04: FUNCTION

Programming Technique I
(SCSJ1013)

Modular Programming

- **Modular programming**: breaking a program up into smaller, manageable functions or modules
- **Function**: a collection of statements to perform a task
- Motivation for modular programming:
  - Improves maintainability of 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

Function

- A **collection of statements** that performs a specific task.
- Commonly used to **break a problem** down into small manageable pieces.
- In C++, there are 2 types of function:
  - Library functions
  - User-defined functions

Library Functions

- "Built-in" functions that **come with the compiler**.
- The source code (definition) for library functions does NOT appear in your program.
- To use a library function, you simply need to **include the proper header file** and know the name of the function that you wish to use.
  - **#include** *compiler directive*

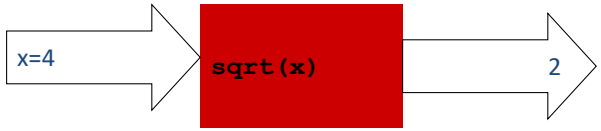
Library Functions (cont.)

- Libraries under discussion at this time:

Compiler directive	Purpose
<ctype>	Character classification and conversion
<cmath>	Math functions
<stdlib>	Data conversion
<time>	Time functions

Library Functions (cont.)

- A collection of specialized functions.
- C++ promotes code reuse with predefined classes and functions in the standard library
- The functions work as a black box:



Math Functions

- Required header: #include <cmath>
- Example functions

Function	Purpose
abs(x)	returns the absolute value of an integer.
pow(x,y)	calculates x to the power of y. If x is negative, y must be an integer. If x is zero, y must be a positive integer.
pow10(x)	calculates 10 to the power of x.
sqrt(x)	calculates the positive square root of x. (x is >=0)

Library Functions: Example 1

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
    double area, radius;

    cout<< "This program calculates the area of a
    circle.\n";
    cout<<"What is the radius of the circle? ";
    cin>>radius;
    area=3.14159 * pow(radius,2.0);
    cout<<"The area is " << area <<endl;
    system ("pause");
    return 0;
}
```


Library Functions: Example 2

```
#include <iostream>
#include <cmath>
using namespace std;
main(){
    int nom1,nom2, result;
    cout<<"Enter two numbers";
    cin>>nom1>>nom2;
    if ((nom1<0)||(nom2<0))
    {   cout<<"negative number/s";
    }
    else
    {
       result= sqrt(nom1 + nom2);
       cout<<"The square root of "<< nom1+nom2 << "is"
    << result;}
}
```


More Mathematical Functions

Function	Standard Library	Purpose: Example	Argument(s)	Result
`abs(x)`	`<cstdlib>`	Returns the absolute value of its integer argument: if x is `-5`, `abs(x)` is `5`	`int`	`int`
`ceil(x)`	`<cmath>`	Returns the smallest integral value that is not less than x: if x is `45.23`, `ceil(x)` is `46.0`	`double`	`double`
`cos(x)`	`<cmath>`	Returns the cosine of angle x: if x is `0.0`, `cos(x)` is `1.0`	`double` (radians)	`double`
`exp(x)`	`<cmath>`	Returns e^x where e = `2.71828...`: if x is `1.0`, `exp(x)` is `2.71828`	`double`	`double`
`fabs(x)`	`<cmath>`	Returns the absolute value of its type `double` argument: if x is `-8.432`, `fabs(x)` is `8.432`	`double`	`double`
`floor(x)`	`<cmath>`	Returns the largest integral value that is not greater than x: if x is `45.23`, `floor(x)` is `45.0`	`double`	`double`

More Mathematical Functions

`log(x)`	`<cmath>`	Returns the natural logarithm of x for `x > 0.0`: if x is `2.71828`, `log(x)` is `1.0`	`double`	`double`
`log10(x)`	`<cmath>`	Returns the base-10 logarithm of x for `x > 0.0`: if x is `100.0`, `log10(x)` is `2.0`	`double`	`double`
`pow(x, y)`	`<cmath>`	Returns x^y. If x is negative, y must be integral: if x is `0.16` and y is `0.5`, `pow(x, y)` is `0.4`	`double`, `double`	`double`
`sin(x)`	`<cmath>`	Returns the sine of angle x: if x is `1.5708`, `sin(x)` is `1.0`	`double` (radians)	`double`
`sqrt(x)`	`<cmath>`	Returns the non-negative square root of x ($\sqrt{x}$) for `x ≥ 0.0`: if x is `2.25`, `sqrt(x)` is `1.5`	`double`	`double`
`tan(x)`	`<cmath>`	Returns the tangent of angle x: if x is `0.0`, `tan(x)` is `0.0`	`double` (radians)	`double`

More Mathematical Functions

- Require `cmath` header file
- Take `double` as input, return a `double`
- Commonly used functions:

`sin`	Sine
`cos`	Cosine
`tan`	Tangent
`sqrt`	Square root
`log`	Natural (e) log
`abs`	Absolute value (takes and returns an int)

More Mathematical Functions

- These require `cstdlib` header file
- `rand`(): returns a random number (`int`) between 0 and the largest int the compute holds. Yields same sequence of numbers each time program is run.
- `srand`(`x`): initializes random number generator with `unsigned int x`

In-Class Exercise

- Do Lab 10, Exercise 2, No. 1 (pg. 141)
- Do Lab 10, Exercise 2, No. 2 (pg. 141)

General Purpose Library

- These require `cstdlib` header file
- `rand()` : returns a random number (`int`) between `0` and the largest int the compute holds. Yields same sequence of numbers each time program is run.
- `srand(x)` : initializes random number generator with `unsigned int x`

Character Manipulation

- The C++ library provides several functions for testing characters.
- To use these functions, you must include the `cctype` header file.

FUNCTION	MEANING
isalpha	true if arg. is a letter, false otherwise
isalnum	true if arg. is a letter or digit, false otherwise
isdigit	true if arg. is a digit 0-9, false otherwise
islower	true if arg. is lowercase letter, false otherwise
isprint	true if arg. is a printable character, false otherwise
ispunct	true if arg. is a punctuation character, false otherwise
isupper	true if arg. is an uppercase letter, false otherwise
isspace	true if arg. is a whitespace character, false otherwise

Example – Character Testing

```
#include <iostream>
#include <cctype>
using namespace std;
int main()
{
    char input;
    cout<<"Enter any character: ";
    cin.get(input);
    if (isalpha(input)){
       cout.put(input);
       cout<<"It is an alphabet";}
    if (isdigit(input))
       cout<<"It is a digit";
    if (islower(input))
       cout<<"The letter entered is lowercase";
    if (isupper(input))
       cout<<"The letter entered is uppercase";
    return 0;
}
```


Exercise 01

- Write a program with if statements that will display the word "digit" **`if`** the variable **`ch`** contains numeric data, or display the word "letter" if the variable **`ch`** contains alphabet data. Otherwise, it should display "special character"

Character Case Conversion

- Require `cctype` header file
- Functions:

  `toupper`: if `char` argument is lowercase letter, return uppercase equivalent; otherwise, return input unchanged

```
char ch1 = 'H';
char ch2 = 'e';
char ch3 = '!';
cout << toupper(ch1);  // displays 'H'
cout << toupper(ch2);  // displays 'E'
cout << toupper(ch3);  // displays '!'
```

Character Case Conversion

- Functions:

  tolower: if char argument is uppercase letter, return lowercase equivalent; otherwise, return input unchanged

```
char ch1 = 'H';
char ch2 = 'e';
char ch3 = '!';
cout << tolower(ch1);   // displays 'h'
cout << tolower(ch2);   // displays 'e'
cout << tolower(ch3);   // displays '!'
```


Example – Character Case Conversion

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

int main()
{
    char input[15];
    cout<<"Enter a name ";
    cin>>input;
    for(int i=0;input[i] != '\0';i++)
    input[i]=toupper(input[i]);
    cout<<"The name in upper case is:" << input;
    return 0;
}
```


Character Manipulating Functions

Display 9.3 Some Functions in <cctype> *(part 1 of 2)*

FUNCTION	DESCRIPTION	EXAMPLE

toupper(*Char_Exp*)	Returns the uppercase version of *Char_Exp* (as a value of type int).	`char c = toupper('a');`
		`cout << c;`
		Outputs: A

tolower(*Char_Exp*)	Returns the lowercase version of *Char_Exp* (as a value of type int).	`char c = tolower('A');`
		`cout << c;`
		Outputs: a

isupper(*Char_Exp*)	Returns true provided *Char_Exp* is an uppercase letter; otherwise, returns false.	`if (isupper(c))`
		`  cout << "Is uppercase.";`
		`else`
		`  cout << "Is not uppercase.";`

islower(*Char_Exp*)	Returns true provided *Char_Exp* is a lowercase letter; otherwise, returns false.	`char c = 'a';`
		`if (islower(c))`
		`  cout << c << " is lowercase.";`
		Outputs: a is lowercase.

isalpha(*Char_Exp*)	Returns true provided *Char_Exp* is a letter of the alphabet; otherwise, returns false.	`char c = '$';`
		`if (isalpha(c))`
		`  cout << "Is a letter.";`
		`else`
		`  cout << "Is not a letter.";`
		Outputs: Is not a letter.

Display 9.3 Some Functions in <cctype> *(part 2 of 2)*

FUNCTION	DESCRIPTION	EXAMPLE

isdigit(*Char_Exp*)	Returns true provided *Char_Exp* is one of the digits '0' through '9'; otherwise, returns false.	`if (isdigit('3'))`
		`  cout << "It's a digit.";`
		`else`
		`  cout << "It's not a digit.";`
		Outputs: It's a digit.

isalnum(*Char_Exp*)	Returns true provided *Char_Exp* is either a letter or a digit; otherwise, returns false.	`if (isalnum('3') && isalnum('a'))`
		`  cout << "Both alphanumeric.";`
		`else`
		`  cout << "One or more are not.";`
		Outputs: Both alphanumeric.

isspace(*Char_Exp*)	Returns true provided *Char_Exp* is a whitespace character, such as the blank or newline character; otherwise, returns false.	`//Skips over one "word" and sets c`
		`//equal to the first whitespace`
		`//character after the "word":`
		`do`
		`{`
		`  cin.get(c);`
		`} while (! isspace(c));`

ispunct(*Char_Exp*)	Returns true provided *Char_Exp* is a printing character other than whitespace, a digit, or a letter; otherwise, returns false.	`if (ispunct('?'))`
		`  cout << "Is punctuation.";`
		`else`
		`  cout << "Not punctuation.";`

isprint(*Char_Exp*)	Returns true provided *Char_Exp* is a printing character; otherwise, returns false.	
isgraph(*Char_Exp*)	Returns true provided *Char_Exp* is a printing character other than whitespace; otherwise, returns false.	
isctrl(*Char_Exp*)	Returns true provided *Char_Exp* is a control character; otherwise, returns false.	

Exercise 02

- Write a program to toggle the contents of a string character from lower to upper case or vice verse.

Review of the Internal Storage of C-Strings

- **C-string**: sequence of characters stored in adjacent memory locations and terminated by `NULL` character
- **String literal** (**string constant**): sequence of characters enclosed in double quotes " " :

```
"Hi there!"
```

H	i		t	h	e	r	e	!	\0

Review of the Internal Storage of C-Strings

- Array of `chars` can be used to define storage for string:

```
const int SIZE = 20;
char city[SIZE];
```

- Leave room for `NULL` at end
- Can enter a value using `cin` or `>>`
  - Input is whitespace-terminated
  - No check to see if enough space
- For input containing whitespace, and to control amount of input, use `cin.getline()`

C-Strings Manipulation Functions

- The C++ library has numerous functions for handling C-strings.
- Requires `cstring` header file be included.
- Functions take one or more C-strings as arguments. Can use:
  - C-string name
  - pointer to C-string
  - literal string

Functions for C-Strings

- Functions:
  - `strlen(str)`: returns length of C-string `str`

```
char city[SIZE] = "Missoula";
cout << strlen(city); // prints 8
```

  - `strcat(str1, str2)`: appends str2 to the end of str1

```
char location[SIZE] = "Missoula, ";
char state[3] = "MT";
strcat(location, state);
// location now has "Missoula, MT"
```


Function for C-Strings

- Functions:
  - `strcpy(str1, str2)`: copies `str2` to `str1`

```
const int SIZE = 20;
char fname[SIZE] = "Maureen",
name[SIZE];
strcpy(name, fname);
```

Note: `strcat` and `strcpy` perform no bounds checking to determine if there is enough space in receiving character array to hold the string it is being assigned.

Function for C-Strings

- Functions:
  - `strstr(str1, str2)`: finds the first occurrence of `str2` in `str1`. Returns a pointer to match, or `NULL` if no match.

```
char river[] = "Wabash";
char word[] = "aba";
cout << strstr(state, word);
// displays "abash"
```


Comparing C-Strings

- Also cannot use operator ==

```
char aString[10] = "Hello";
char anotherString[10] = "Goodbye";
aString == anotherString; // NOT allowed!
```

- Must use library function again:

```
if (strcmp(aString, anotherString))
    cout << "Strings NOT same.";
else
    cout << "Strings are same.";
```

Working with C-Strings - Example

```
#include <iostream>
#include <cstring>
using namespace std;
int main()
{  char reply;
   char garment[]="overcoat";
   cout<<"Is it raining outside? Answer y/n\n";
   cin>>reply;
   if(reply=='y')
      strcpy(garment,"raincoat");
  cout<<"before you go out today take your "<<garment;
    return 0;
}
```


Predefined C-String Functions

Display 9.1 **Some Predefined C-String Functions in <cstring> *(part 1 of 2)***

FUNCTION	DESCRIPTION	CAUTIONS
strcpy(*Target_String_Var*, *Src_String*)	Copies the C-string value *Src_String* into the C-string variable *Target_String_Var*.	Does not check to make sure *Target_String_Var* is large enough to hold the value *Src_String*.
strcpy(*Target_String_Var*, *Src_String*, *Limit*)	The same as the two-argument strcpy except that at most *Limit* characters are copied.	If *Limit* is chosen carefully, this is safer than the two-argument version of strcpy. Not implemented in all versions of C++.
strcat(*Target_String_Var*, *Src_String*)	Concatenates the C-string value *Src_String* onto the end of the C-string in the C-string variable *Target_String_Var*.	Does not check to see that *Target_String_Var* is large enough to hold the result of the concatenation.
strcat(*Target_String_Var*, *Src_String*, *Limit*)	The same as the two argument strcat except that at most *Limit* characters are appended.	If *Limit* is chosen carefully, this is safer than the two-argument version of strcat. Not implemented in all versions of C++.

Predefined C-String Functions

Display 9.1 **Some Predefined C-String Functions in <cstring> *(part 2 of 2)***

FUNCTION	DESCRIPTION	CAUTIONS
strlen(*Src_String*)	Returns an integer equal to the length of *Src_String*. (The null character, '\0', is not counted in the length.)	
strcmp(*String_1*, *String_2*)	Returns 0 if *String_1* and *String_2* are the same. Returns a value < 0 if *String_1* is less than *String_2*. Returns a value > 0 if *String_1* is greater than *String_2* (that is, returns a nonzero value if *String_1* and *String_2* are different). The order is lexicographic.	If *String_1* equals *String_2*, this function returns 0, which converts to false. Note that this is the reverse of what you might expect it to return when the strings are equal.
strcmp(*String_1*, *String_2*, *Limit*)	The same as the two-argument strcat except that at most *Limit* characters are compared.	If *Limit* is chosen carefully, this is safer than the two-argument version of strcmp. Not implemented in all versions of C++.

String/Numeric Conversion Functions

- These functions convert between string and numeric forms of numbers
- Need to include `cstdlib` header file

FUNCTION	PARAMETER	ACTION
atoi	C-string	converts C-string to an int value, returns the value
atol	C-string	converts C-string to a long value, returns the value
atof	C-string	converts C-string to a double value, returns the value
itoa	int,C-string, int	converts 1st int parameter to a C-string, stores it in 2nd parameter. 3rd parameter is base of converted value

String/Numeric Conversion Functions

- `atoi` converts alphanumeric to int
- `atol` converts alphanumeric to long
- `atof` converts a numeric string to a double

- if C-string being converted contains non-digits, results are undefined
  - function may return result of conversion up to first non-digit
  - function may return 0

String/Numeric Conversion Functions

- Examples:
  - `int number;`
  - `long lnumber;`
  - `double dnumber;`
  - `number = atoi("57");`
  - `lnumber = atol("50000");`
  - `dnumber = atof("590.55");`

String/Numeric Conversion Functions

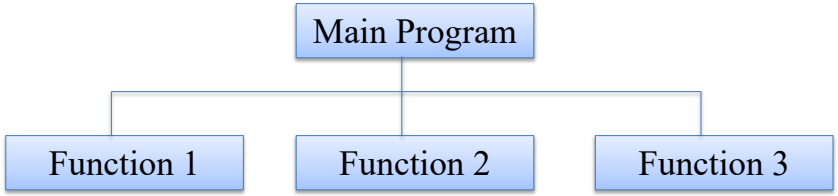
- `itoa` converts an int to an alphanumeric string
- Allows user to specify the base of conversion

```
itoa(int num, char numStr, int base)
```

  - **`num`** : number to convert
  - **`numStr`**: array to hold resulting string
  - **`base`**: base of conversion
- Example: To convert the number 1200 to a hexadecimal string
  - `char numStr[10];`
  - `itoa(1200, numStr, 16);`

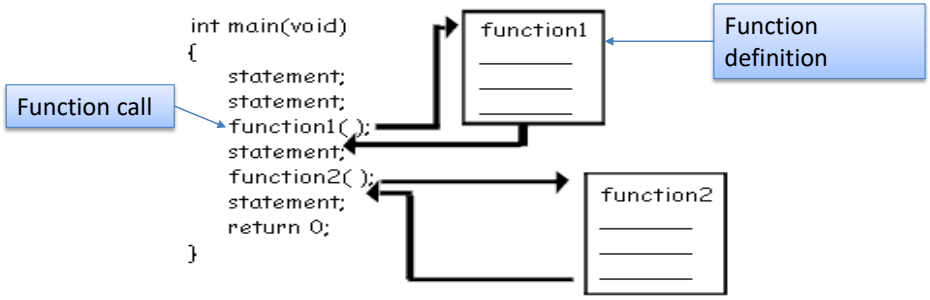
User-Defined Functions

- User-defined functions are created by you, the programmer.
- Commonly used to break a problem down into small manageable pieces.
- You are already familiar with the one function that every C++ program possesses: `int main()`
  - Ideally, your `main()` function should be very short and should consist primarily of function calls.



Defining and Calling Functions

- Every functions must have:
  - **Function call:** statement causes a function to execute
  - **Function definition:** statements that make up a function



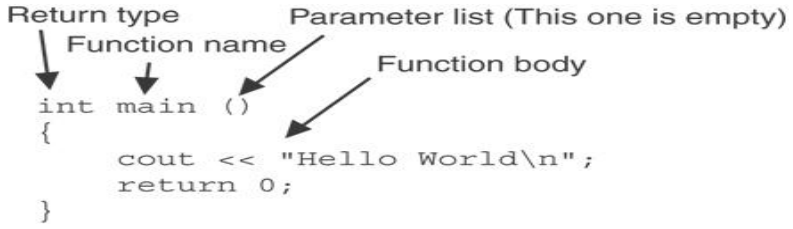
Function Definition

- Definition includes:
  - return type: data type of the value that function returns to the part of the program that calls it
  - name: name of the function. Function names follow same rules as variables
  - parameter list: variables containing values passed to the function
  - body: statements that perform the function's task, enclosed in { }

Function Definition (cont.)

- The general form of a function definition in C++ is as follows:

```
function-returntype function-name( parameter-list )
{
        local-definitions;
        function-implementation;
}
```



Note: The line that reads `int main()` is the function header.

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";
}
```

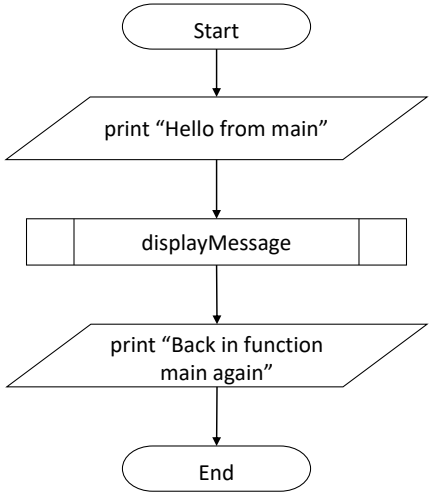
Calling a Function

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

```
printHeading();
```

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

The Flowchart

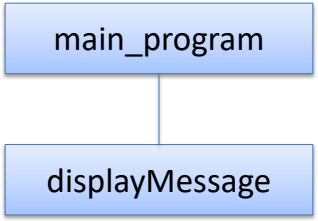


The Pseudo Code

- Start
- Print “Hello from main”
- call displayMessage
- Print “Back in function main again”
- End

- displayMessage:
  - Print “Hello from the function displayMessage”

The Structure Chart



Calling a Function - Example

Function definition

Function call

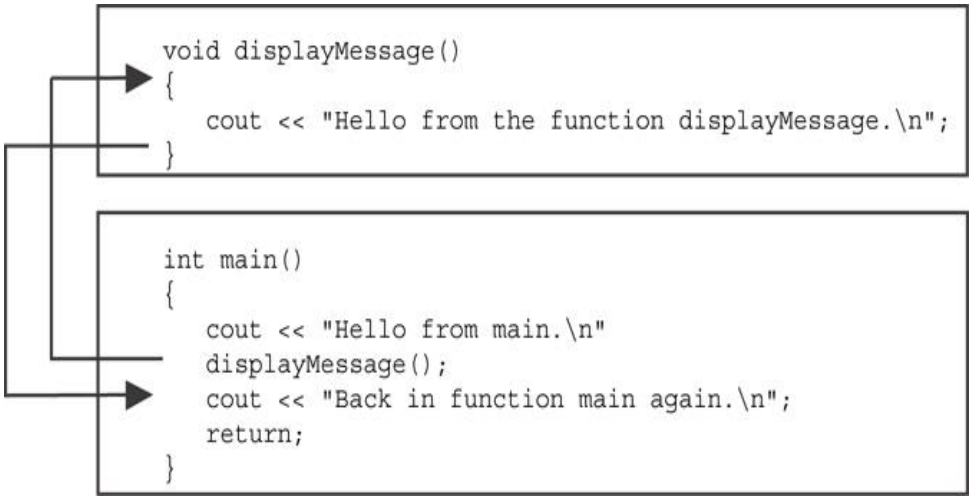
Program 6-1

```
1  // This program has two functions: main and displayMessage
2  #include <iostream>
3  using namespace std;
4
5  //*****************************************
6  // Definition of function displayMessage  *
7  // This function displays a greeting.     *
8  //*****************************************
9
10 void displayMessage()
11 {
12    cout << "Hello from the function displayMessage.\n";
13 }
14
15 //*****************************************
16 // Function main                          *
17 //*****************************************
18
19 int main()
20 {
21    cout << "Hello from main.\n";
22    displayMessage();
23    cout << "Back in function main again.\n";
24    return 0;
25 }
```

Program Output

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


Flow of Control in Program 6-1



User-Defined Functions: Example 2

```
01 #include <iostream>
02 #include <cmath>
03 using namespace std;
04
05 float distance(float x, float y)
06 {   float dist;
07     dist = sqrt(x * x + y * y);
08     return dist;
09 }
10
11 void main()
12 {
13     float x,y,dist;
14     cout << "Testing function distance(x,y)"  << endl;
15     cout << "Enter values for x and y: ";
16     cin >> x >> y;
17     dist = distance(x,y);
18     cout << "Distance of (" << x << ',' << y << ") from origin is
       " << dist << endl << "Tested" << endl;
19 }
```


Calling Functions

- `main` 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

In-Class Exercise

- Do Lab 11, Exercise 1, No 1 (pg. 147-149)
- Which of the following function headers are valid? If they are invalid, explain why.
  - `one (int a, int b)`
  - `int thisone(char x)`
  - `char another (int a, b)`
  - `double yetanother`

Function Prototypes

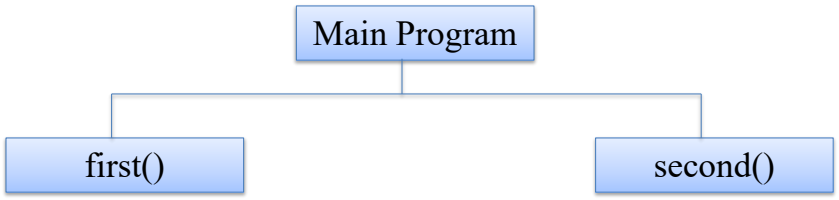
- Ways to **notify the compiler** about a function before a call to the function:
  - Place function definition before calling function's definition
  - Use a function prototype (function declaration) – like the function definition without the body
    - Header: `void printHeading()`
    - Prototype: `void printHeading();`

User-Defined Functions: Function Prototypes

```
01  #include <iostream>
02
03  void first();
04  void second();
05
06  void main()
07  {
08     cout<< "Starting in main function \n";
09     first();
10     second();
11     cout<<"Control back to main\n";
12  }
13
14  void first()
15  {  cout<<"Inside the first function\n";}
16
17  void second()
18  {  cout<<"Inside the second function\n";}
```

Function prototypes

Structured Chart



Prototype Notes

- Place prototypes near **top** of program
- Program must include either prototype **or** full function definition before any call to the function – compiler error otherwise
- When using prototypes, can place function definitions in any order in source file

User-Defined Functions: Functions with No Parameters

```
01  #include <iostream>
02  void printhi();
03
04  void main(){
05     cout << "Testing function printhi()" << endl;
06     printhi();
07     cout << "Tested" << endl;
08  } // End of main
09
10  // Function Definitions
11  void printhi()
12  {
13     cout << "Hi \n";
14  }
```


In-Class Exercise

- Do Lab 11, Exercise 3, No. 1 (pg. 163)

Sending Data into a Function

- Can **pass values** into a function at time of call:

```
c = pow(a, b);
```

- Values passed to function are arguments
- Variables in a function that **hold the values passed** as arguments are parameters

A Function with a Parameter Variable

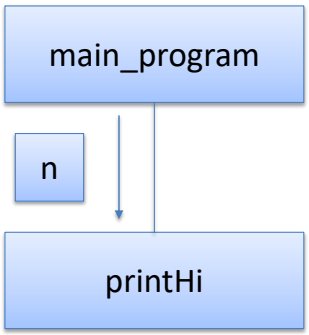
```
01  void displayValue(int num)
02  {
03      cout << "The value is " << num << endl;
04  }
```

The integer variable `num` is a **parameter**.
It accepts any integer value passed to the function.

User-Defined Functions: Functions with Parameter and No Return Values (1)

```
01  #include <iostream>
02  using namespace std;
03  void printhi(int);
04
05  void main(){
06     int n;
07     cout <<"Enter a value for n: ";
08     cin >> n;
09     printhi(n);
10     cout << "Tested \n";}
11
12  void printhi(int n)
13  {
14     int i;
14     for (i = 0; i < n; i++)
15     cout << "Hi \n";
16  }
```


The Structure Chart



User-Defined Functions: Functions with Parameter and No Return Values (2)

```
01  #include <iostream>
02  using namespace std;
03
04  void displayValue(int);
05
06  void main()
07  {
08     cout<<"Passing number 5 to displayValue\n";
09      displayValue(5);
10      cout<<"Back in main\n";
11  }
12
13  void displayValue(int n)
14  {
15     cout<<"The value is " << n << endl;
16  }
```

Other Parameter Terminology

- A parameter can also be called a formal parameter or a formal argument
- An argument can also be called an actual parameter or an actual argument

Parameters, Prototypes, and Function Headers

- For each function argument,
  - the **prototype** must include the **data type** of each parameter inside its parentheses
  - the **header** must include a **declaration** for each parameter in its ()

```
void evenOrOdd(int);   //prototype
void evenOrOdd(int num) //header
evenOrOdd(val);        //call
```


Function Call Notes

- Value of argument is **copied** into parameter when the function is called
- A parameter's scope is the function which uses it
- Function can have **multiple** parameters
- There **must** be a **data type** listed in the prototype () and an **argument declaration** in the function header () for each parameter
- Arguments will be promoted/demoted as necessary to **match** parameters

In-Class Exercise

- What is the output of this program?

```
01  #include <iostream>
02
03  // Function prototype
04  void showDouble(int);
05
06  int main(){
07      int num;
08      for (num = 0; num < 10; num++)
09      showDouble(num);
10      system("pause");
11      return 0;
12  }
13
14  //Definition of function
15  void showDouble(int value) {
16      cout<<value <<"\t";
17      cout << (value * 2)<< endl;
18  }
```

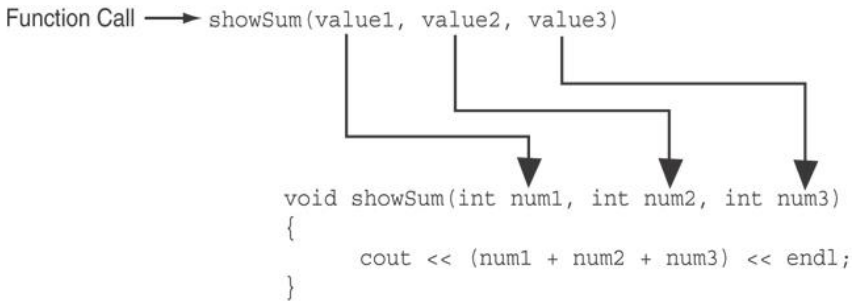
User Defined Functions: Passing Multiple Arguments

- When calling a function and passing multiple arguments:
  - the number of arguments in the call must match the prototype and definition
  - the first argument will be used to initialize the first parameter, the second argument to initialize the second parameter, etc.

User-Defined Functions: Passing Multiple Arguments (cont.)

```
01  #include <iostream>
02  using namespace std;
03  void showSum(int, int, int);
04
05  int main()
06  {
07     int value1, value2, value3;
08
09     cout<<"Enter 3 integers: ";
10     cin>> value1 >> value2 >> value3;
11     showSum(value1, value2, value3);
12
13     return 0;
14  }
15
16  void showSum(int a, int b, int c)
17  {
18     cout<<"The sum: "<<a+b+c;
19  }
```


Passing Multiple Arguments (cont.)



The function call in line 18 passes value1, value2, and value3 as a arguments to the function.

In-Class Exercise

- What is the output of this program?

```
01   #include <iostream>
02
03   // Function prototype
04   void func1(double, int);
05
06   int main(){
07       int x = 0; double y = 1.5;
08       cout << x << " " <<y<< endl;
09       func1 (y, x);
10       cout << x << " " <<y<< endl;
11       system ("pause");
12       return 0;
13   }
14
15   void func1(double a, int b){
16       cout << a << " " <<b<< endl;
17       a=0.0; b=10;
18       cout << a << " " <<b<< endl;
19   }
```

User-Defined Functions: Passing Data

- Passing by Value
- Passing by Reference

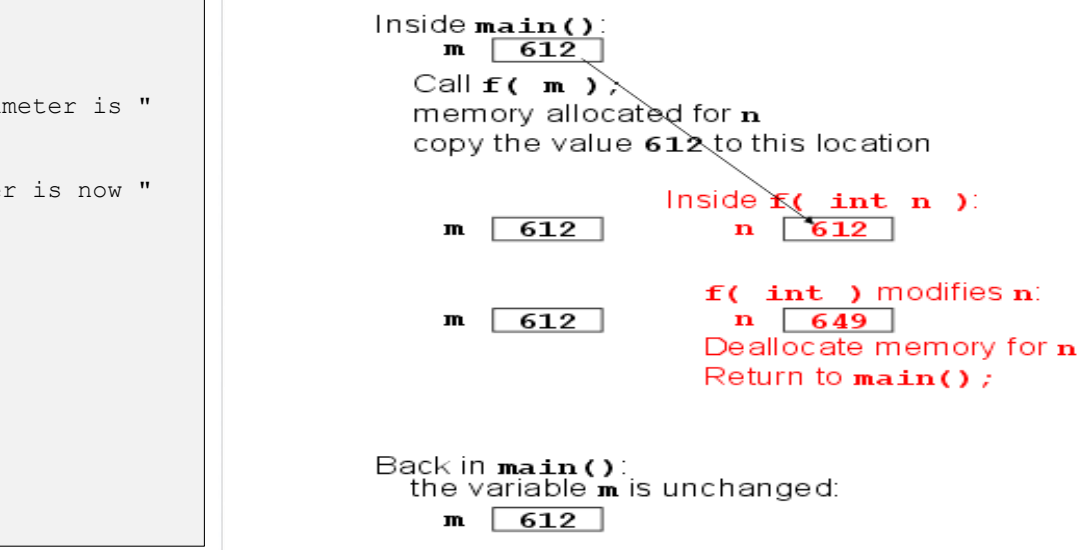
Passing Data by Value

- Pass by value: when an argument is passed to a function, its value is **copied** into the parameter.
- Changes to the parameter in the function do not affect the value of the argument

User-Defined Functions: Passing Data by Value (cont.)

```
01  #include <iostream>
02  using namespace std;
03
04  void f( int n ) {
05     cout << "Inside f( int ), the value of the parameter is "
       << n << endl;
06     n += 37;
07     cout << "Inside f( int ), the modified parameter is now "
       << n << endl;}
08
09  int main() {
10     int m = 612;
11
12     cout << "The integer m = " << m << endl;
13     cout << "Calling f( m )..." << endl;
14     f( m );
15     cout << "The integer m = " << m << endl;
16     return 0;
17  }
```


User-Defined Functions: Passing Data by Value (cont.)



Passing Information to Parameters by Value

- Example: `int val=5;`
  `evenOrOdd(val);`

- `evenOrOdd` can change variable `num`, but it will have no effect on variable `val`

In-Class Exercise

- Do Lab 11, Exercise 1, No. 12 (pg. 152)
- Do Lab 11, Exercise 3, No. 3 (pg. 164)

Using Functions in Menu-Driven Programs

- Functions can be used
  - to implement user choices from menu
  - to implement general-purpose tasks:
    - Higher-level functions can call general-purpose functions, minimizing the total number of functions and speeding program development time
- **See Program 6-10 in the text book (pg. 310-311)**

The `return` Statement

- Used to end execution of a function
- Can be placed anywhere in a function
  - Statements that follow the `return` statement will not be executed
- Can be used to prevent abnormal termination of program
- In a `void` function without a `return` statement, the function ends at its last }

Returning a Value from a Function

- A function can return a value back to the statement that called the function.
- You've already seen the `pow` function, which returns a value:

```
double x;
x = pow(2.0, 10.0);
```


Returning a Value from a Function

- In a value-returning function, the `return` statement can be used to return a value from function to the point of call. Example:

```
int sum(int num1, int num2)
{
  double result;
  result = num1 + num2;
  return result;
}
```


A Value-Returning Function

Return Type

```
int sum(int num1, int num2)
{
    double result;
    result = num1 + num2;
    return result;
}
```

Value Being Returned

A Value-Returning Function

```
int sum(int num1, int num2)
{
    return num1 + num2;
}
```

Functions can return the values of expressions, such as `num1 + num2`

The `return` Statement - Example

Program 6-11

```
1  // This program uses a function to perform division. If division
2  // by zero is detected, the function returns.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype.
7  void divide(double, double);
8
9  int main()
10 {
11    double num1, num2;
12
13    cout << "Enter two numbers and I will divide the first\n";
14    cout << "number by the second number: ";
15    cin >> num1 >> num2;
16    divide(num1, num2);
17    return 0;
18 }
```


The `return` Statement - Example

Program 6-11(Continued)

Return to called function

```
20 //*************************************************************
21 // Definition of function divide.                            *
22 // Uses two parameters: arg1 and arg2. The function divides arg1*
23 // by arg2 and shows the result. If arg2 is zero, however, the  *
24 // function returns.                                          *
25 //*************************************************************
26
27 void divide(double arg1, double arg2)
28 {
29    if (arg2 == 0.0)
30    {
31       cout << "Sorry, I cannot divide by zero.\n";
32       return;
33    }
34    cout << "The quotient is " << (arg1 / arg2) << endl;
35 }
```

Program Output with Example Input Shown in Bold

```
Enter two numbers and I will divide the first
number by the second number: 12 0 [Enter]
Sorry, I cannot divide by zero.
```


Returning a Value From a Function

Program 6-12

```
1  // This program uses a function that returns a value.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype
6  int sum(int, int);
7
8  int main()
9  {
10    int value1 = 20,    // The first value
11        value2 = 40,    // The second value
12        total;          // To hold the total
13
14    // Call the sum function, passing the contents of
15    // value1 and value2 as arguments. Assign the return
16    // value to the total variable.
17    total = sum(value1, value2);
18
19    // Display the sum of the values.
20    cout << "The sum of " << value1 << " and "
21         << value2 << " is " << total << endl;
22    return 0;
23 }
```


Returning a Value From a Function

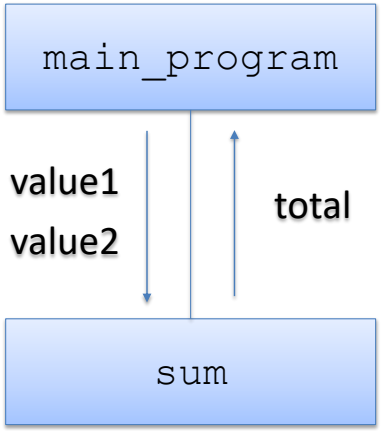
Program 6-12 (Continued)

```
24
25 //*********************************************************
26 // Definition of function sum. This function returns  *
27 // the sum of its two parameters.                      *
28 //*********************************************************
29
30 int sum(int num1, int num2)
31 {
32    return num1 + num2;
33 }
```

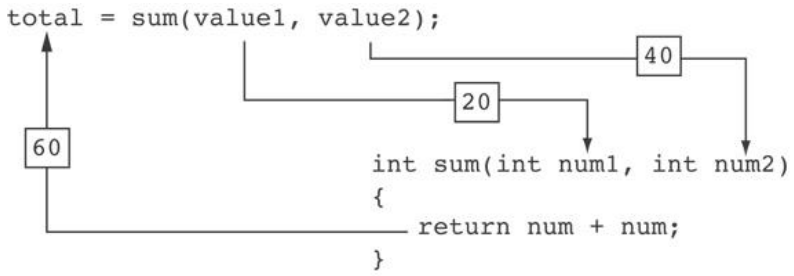
Program Output

```
The sum of 20 and 40 is 60
```

The Structure Chart



Returning a Value From a Function



The statement in line 17 calls the sum function, passing `value1` and `value2` as arguments. The return value is assigned to the `total` variable.

Returning a Value From a Function

- The prototype and the definition must indicate the data type of return value (not `void`)
- Calling function should use return value:
  - assign it to a variable
  - send it to `cout`
  - use it in an expression

Returning a Boolean Value

- Function can return `true` or `false`
- Declare return type in function prototype and heading as `bool`
- Function body must contain `return` statement(s) that return `true` or `false`
- Calling function can use return value in a relational expression

Returning a Boolean Value

Program 6-14

```
1  // This program uses a function that returns true or false.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype
6  bool isEven(int);
7
8  int main()
9  {
10    int val;
11
12    // Get a number from the user.
13    cout << "Enter an integer and I will tell you ";
14    cout << "if it is even or odd: ";
15    cin >> val;
16
```


Returning a Boolean Value

Program 6-14 *(continued)*

```
17    // Indicate whether it is even or odd.
18    if (isEven(val))
19       cout << val << " is even.\n";
20    else
21       cout << val << " is odd.\n";
22    return 0;
23 }
24
25 //*****************************************************************
26 // Definition of function isEven. This function accepts an        *
27 // integer argument and tests it to be even or odd. The function  *
28 // returns true if the argument is even or false if the argument  *
29 // is odd. The return value is an bool.                           *
30 //*****************************************************************
31
32 bool isEven(int number)
33 {
34    bool status;
35
36    if (number % 2)
37       status = false;   // number is odd if there's a remainder.
38    else
39       status = true;    // Otherwise, the number is even.
40    return status;
41 }
```

Program Output with Example Input Shown In Bold

```
Enter an integer and I will tell you if it is even or odd: 5 [Enter]
5 is odd.
```


In-Class Exercise

```
#include <iostream>
using namespace std;
void try1(int p);
int try3(int r);

int main()
{ int a=2;
 cout << a <<endl;
 try1(a);
 cout << a <<endl;
 int b=3;
 cout << b <<endl;
 int c=4;
 try3(c);
 cout << c <<endl;
 c=try3(c);
 cout << c <<endl;
 cout << try3(5) <<endl;
  return 0;}
```

```
void try1(int p)
{
 p++;
 cout << p <<endl;
}

int try3(int r)
{
  return r*r;
}
```


In-Class Exercise

- Do Lab 11, Exercise 2, No. 2 – Program 11.9 (pg. 159)
- Do Lab 11, Exercise 2, No. 3 (pg. 164)
- Write a function prototype and header for a function named `distance`. The function should return a `double` and have a two `double` parameters: `rate` and `time`.
- Write a function prototype and header for a function named `days`. The function should return an integer and have three integer parameters: `years`, `months` and `weeks`.
- Examine the following function header, then write an example call to the function.
  - `void showValue(int quantity)`

In-Class Exercise

- The following statement calls a function named `half`. The `half` function returns a value that is half that of the argument. Write the function.

```
result = half(number);
```

- A program contains the following function:

```
int cube (int num)
{
    return num*num*num;
}
```

Write a statement that passes the value 4 to this function and assigns its return value to the variable `result`.

In-Class Exercise

- Write a C++ program to calculate a rectangle's area. The program consists of the following functions:
  - **`getLength`** – This function should ask the user to enter the rectangle's length, and then returns that value as a double.
  - **`getWidth`** – This function should ask the user to enter the rectangle's width, and then returns that value as a double.
  - **`getArea`** – This function should accept the rectangle's length and width as arguments and return the rectangle's area.
  - **`displayData`** – This function should accept the rectangle's length, width and area as arguments, and display them in an appropriate message on the screen.
  - **`main`** – This function consists of calls to the above functions.

Local and Global Variables

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

Local and Global Variables - Example

Program 6-15

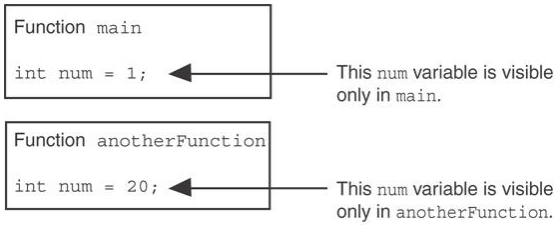
```
1  // This program shows that variables defined in a function
2  // are hidden from other functions.
3  #include <iostream>
4  using namespace std;
5
6  void anotherFunction(); // Function prototype
7
8  int main()
9  {
10    int num = 1;   // Local variable
11
12    cout << "In main, num is " << num << endl;
13    anotherFunction();
14    cout << "Back in main, num is " << num << endl;
15    return 0;
16 }
17
18 //*********************************************************
19 // Definition of anotherFunction                          *
20 // It has a local variable, num, whose initial value      *
21 // is displayed.                                          *
22 //*********************************************************
23
24 void anotherFunction()
25 {
26    int num = 20;  // Local variable
27
28    cout << "In anotherFunction, num is " << num << endl;
29 }
```

Program Output

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

Local and Global Variables - Example

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



Local Variable Lifetime

- A function's local variables exist only while the **function is executing**. This is known as the *lifetime* of a local variable.
- When the function begins, its local variables and its parameter variables are **created** in memory, and when the function ends, the local variables and parameter variables are **destroyed**.
- This means that any value stored in a local variable is lost between calls to the function in which the variable is declared.

Global Variables and Global Constants

- 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

Global Variables and Global Constants

- You should **avoid** using global variables because they make programs difficult to debug.
- Any global that you create should be ***global constants***.

Global Constants – Example

Program 6-18

```
1  // This program calculates gross pay.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  // Global constants
7  const double PAY_RATE = 22.55;     // Hourly pay rate
8  const double BASE_HOURS = 40.0;    // Max non-overtime hours
9  const double OT_MULTIPLIER = 1.5; // Overtime multiplier
10
11 // Function prototypes
12 double getBasePay(double);
13 double getOvertimePay(double);
14
15 int main()
16 {
17    double hours,           // Hours worked
18           basePay,         // Base pay
19           overtime = 0.0,  // Overtime pay
20           totalPay;        // Total pay
```

Global constants defined for values that do not change throughout the program's execution.

Global Constants – Example

The constants are then used for those values throughout the program.

```
29     // Get overtime pay, if any.
30     if (hours > BASE_HOURS)
31        overtime = getOvertimePay(hours);
```

```
56     // Determine base pay.
57     if (hoursWorked > BASE_HOURS)
58        basePay = BASE_HOURS * PAY_RATE;
59     else
60        basePay = hoursWorked * PAY_RATE;
```

```
75     // Determine overtime pay.
76     if (hoursWorked > BASE_HOURS)
77     {
78        overtimePay = (hoursWorked - BASE_HOURS) *
79                 PAY_RATE * OT_MULTIPLIER;
```


Initializing Local and Global Variables

- Local variables are not automatically initialized. They must be initialized by programmer.
- Global variables (not constants) are automatically initialized to `0` (numeric) or `NULL` (character) when the variable is defined.

In-Class Exercise

- Do Lab 11, Exercise 1, No. 17 (pg. 155 -156)

Static Local Variables

- Local variables only exist while the function is executing. When the function terminates, the contents of local variables are lost.
- `static` local variables retain their contents between function calls.
- `static` local variables are defined and initialized only the first time the function is executed. `0` is the default initialization value.

Local Variables - Example

Program 6-20

```
1  // This program shows that local variables do not retain
2  // their values between function calls.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype
7  void showLocal();
8
9  int main()
10 {
11    showLocal();
12    showLocal();
13    return 0;
14 }
15
```


Local Variables - Example

Program 6-20 *(continued)*

```
16 //*************************************************************
17 // Definition of function showLocal.                          *
18 // The initial value of localNum, which is 5, is displayed.  *
19 // The value of localNum is then changed to 99 before the     *
20 // function returns.                                          *
21 //*************************************************************
22
23 void showLocal()
24 {
25    int localNum = 5; // Local variable
26
27    cout << "localNum is " << localNum << endl;
28    localNum = 99;
29 }
```

Program Output

```
localNum is 5
localNum is 5
```

In this program, each time `showLocal` is called, the `localNum` variable is re-created and initialized with the value 5.

A Different Approach, Using a Static Variable

Program 6-21

```
1  // This program uses a static local variable.
2  #include <iostream>
3  using namespace std;
4
5  void showStatic(); // Function prototype
6
7  int main()
8  {
9     // Call the showStatic function five times.
10    for (int count = 0; count < 5; count++)
11       showStatic();
12    return 0;
13 }
14
```

Using a Static Variable - Example

Program 6-21 *(continued)*

```
15  //*************************************************************
16  // Definition of function showStatic.                         *
17  // statNum is a static local variable. Its value is displayed *
18  // and then incremented just before the function returns.     *
19  //*************************************************************
20
21  void showStatic()
22  {
23     static int statNum;
24
25     cout << "statNum is " << statNum << endl;
26     statNum++;
27  }
```

Program Output

```
statNum is 0
statNum is 1
statNum is 2
statNum is 3
statNum is 4
```

`statNum` is automatically initialized to 0. Notice that it retains its value between function calls.

Using a Static Variable - Example

If you do initialize a local static variable, the initialization only happens once. See Program 6-22…

Program 6-22 *(continued)*

```
16  //*************************************************************
17  // Definition of function showStatic.                         *
18  // statNum is a static local variable. Its value is displayed *
19  // and then incremented just before the function returns.     *
20  //*************************************************************
21
22  void showStatic()
23  {
24     static int statNum = 5;
25
26     cout << "statNum is " << statNum << endl;
27     statNum++;
28  }
```

Program Output

```
statNum is 5
statNum is 6
statNum is 7
statNum is 8
statNum is 9
```


In-Class Exercise

- Given the following programs compare the output and reason the output.

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

void showVar();

int main ( ) {
  for (int count=0;count<10; count++)
  showVar();
  system("pause");
  return 0;
}

void showVar() {
  static int var = 10;
  cout << var << endl;
  var++;
}
```

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

void showVar();

int main ( ) {
  for(int count=0;count<10; count++)
  showVar();
  system("pause");
  return 0;
}

void showVar() {
  int var = 10;
  cout << var << endl;
  var++;
}
```


In-Class Exercise

- Identify global variables & local variables in the following program. What is the output?

```
#include <iostream>
using namespace std;
int j = 8;

int main()
{
    int i=0;
    cout<<"i: "<<i<<endl;
    cout<<"j: "<<j<<endl;
    system("pause");
    return 0;
}
```

- Identify global variables, local variables and static local variables in the following program. What is the output?

```
#include <iostream>
using namespace std;
int j = 40;

void p()
{     int i=5;
      static int j=5;
      i++;
      j++;
      cout<<"i: "<<i<<endl;
      cout<<"j: "<<j<<endl;
}
int main()
{    p();
     p();
     return 0;}
```

Using Reference Variables as Parameters

- A mechanism that allows a function to work with the **original argument** from the function call, not a copy of the argument
- Allows the function to **modify values** stored in the calling environment
- Provides a way for the function to 'return' more than one value

Passing by Reference

- A reference variable is an alias for another variable
- Defined with an ampersand (&)

```
void getDimensions(int&, int&);
```

- Changes to a reference variable are made to the variable it refers to
- Use reference variables to implement passing parameters *by reference*

Passing by Reference – Example

The & here in the prototype indicates that the parameter is a reference variable.

Program 6-24

```
1  // This program uses a reference variable as a function
2  // parameter.
3  #include <iostream>
4  using namespace std;
5
6  // Function prototype. The parameter is a reference variable.
7  void doubleNum(int &);
8
9  int main()
10 {
11    int value = 4;
12
13    cout << "In main, value is " << value << endl;
14    cout << "Now calling doubleNum..." << endl;
15    doubleNum(value);
16    cout << "Now back in main. value is " << value << endl;
17    return 0;
18 }
19
```

Here we are passing value by reference.

Passing by Reference - Example

Program 6-24 *(Continued)*

The & also appears here in the function header.

```
20  //*********************************************************
21  // Definition of doubleNum.                               *
22  // The parameter refVar is a reference variable. The value *
23  // in refVar is doubled.                                  *
24  //*********************************************************
25
26  void doubleNum (int &refVar)
27  {
28     refVar *= 2;
29  }
```

Program Output

```
In main, value is 4
Now calling doubleNum...
Now back in main. value is 8
```


Reference Variables Notes

- Each reference parameter must contain &
- Space between type and & is unimportant
- Must use & in both prototype and header
- Argument passed to reference parameter must be a variable – cannot be an expression or constant
- Use when appropriate – don't use when argument should not be changed by function, or if function needs to return only 1 value

In-Class Exercise

- Do Lab 11, Exercise 1, No. 19 (pg. 157 – 158)
- Do Lab 11, Exercise 2, No. 2 – Program 11.10 (pg. 159)
- Do Lab 11, Exercise 2, No. 4 (pg. 160)


```
#include <iostream>
using namespace std;
void test(int, int&);

int main()
{
    int num;
    num=5;
    test(24, num);
    cout<<num<<endl;
    test(num,num);
    cout<<num<<endl;
    test(num*num, num);
    cout<<num<<endl;
    test(num+num,num);
    cout<<num<<endl;
    system("pause");
    return 0;
}
```

```
void test(int first, int& second)
{
    int third;
    third=first+second*second+2;
    first=second-first;
    second=2*second;
    cout<<first<<" "<<second<<"
  "<<third<<endl;
}
```

```
#include <iostream>
using namespace std;
void test(int&, int&,int,int&);

int main()
{   int a,b,c,d;
   a=3;  b=4;  c=20;  d=78;
 cout<<a<<" "<<b<<" "<<c<<"
   "<<d<<endl;
    test(a,b,c,d);
 cout<<a<<" "<<b<<" "<<c<<"
   "<<d<<endl;
    d=a*b+c-d;
    test(a,b,c,d);
    cout<<a<<" "<<b<<" "<<c<<"
   "<<d<<endl;
    return 0;
}
```

```
void test(int& a, int& b, int c, int&
   d)
{
     cin>>a >> b>> c>> d;
     c = a* b+d-c;
     c=2*c;
}
```

The input:

6 8 12 35
8 9 30 45

Default Arguments

- A Default argument is an argument that is passed automatically to a parameter if the argument is missing on the function call.
- Must be a constant declared in prototype:

```
void evenOrOdd(int = 0);
```

- Can be declared in header if no prototype
- Multi-parameter functions may have default arguments for some or all of them:

```
int getSum(int, int=0, int=0);
```


Default Arguments – Example

Default arguments specified in the prototype

Program 6-23

```
1  // This program demonstrates default function arguments.
2  #include <iostream>
3  using namespace std;
4
5  // Function prototype with default arguments
6  void displayStars(int = 10, int = 1);
7
8  int main()
9  {
10    displayStars();      // Use default values for cols and rows.
11    cout << endl;
12    displayStars(5);     // Use default value for rows.
13    cout << endl;
14    displayStars(7, 3);  // Use 7 for cols and 3 for rows.
15    return 0;
16 }
```

(Program Continues)

Default Arguments – Example

Program 6-23 *(Continued)*

```
18 //*********************************************************
19 // Definition of function displayStars.                   *
20 // The default argument for cols is 10 and for rows is 1.*
21 // This function displays a square made of asterisks.     *
22 //*********************************************************
23
24 void displayStars(int cols, int rows)
25 {
26    // Nested loop. The outer loop controls the rows
27    // and the inner loop controls the columns.
28    for (int down = 0; down < rows; down++)
29    {
30       for (int across = 0; across < cols; across++)
31          cout << "*";
32       cout << endl;
33    }
34 }
```

Program Output

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

*****

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

Default Arguments

- If not all parameters to a function have default values, the defaultless ones are declared first in the parameter list:

```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int);  // NO
```

- When an argument is omitted from a function call, all arguments after it must also be omitted:

```
sum = getSum(num1, num2);     // OK
sum = getSum(num1, , num3);   // NO
```


Default Arguments

- Consider the following function prototype:

```
void funcExp(int,int, double=55.5,char='A');
```

- The following function calls are legal:
  - `funcExp(3,4,45.5, 'B');`
  - `funcExp(3,4,45.5);`
  - `funcExp(3,4);`
- The following function calls are illegal:
  - `funcExp(3,4, 'C');`

Default Arguments

- The following are illegal function prototypes with default arguments:
  - `void funcOne(int,double=23.45,char, int=45);`
  - `int funcTwo(int=1, int, int=1);`
  - `void funcThree(int, int&=16,double=34);`

In-Class Exercise

- Do Lab 11, Exercise 2, No. 5 (pg. 160)

- Consider the following function prototype & function definition:

```
void testDefaultParam(int , int=5, double=3.2);

void testDefaultParam(int a, int b, double z)
{
     int u;
     a=a+static_cast<int>(2*b+z);
     u=a+b*z;
     cout<<"u = "<<a<<endl;
}
```

What is the output of the following function calls?

```
a) testDefaultParam(6);
b) testDefaultParam(3,4);
c) testDefaultParam(3,4.5);
d) testDefaultParam(3,4, 5.5);
e) testDefaultParam(3.4);
```

In-Class Exercise

- Write a function prototype and function header for a function called `compute`. The function should have 3 parameters: an `int`, a `double` and a `long`. The `int` parameter should have a default argument of 5, and the `long` parameter should have a default argument of 65536. The `double` parameter should have no default arguments. The parameters no necessarily in the order.

- Write a function prototype and function header for a function called `calculate`. The function should have 3 parameters: an `int`, a reference to a `double` and a `long`. Only the `int` parameter should have a default argument, which is 47. The parameters no necessarily in the order.

Overloading Functions

- Overloaded functions have the same name but different parameter lists
- Can be used to create functions that perform the same task but take **different parameter types or different number of parameters**
- Compiler will determine which version of function to call by argument and parameter lists

Function Overloading Examples

- Using these overloaded functions,

```
void getDimensions(int);            // 1
void getDimensions(int, int);       // 2
void getDimensions(int, double);    // 3
void getDimensions(double, double);// 4
```

- the compiler will use them as follows:

```
int length, width;
double base, height;
getDimensions(length);            // 1
getDimensions(length, width);     // 2
getDimensions(length, height);    // 3
getDimensions(height, base);      // 4
```

Function Overloading – Example

Program 6-26

```
1  // This program uses overloaded functions.
2  #include <iostream>
3  #include <iomanip>
4  using namespace std;
5
6  // Function prototypes
7  int square(int);
8  double square(double);
9
10 int main()
11 {
12    int userInt;
13    double userFloat;
14
15    // Get an int and a double.
16    cout << fixed << showpoint << setprecision(2);
17    cout << "Enter an integer and a floating-point value: ";
18    cin >> userInt >> userFloat;
19
20    // Display their squares.
21    cout << "Here are their squares: ";
22    cout << square(userInt) << " and " << square(userFloat);
23    return 0;
24 }
```

The overloaded functions have different parameter lists

Passing a double

Passing an int

(Program Continues)

Function Overloading – Example

Program 6-26 *(Continued)*

```
26 //****************************************************************
27 // Definition of overloaded function square.                     *
28 // This function uses an int parameter, number. It returns the *
29 // square of number as an int.                                   *
30 //****************************************************************
31
32 int square(int number)
33 {
34    return number * number;
35 }
36
37 //****************************************************************
38 // Definition of overloaded function square.                     *
39 // This function uses a double parameter, number. It returns    *
40 // the square of number as a double.                             *
41 //****************************************************************
42
43 double square(double number)
44 {
45    return number * number;
46 }
```

Program Output with Example Input Shown in Bold

```
Enter an integer and a floating-point value: 12 4.2 [Enter]
Here are their squares: 144 and 17.64
```


The `exit()` Function

- **Terminates** the execution of a program
- Can be called from any function
- Can pass an `int` value to operating system to indicate status of program termination
- Usually used for **abnormal termination** of program
- Requires `cstdlib` header file (Borland)

The `exit()` Function

- Example:

```
exit(0);
```

- The `cstdlib` header defines two constants that are commonly passed, to indicate success or failure:

```
exit(EXIT_SUCCESS);
exit(EXIT_FAILURE);
```

In-Class Exercise

- What is the output for the following programs

```
#include <iostream>
using namespace std;
void function();
int main(){
   function();
   cout << "Bye from main.\n";
   system ("pause");   return 0;
}

void function(){
   cout << "Bye! from function
before exit\n";
   exit(0);
   cout << "Bye! from function
before exit\n";
}
```

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

int function();
int main(){
   function();
   cout << "Bye from main.\n";
   system ("pause"); return 0;
}

int function(){
   cout << "Bye! from function
before return\n";
   return 0;
   cout << "Bye! from function
before return\n";
}
```


In-Class Exercise

- Do Lab 11, Exercise 2, No. 7 (pg. 162)
- Do Lab 11, Exercise 3, No. 5 (pg. 165)

In-Class Exercise

- Write a program that calculates the average of a group of test scores, where the lowest score in the group is dropped. It should use the following functions:
  - `getScore` – This function ask the user for a test score, store it in a reference parameter variable, and validate it. For input validation, do not accept test scores lower than 0 or higher than 100. This function should be called by `main()` once for each of the five scores to be entered.
  - `calcAverage` – This function calculates and display the average of the four highest score. This function should be called just once by `main()`, by should be passed the five scores.
  - `findLowest` – This function finds and returns the lowest of the five scores passed to it. It should be called by `calcAverage` function, which uses the function to determine which of the five scores to drop.