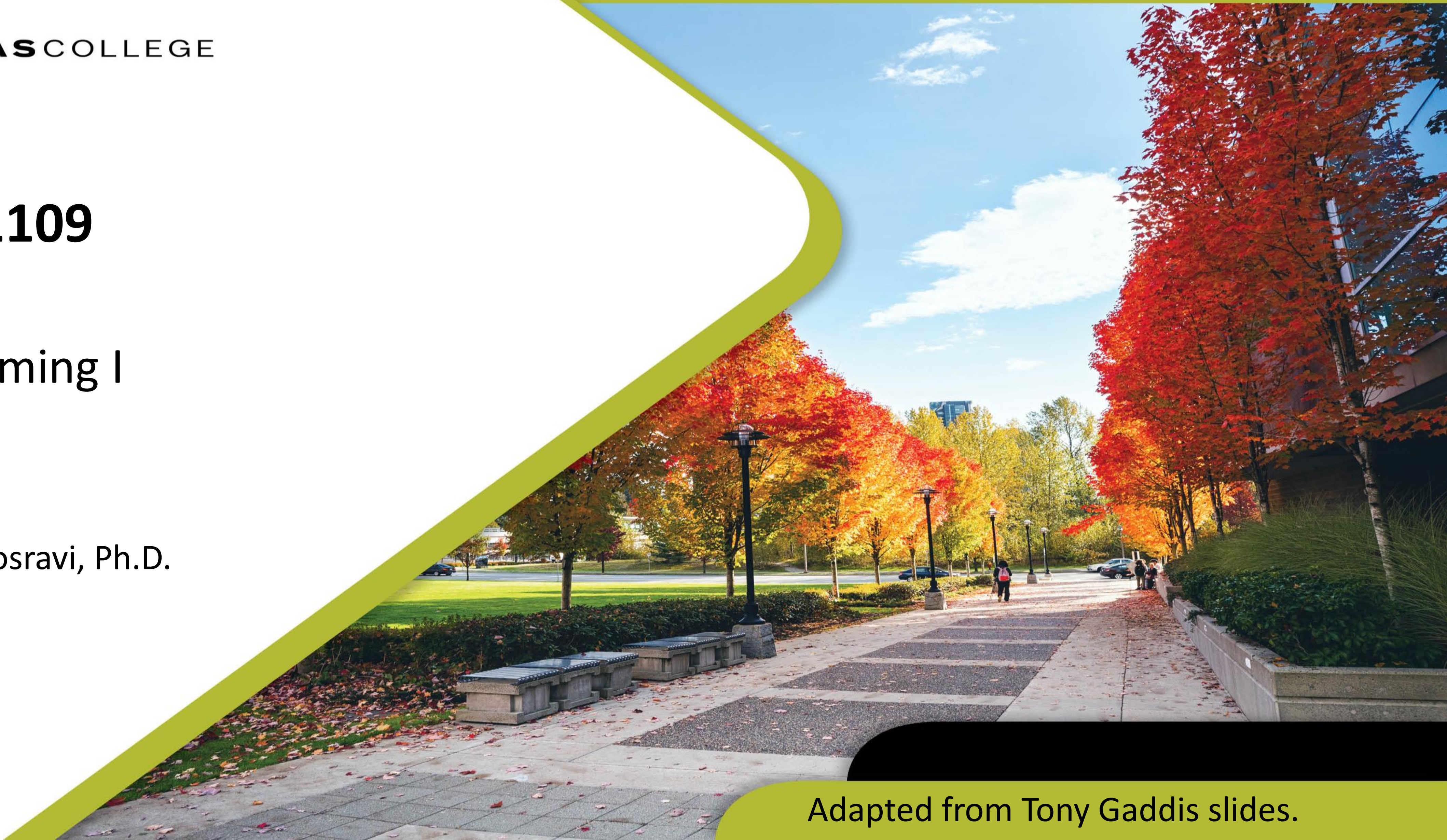
DOUGLAS COLLEGE

CMPT 1109

Programming I

Shahriar Khosravi, Ph.D.

Lecture 5



Adapted from Tony Gaddis slides.

Plan for Today

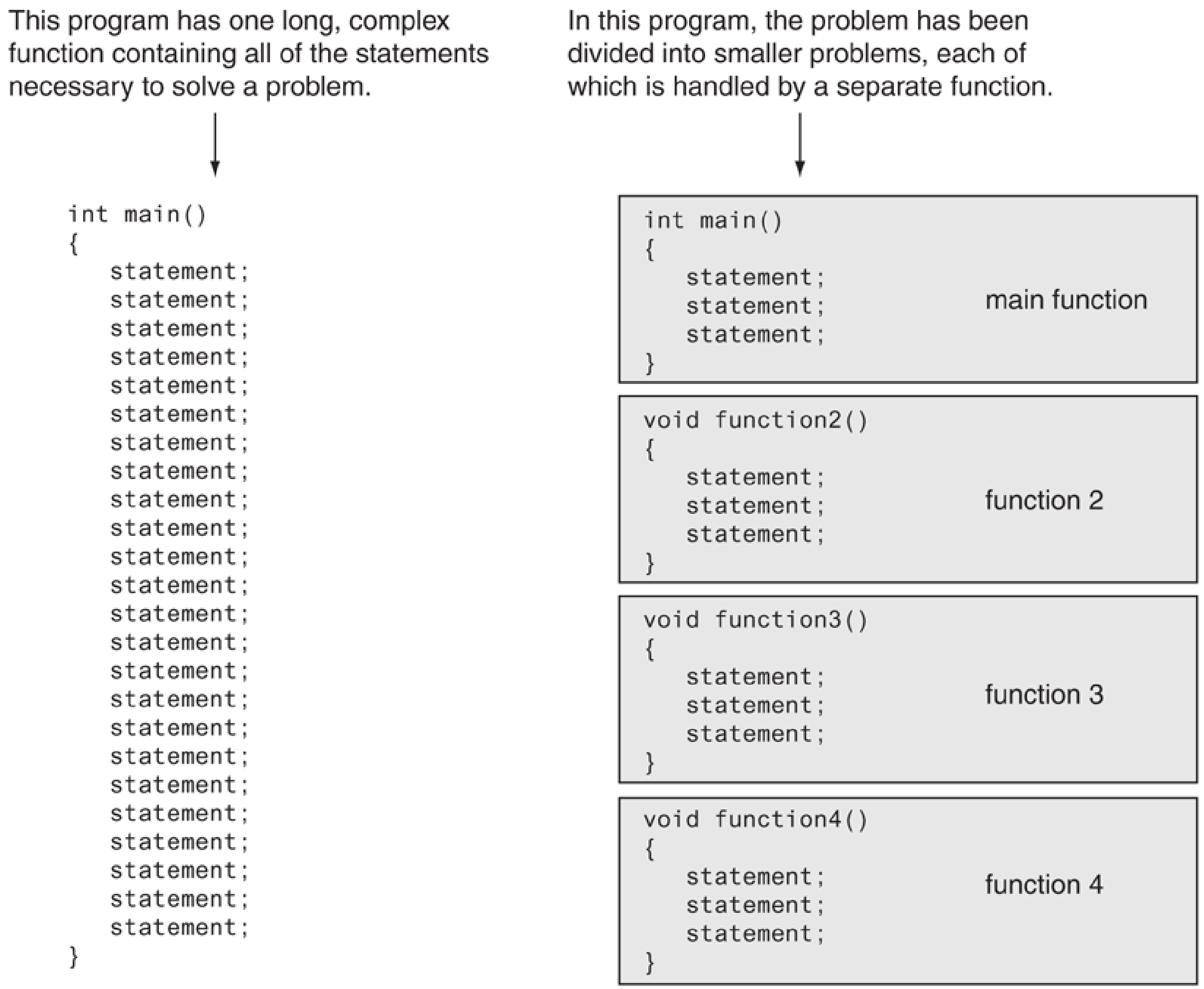
- Modular Programming
- Function Prototypes
- Sending Data into a Function
- Passing Data by Value
- The **`return`** Statement
- Returning a Value from a Function
- Returning a Boolean Value
- Local and Global Variables
- Static Local Variables
- Default Arguments
- Using Reference Variables as Parameters
- Overloading Functions
- The **`exit()`** Function

DOUGLAS COLLEGE

Modular Programming and Functions

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.

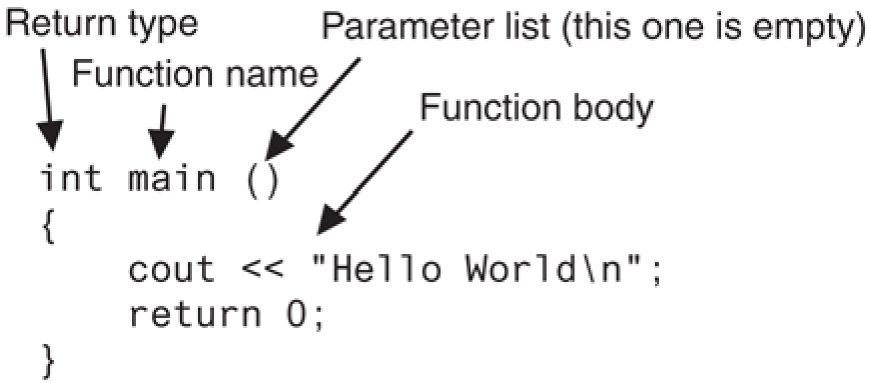
```
int main()
{
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;
    statement;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

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

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

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

Defining and Calling Functions

- **Function call**: statement causes a function to execute.
- **Function definition**: statements that make up a function.
- Definition includes:
  - **return type**: data type of the value that function returns to the part of the program that called 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 **{ }**



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

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

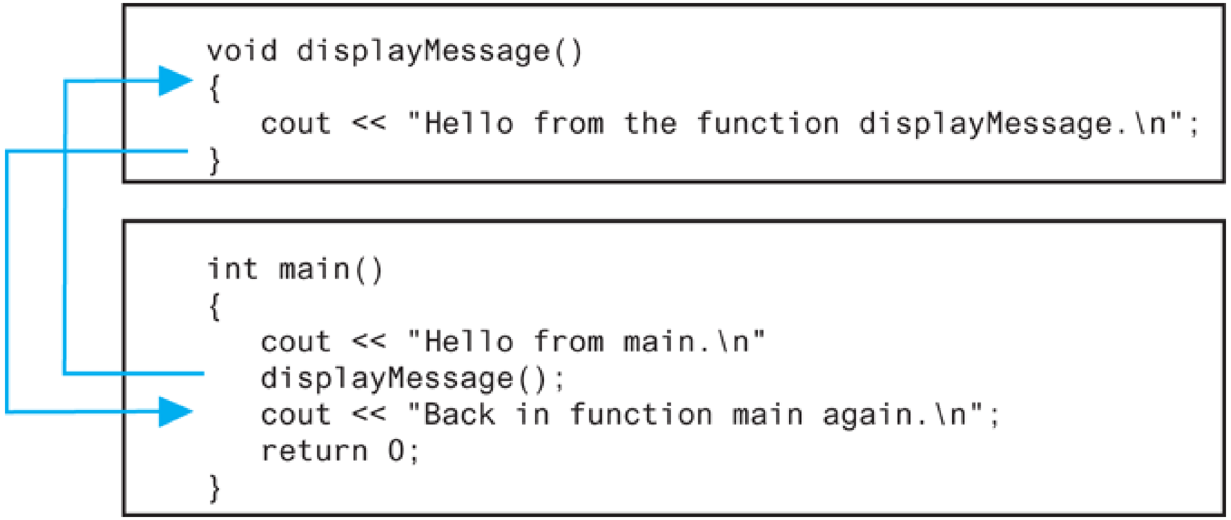
- To call a function, use the function name followed by **()** and **;**

```
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.

```
// This program has two functions: main and displayMessage
#include <iostream>
using namespace std;

//******************************************
// Definition of function displayMessage   *
// This function displays a greeting.      *
//******************************************

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

//******************************************
// Function main                           *
//******************************************

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

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 0;
}
```

```
// This program has two functions: main and displayMessage
#include <iostream>
using namespace std;

//*****************************************
// Definition of function displayMessage  *
// This function displays a greeting.     *
//*****************************************

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

//*****************************************
// Function main                          *
//*****************************************

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

Calling Functions

- The **`main()`** function 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

```cpp
// This program has three functions: main, deep, and deeper
#include <iostream>
using namespace std;

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

void deep()
{
    cout << "I am now inside the function deep.\n";
    deeper();   // Call function deeper
    cout << "Now I am back in deep.\n";
}

int main()
{
    cout << "I am starting in function main.\n";
    deep();     // Call function deep
    cout << "Back in function main again.\n";
    return 0;
}
```

Function Prototypes

- Ways to notify the compiler about a function before a call to the function:

  1. Place function definition **before** calling function’s definition.
  2. Use a function prototype:
     - Prototype: **`void printHeading();`**

- Although some programmers make **`main()`** the last function in the program, many prefer it to be first because it is the program’s starting **entry point**.

```cpp
// This program has three functions: main, first, and second.
#include <iostream>
using namespace std;

// Function Prototypes
void first();
void second();

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

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

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

DOUGLASCOLLEGE

Sending Data into A Function

Sending Data into a Function

- We can pass values into a function at time of call:

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

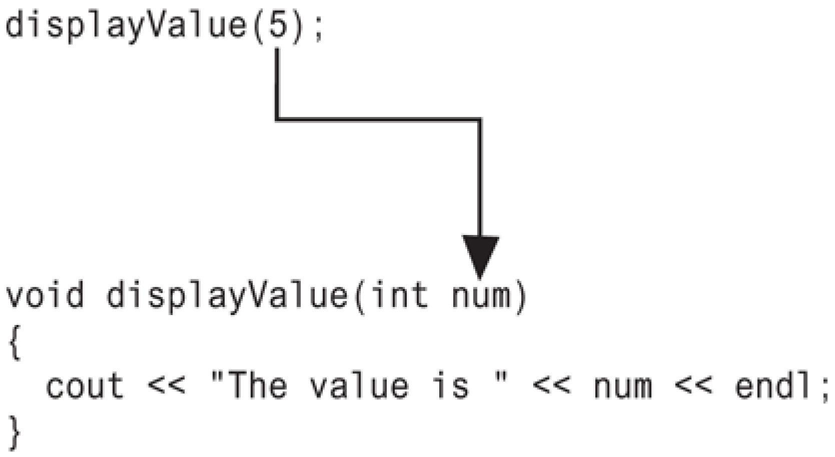
- Values passed to function are **arguments**.
- Variables in a function that hold the values passed as arguments are **parameters**.

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

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

Example

```
// This program demonstrates a function with a parameter.
#include <iostream>
using namespace std;

// Function Prototype
void displayValue(int);

int main()
{
    cout << "I am passing 5 to displayValue.\n";
    // Call displayValue with argument 5
    displayValue(5);
    cout << "Now I am back in main.\n";
    return 0;
}

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



```
displayValue(5);

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

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
```

- Value of argument is copied into parameter when the function is called.
- A **parameter's scope** is limited to the body of the function that uses it.
- Functions 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.

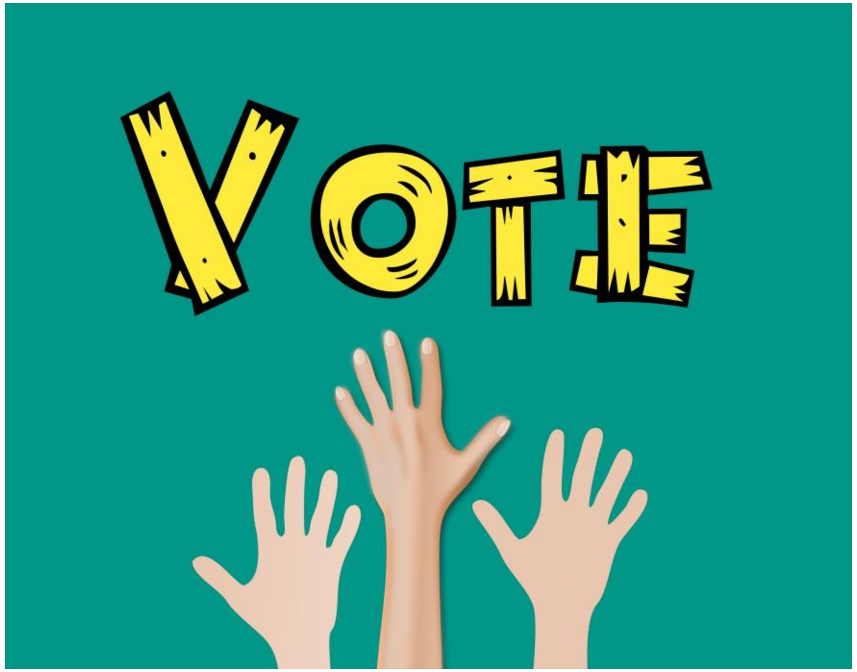
Poll 1 (Extra Credit)

Is the following function prototype syntactically legal?

```
void showSum(int num1, num2, num3);
```

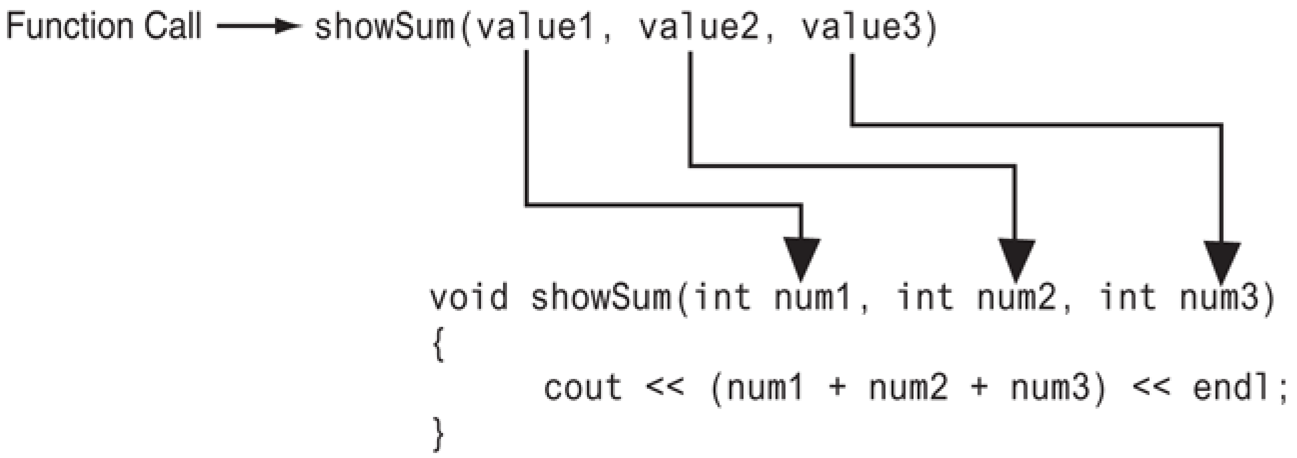
a) Yay!
b) Nay!

Please use the “Poll” window to participate for extra credit! One answer only please!



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.



```
showSum(value1, value2, value3)
```

```
void showSum(int num1, int num2, int num3)
{
      cout << (num1 + num2 + num3) << endl;
}
```

Passing Data by Value

- When an argument is passed into a parameter, **only a copy of the argument's value is passed**.
- **Changes to the parameter do not affect the original argument!**

```
// This program demonstrates that changes to a function parameter
// have no effect on the original argument.
#include <iostream>
using namespace std;

// Function Prototype
void changeMe(int);

int main()
{
    int number = 12;

    // Display the value in number.
    cout << "number is " << number << endl;

    // Call changeMe, passing the value in number
    // as an argument.
    changeMe(number);

    // Display the value in number again.
    cout << "Now back in main again, the value of ";
    cout << "number is " << number << endl;
    return 0;
}

void changeMe(int myValue)
{
    // Change the value of myValue to 0.
    myValue = 0;

    // Display the value in myValue.
    cout << "Now the value is " << myValue << endl;
}
```

Value-Returning Functions

The `return` Statement

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

```cpp
// This program uses a function to perform division. If division
// by zero is detected, the function returns.
#include <iostream>
using namespace std;

// Function prototype.
void divide(double, double);

int main()
{
    double num1, num2;

    cout << "Enter two numbers and I will divide the first\n";
    cout << "number by the second number: ";
    cin >> num1 >> num2;
    divide(num1, num2);
    return 0;
}

void divide(double arg1, double arg2)
{
    if (arg2 == 0.0)
    {
        cout << "Sorry, I cannot divide by zero.\n";
        return;
    }
    cout << "The quotient is " << (arg1 / arg2) << endl;
}
```

Returning a Value From a Function

- A function can **return** a value back to the statement that called the function.
- We have already seen the **pow()** function, which returns a value:

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

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

Return Type

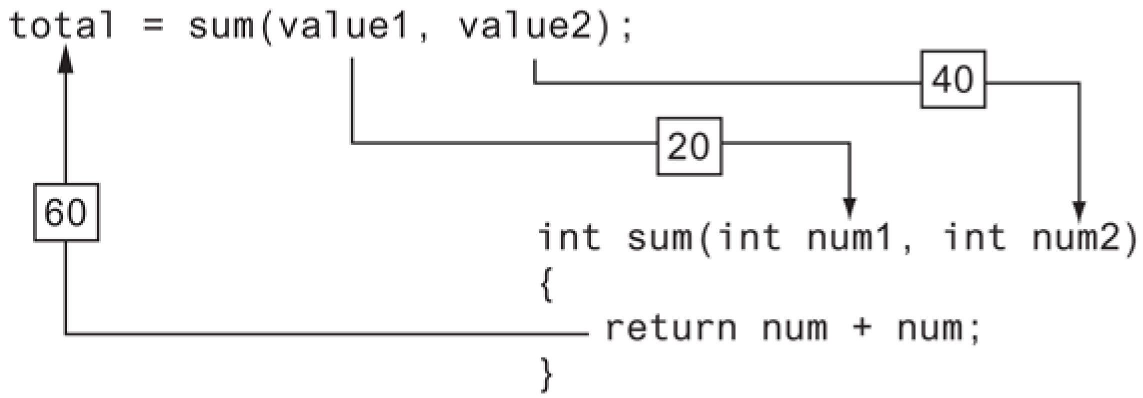
```
int sum(int num1, int num2)
{
    double result;
    result = num1 + num2;
    return result;
}
```

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

Can also return expressions

Example

```
// This program uses a function that returns a value.
#include <iostream>
using namespace std;

// Function prototype
int sum(int, int);

int main()
{
    int value1 = 20,   // The first value
        value2 = 40,   // The second value
        total;         // To hold the total

    // Call the sum function, passing the contents of
    // value1 and value2 as arguments. Assign the return
    // value to the total variable.
    total = sum(value1, value2);

    // Display the sum of the values.
    cout << "The sum of " << value1 << " and "
        << value2 << " is " << total << endl;
    return 0;
}

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



```
total = sum(value1, value2);
```

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

In-Class Exercise

Write a C++ program that calculates the area of a circle, has two functions in addition to `main()`. One of the functions is named `square()`, and it returns the square of any number passed to it as an argument. The program also has a function named `getRadius()`, which prompts the user to enter the circle's radius. The value entered by the user is returned from the function.

In-Class Exercise

```
#include <iostream>
#include <iomanip>
using namespace std;
//Function prototypes
double getRadius();
double square(double);

int main()
{
    const double PI = 3.14159; // Constant for pi
    double radius;             // To hold the circle's radius
    double area;               // To hold the circle's area

    // Set the numeric output formatting.
    cout << fixed << showpoint << setprecision(2);

    // Get the radius of the circle.
    cout << "This program calculates the area of ";
    cout << "a circle.\n";
    radius = getRadius();

    // Calculate the area of the circle.
    area = PI * square(radius);

    // Display the area.
    cout << "The area is " << area << endl;
    return 0;
}
double getRadius()
{
    double rad;

    cout << "Enter the radius of the circle: ";
    cin >> rad;
    return rad;
}

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

Returning a Boolean Value

- Functions can also simply **return true** or **false**.
- To do so, declare the **return** type in function prototype and heading as **bool**.
- The function body must contain **return** statement(s) that return **true** or **false**.
- The calling function can then use the returned value in a relational expression.

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

// Function prototype
bool isEven(int);

int main()
{
    int val;

    // Get a number from the user.
    cout << "Enter an integer and I will tell you ";
    cout << "if it is even or odd: ";
    cin >> val;

    // Indicate whether it is even or odd.
    if (isEven(val))
        cout << val << " is even.\n";
    else
        cout << val << " is odd.\n";
    return 0;
}

bool isEven(int number)
{
    bool status;

    if (number % 2 == 0)
        status = true;
    else
        status = false;
    return status;
}
```

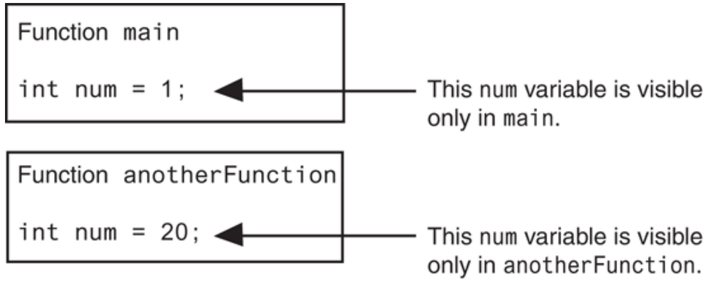
DOUGLAS COLLEGE

Local and Global Variables

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**.

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

void anotherFunction();

int main()
{
    int num = 1;    // Local variable

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

void anotherFunction()
{
    int num = 20;   // Local variable

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



Function main

int num = 1;

This num variable is visible only in main.

Function anotherFunction

int num = 20;

This num variable is visible only in anotherFunction.

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.

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

Local Variable Lifetime

- 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**.
- We should always avoid using global variables because they make programs difficult to debug.
- Any global items that we create should be **global constants**.

Example

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

void anotherFunction(); // Function prototype
int num = 2;            // Global variable

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

void anotherFunction()
{
    cout << "In anotherFunction, num is " << num << endl;
    num = 50;
    cout << "But, it is now changed to " << num << endl;
}
```

Example – example_01.cpp

DOUGLASCOLLEGE

Static Variables

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. Zero is the default initialization value.

```
// This program shows that local variables do not retain
// their values between function calls.
#include <iostream>
using namespace std;

// Function prototype
void showLocal();

int main()
{
    showLocal();
    showLocal();
    return 0;
}

void showLocal()
{
    int localNum = 5; // Local variable

    cout << "localNum is " << localNum << endl;
    localNum = 99;
}
```

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 and exist for the program's lifetime.
- **static** local variables are defined and initialized only the first time the function is executed. Zero is the default initialization value.
- Here, **statNum** is automatically initialized to **0**. Notice that it retains its value between function calls.

```cpp
// This program uses a static local variable.
#include <iostream>
using namespace std;

void showStatic(); // Function prototype

int main()
{
    // Call the showStatic function five times.
    for (int count = 0; count < 5; count++)
        showStatic();
    return 0;
}

void showStatic()
{
    static int statNum;

    cout << "statNum is " << statNum << endl;
    statNum++;
}
```

Initializing Static Local Variables

- If you do initialize a local **static** variable, the initialization only happens once.
- This is because initialization normally happens when the variable is created, and **static** local variables are only created once during the running of a program.

```
// This program shows that a static local variable is only
// initialized once.
#include <iostream>
using namespace std;

void showStatic(); // Function prototype

int main()
{
    // Call the showStatic function five times.
    for (int count = 0; count < 5; count++)
        showStatic();
    return 0;
}

void showStatic()
{
    static int statNum = 5;

    cout << "statNum is " << statNum << endl;
    statNum++;
}
```

DOUGLAS COLLEGE

Default Arguments

Default Arguments

- A **default argument** is an argument that is passed automatically to a parameter **if no argument is provided in the function call**.
- Must be a constant declared in prototype:

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

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

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

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

```
int getSum(int, int=0, int=0);// OK
int getSum(int, int=0, int);  // NOT OK
```

- 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);  // NOT OK
```

DOUGLAS COLLEGE

References

Using Reference Variables as Parameters

- **Reference variables** provide a mechanism that allows a function to work with the original argument from the function call, not a copy of the argument.
- **This allows the function to modify values stored in the calling function** and provides a way for the function to ‘return’ more than one value.
- A **reference variable** is an alias for another variable.
- It is defined with an ampersand (**&**), as shown below:

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

- **Changes to a reference variable are made to the variable it refers to**.
- We can use reference variables to implement **passing parameters by reference**.

```
void doubleNum(int& refVar)
{
    refVar *= 2;
}
```

The variable **refVar** is called “a reference to an **int**”.

Example

```cpp
// This program uses reference variables as function parameters.
#include <iostream>
using namespace std;

// Function prototypes. Both functions use reference variables
// as parameters.
void doubleNum(int&);
void getNum(int&);

int main()
{
    int value;

    // Get a number and store it in value.
    getNum(value);

    // Double the number stored in value.
    doubleNum(value);

    // Display the resulting number.
    cout << "That value doubled is " << value << endl;
    return 0;
}

void getNum(int& userNum)
{
    cout << "Enter a number: ";
    cin >> userNum;
}

void doubleNum(int& refVar)
{
    refVar *= 2;
}
```

Reference Variables

- Each reference parameter must contain **&**.
- Space between data type specification and **&** is unimportant.
- Must use **&** in both prototype and header (i.e., **the prototype and function header must be consistent as far as parameter types are considered**).
- Argument passed to reference parameter must be a variable – **cannot be an expression or constant**.
- Use when appropriate – avoid using references when an argument should not be changed by a function, or if the function needs to **`return`** only a single value.

DOUGLAS COLLEGE

Function Overloading

Overloading Functions

- Two or more functions may have the same name, as long as their parameter lists are different. This is known as **function overloading**.
- Function overloading can be used **to create functions that perform the same task, but take different parameter types or different number of parameters**.
- **The compiler will determine which version of function to call by argument and parameter lists**.
- **Note, a function's `return` type specification is not part of the signature!!!**

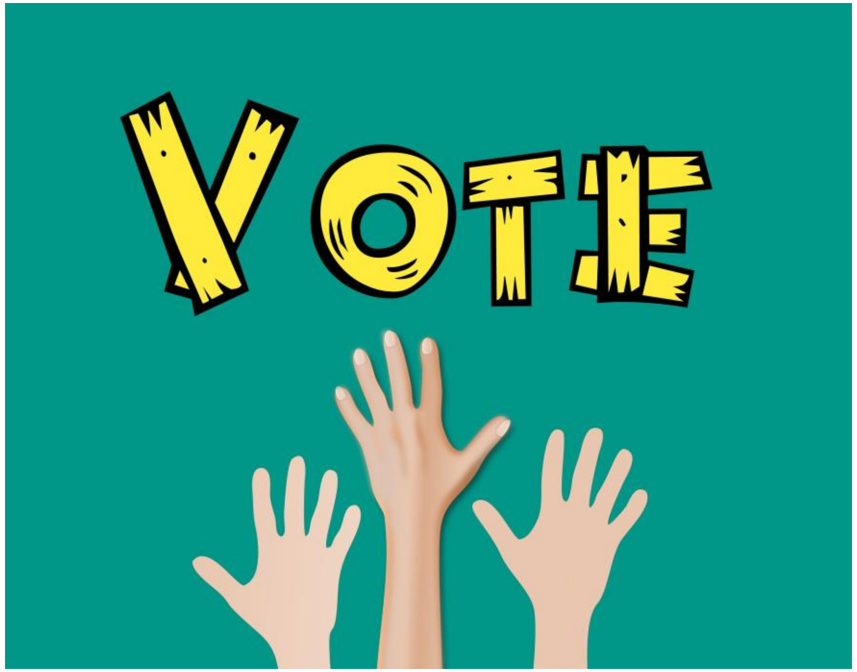
```
// This program uses overloaded functions.
#include <iostream>
#include <iomanip>
using namespace std;

// Function prototypes
int square(int);
double square(double);

int main()
{
    int userInt;
    double userFloat;

    // Get an int and a double.
    cout << fixed << showpoint << setprecision(2);
    cout << "Enter an integer and a floating-point value: ";
    cin >> userInt >> userFloat;

    // Display their squares.
    cout << "Here are their squares: ";
    cout << square(userInt) << " and " << square(userFloat);
    return 0;
}

int square(int number)
{
    return number * number;
}

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

Poll 2 (Extra Credit)

The following functions show a correct application of function overloading.

```
 int square(int number)
{
    return number * number
}

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

a) Yay!
b) Nay!



Please use the "Poll" window to participate for extra credit! One answer only please!

The `exit()` Function

- 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.
  - Typically used for abnormal termination of program.
  - Requires **`<cstdlib>`** header file.
- Example:

```
exit(0);
```

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

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

- Use it with caution since it unconditionally terminates your program..

Example

```cpp
// This program shows how the exit function causes a program
// to stop executing.
#include <iostream>
#include <cstdlib>   // Needed for the exit function
using namespace std;

void function();  // Function prototype

int main()
{
    function();
    return 0;
}

void function()
{
    cout << "This program terminates with the exit function.\n";
    cout << "Bye!\n";
    exit(EXIT_SUCCESS);
    cout << "This message will never be displayed\n";
    cout << "because the program has already terminated.\n";
}
```

Thank you.

DOUGLAS COLLEGE