



Chapter 7

Functions



Types of Functions

- Value returning
 - Functions that return a value through the use of a return statement
 - They allow statements such as this:
 - `Y = 3.8 * sqrt(x);`
- Void Functions
 - They do not return anything through their return statement
 - `cin.get(someChar);`



When do you create Functions?

- One simple heuristic to use
 - If the overall program is easier to understand as a result of creating a function, then the function should be created.



Void Functions

```
#include <iostream>
void Print2Lines();
void Print4Lines();
int main (){
    Print2Lines();
    std::cout << " Welcome Home!\n";
    Print4Lines();
    return 0;
}
```



More Void

```
void Print2Lines()
{
    std::cout << "*****\n";
    std::cout << "*****\n";
}
```



One More

```
void Print4Lines()
{
    std::cout << "*****\n";
    std::cout << "*****\n";
    std::cout << "*****\n";
    std::cout << "*****\n";
}
```



Flow of Control

- Functions definitions can appear in any order in the source file
- main usually comes first
- Flow of control begins with the first line in main and continues until a function invocation
- Flow is passed to the function
- When the function is done, flow returns to the first statement after the function call



Parameters

- Can the first program be written better?
- How?
- Use a parameter to indicate how many lines to print



Types of Parameters

- Argument
 - A variable or expression listed in a call to a function.
 - Also called *actual argument* or *actual parameter*
- Parameter
 - A variable declared in a function heading.
 - Also called *formal argument* or *formal parameter*



New Welcome program

```
void PrintLines( int );
int main()
{
    PrintLines( 2 );
    std::cout << "Welcome Home!\n";
    PrintLines( 4 );
    return 0;
}
```



New PrintLines

```
void PrintLines( int numLines )
{
    int count = 0;
    while ( count < numLines )
    {
        std::cout << "*****\n";
    }
}
```



Syntax and Semantics

- Function Call: a statement that transfers control to a void function.
- FunctionName (ArgumentList);
- ArgumentList is a comma separated list of values and variables



Declarations and Definitions

- Function prototype:
 - A function declaration without the body of the function
 - Does not allocate memory
- Function definition:
 - A function declaration that includes the body of the function
 - Allocates memory



Local Variables

- Any function you write can also have variables
- Those variables that are inside of a function are called local variables
- They go away when the function ends



Global Variables

- Variables that are declared outside of any functions are called Global variables
- Any function can use global variables
- Global variables should be avoided



Return Statement

```
void SomeFunc( int n )
{
    if ( n > 50 )
    {
        cout << "The value is out of range.";
        return;
    }
    n = 412 * n;
    cout << n;
}
```



Return Again

```
void SomeFunc( int n )
{
    if ( n > 50 )
        cout << "The value is out of range.";
    else
    {
        n = 412 * n;
        cout << n;
    }
}
```



Single-Entry / Single-Exit

- What's the difference in the preceding two examples?
 - The return
 - You can argue the each example is a better approach
 - I prefer the second; It uses Single-entry/single-exit



Parameters

- Two types
 - Value Parameter
 - Simply pass information in to a function
 - Reference Parameter
 - Can pass information in and receive information from a function



Value Parameters

- `void PrintLines (int numLines)`
- When invoke like this:
 - `PrintLines (LineCount)`
 - The function receives a copy of the variable `LineCount`
 - You can put anything in the function call that produces a value
 - `consts`, `expressions`, `variables`



Reference Parameters

- void CountandSum(int List, int &Sum)
- When invoked like: CountandSum(MyList, sum);
 - The function receives the MyList by value and sum by reference.
 - The function is now allowed to inspect and change sum and the calling function will be aware of the changes
 - The & actually gives the function the address of the variable



Designing Functions

- Two Key Ideas in Programming
- Interface
 - A connecting link at a shared boundary that permits independent systems to meet and act on or communicate with each other.
- Encapsulation
 - Hiding a module's implementation in a separate block with a formally specified interface



More on Designing

- Need a list of incoming values, outgoing values, and incoming/outgoing values
- Decide which of the values from the list need to be given to the function
- These values are declared in the function heading with the appropriate passing mechanism
- All others are local variables



Writing Assertions as Comments

```
void PrintAverage ( float sum, int count )
//Precondition:
// sum is assigned && count > 0
//Postcondition:
// The average sum/count has been
// outputted on one line
{
    cout << "Average is " << sum/float (count);
}
```



Flow of Data

- Data Flow
 - The flow of information from the calling code to a function and from the function back to the calling code.
- Two Directions/Three Data Flows
 - In
 - Out
 - In/Out



Data Flow and Passing Mechanism

- | ○ Data Flow for a Parameter | ○ Argument-Passing Mechanism |
|-----------------------------|------------------------------|
| ○ Incoming | ○ Pass-by-value |
| ○ Outgoing | ○ Pass-by-reference |
| ○ Incoming/Outgoing | ○ Pass-by-reference |



Quiz

- What are the two passing mechanisms for passing a variable to a function?
- What does the & return for a variable?
- Why might you want to use pass-by-reference?