][3];  // defines an array with 2 rows and 3 columns [2 - 3 element arrays]
```

<table>
<thead>
<tr>
<th>Row 0</th>
<th>Column 0</th>
<th>Column 1</th>
<th>Column 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array[0][0]</td>
<td>Array[0][1]</td>
<td>Array[0][2]</td>
<td></td>
</tr>
<tr>
<td>Row 1</td>
<td>Column 0</td>
<td>Column 1</td>
<td>Column 2</td>
</tr>
<tr>
<td>Array[1][0]</td>
<td>Array[1][1]</td>
<td>Array[1][2]</td>
<td></td>
</tr>
</tbody>
</table>
When initializing a two-dimensional array, it is helpful to enclose each row’s initialization list in a set of braces.

```c
int Hours [3][2] = { {8, 5},   {7, 9},   {6, 3} };
    row 0  row 1  row 2
```

Passing a multi-dimensional array to a function

In the function prototype and function header, **constant integer size declarators must be specified for all but the first dimension.**

```c
int ShowArray( int [ ][3], int rows);     // prototype for a function that will be passed a 2-dimensional array
ShowArray( Sales, 4);       // function call that passes ShowArray array Sales, a two dimensional array
    // with 4 rows and 3 columns.
```

---

## Chapter 8 - Searching and Sorting

### Searching

Search through an array of elements looking for an exact match.

#### Linear Search

Search through an array of sorted or unsorted elements looking for an exact match. If a match is found, return the subscript value for that element, otherwise return –1 to indicate no match was found. If the size of the array is N, the linear search does at most N comparisons. On average, it does N/2 comparisons. The minimum number of comparisons necessary is 1.

#### Binary Search

Can only be used on SORTED arrays. If a match is found, return the subscript value for that element, otherwise return –1 to indicate no match was found. Powers of two are used to calculate the maximum number of comparisons the binary search will make on an array of any size. The minimum number of comparisons it will make is 1.

To find the maximum number of comparisons that may be made:

- Find the smallest value of \( n \) such that \( 2^n > \) the number of elements in the array.
- If the array size is 25: \( 2^4 < 25 < 2^5 \)
- \( n = 5 \), therefore the maximum number of comparisons which may be necessary is 5.

### Sorting

Two sorting algorithms – Bubble Sort and Selection Sort were presented. Both are designed to sort the elements in an array in ascending or descending sequence.

Bubble Sort is the simplest sort algorithm. Bubble sort performs a series of passes. During each pass, exchanges (swaps) are performed when adjacent elements are found to be out of sequence. Passes are repeated until a pass occurs where there are no swaps necessary.

Selection Sort is a bit more complex, but is also more efficient. Selection sort only does one exchange per pass. It puts the current value in its correct position in the array.
Address operator ( & )

Every variable (or array element) has an address in memory. To retrieve the address of a variable, precede the variable name with the address operator ( & ).

```c
int x, int y[5];
    &x .......... returns the address of variable x
    &y[3]..... returns the address of the 4th element in array y.
```

Note: The address operator is NOT used with an array name when no element is being referenced. 
    y............. returns the address of the first element in the array. Equivalent to &y[0]

Pointer variables ( also simply referred to as pointers )

Pointer variables are used to indirectly manipulate the data stored in other variables.

The ( * ) is used in the declaration of a pointer variable. * is called the indirection operator.

```c
int *x, *y; // declares two pointer variables, x and y. The * is necessary to declare these as pointers
            // and not simply as integer variables.
```

The ( * ) is also used as an operator to dereference a pointer ( indirectly reference the value of the variable the pointer points to ). It is called the indirection operator or dereference operator.

```c
int x, y, *xPtr, *yPtr; // declares two integer variables and two variables that are pointer to integer
                        xPtr = &x;  yPtr = &y; // initializes the values of the pointer variables ( assigns addresses )
int x, y, *xPtr = &x, *yPtr = &y; // declare and initialize pointer variables in one statement
int x = 40, *ptr = &x;
    cout <<     ptr << endl;   // would print the value of the pointer variable ( an address )
    cout << &ptr << endl;     // would print the address of the variable ptr
    cout << *ptr << endl;     // would print the value of the variable ptr points to
```

Pointer variables must be declared to “point” to a specific type of variable (int, double, float, char, etc…)
A pointer to one type may not be assigned the address of a variable of a different type.

```c
int x; float y; int *ptr = &y; // this is illegal since ptr is a pointer to an integer.
```

Array names may be used as pointers and vice-versa, however, an array name is a constant pointer and cannot be changed.

(1) int x[5], *IntPtr = x; // assigns the address of the 1st element in array x to IntPtr.
(2) int x[5], *IntPtr = &x[0]; // this is an equivalent declaration using a subscripted array element

Two notations for referring to the value of an array element when there is a pointer to the array.

<table>
<thead>
<tr>
<th>Subscript notation</th>
<th>Offset notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>x [1]</td>
<td>*(x + 1)</td>
</tr>
<tr>
<td>intPtr[1]</td>
<td>*(intPtr + 1)</td>
</tr>
</tbody>
</table>

When you add a value to a pointer, you are actually adding that value times the size of the data type referenced by the pointer. This is called "pointer arithmetic".

Note: The * operator has precedence over the + operator so *x + 1 is different than *(x + 1).
Array names

An array name is a constant pointer. An array name cannot be assigned an address of another array. The "value" of an array name is the address of the first element in the array.

Given these declarations:  int x[10], y[10], *xPtr = x, *yPtr = y;
We could assign different values for the pointers:
  xPtr = y;  yPtr = x;  // change the arrays the pointers point to...
We could not do the same operations using the array names!:
  x = y;  // NO! an array name is a constant pointer.

Given this declaration:  int count[20];  These two expressions are equivalent:  count and  &count[0]

Pointer arithmetic

Mathematical operations
The following operators may be used with pointer variables:
  add (+),  subtract (-),  add/assign (+=),  subtract/assign (-=),
  increment (++),  decrement (--)  

If Ptr is a pointer variable, then expressions such as:  Ptr + 3,  Ptr -2,  Ptr++  are all examples of pointer arithmetic. The result of pointer arithmetic is the calculation of an address. If Ptr was defined as a pointer to int,  Ptr +3 would get the address in Ptr and add ( 3 * the size in bytes of an integer ).

Initializing Pointers

A pointer is designed to point to an object of a specific data type.
A pointer may be initialized to the values:  0,  NULL (symbolic constant with the value 0) or to the address of an object of the type the pointer can point to.

Relational operations may be performed on pointer variables of the same type

All of the relational comparisons are valid on pointers of the same type.  >, <=, >=, ==, !=

int x, y, *xPtr = &x, *yPtr = &y;

Examples  :xPtr = yPtr   xPtr > yPtr   xPtr >= yPtr   yPtr > xPtr   yPtr != xPtr, etc.

Pointers as function arguments.

Pointers can be used to pass arguments by reference to a function

void double ( int *val );  // prototype for a function that expects to be passed an address
Note:  The words pointer and address are used as synonyms. The "value" of a pointer is an address…
We would say this is  call by reference using a pointer ( rather than call by reference using a reference ).

Given the declaration  int x=7;  in a function.  A call to double using x would be:  double ( &x );

Function double must be passed an address to assign to its pointer variable.

void square ( int & );
  // prototype for a function that gets an argument passed by reference using a reference
  // given x is an integer variable in a calling function, the function call to pass x would be  square(x);
void square ( int * );
  // prototype for a function that gets an argument passed by reference using a pointer
  // given x is an integer variable in a calling function, the function call to pass x would be  square(&x);
Dynamic memory allocation

For a function that declares an array, the size declarator must be an integer constant expression. The compiler must be able to determine (at compile time) how much memory must be allocated at runtime to hold the array.

When it is not known how many elements storage is needed for and there could be potentially a sizeable difference between different executions of a program, memory can be allocated dynamically. This allows for the allocation of an arbitrary amount of memory at runtime.

Memory can also be allocated for a single element. The new and delete operators are used to allocate and release dynamic memory. These operators require the use of pointers.

Memory is dynamically allocated using the **new** operator and a pointer variable.

```cpp
int *p = new int;       // dynamically allocate space for a single integer
int *p = new int[100];  // dynamically allocate space for 100 integers
```

Both of these declarations assign to `p` the address of the first (or only) element.

```cpp
int *p = new int[Count];
Dynamically allocate space for count elements. Count does NOT have to be declared as a named constant. Also, the value of count does NOT need to be known at compile time!
cout << "For how many students do you have test scores? ";
cin >> count;    int *p = new int[count];
delete p;       // release the memory previously dynamically allocated for a single element
delete [ ] p;   // release the memory previously dynamically allocated for a group of elements
```

In a declaration such as `int *p = new int[100];` `p` can be used to refer to the elements using subscript notation. `p[0], p[10], p[99],` etc.