CSE2050
Programming in a Second Language (C++)
Today’s lecture

- Modular programming
- Function prototypes
- Sending data to functions
- Global and local variables
- Static local variables
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
Chapter 6 Functions

Divided into several smaller problems that are easily solved. Figure 6-1 illustrates this idea by comparing two programs: one that uses a long complex function containing all of the statements necessary to solve a problem, and another that divides a problem into smaller problems, each of which are handled by a separate function.

Another reason to write functions is that they simplify programs. If a specific task is performed in several places in a program, a function can be written once to perform that task, and then be executed anytime it is needed. This benefit of using functions is known as code reuse because you are writing the code to perform a task once and then reusing it each time you need to perform the task.

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

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

CONCEPT: A function call is a statement that causes a function to execute. A function definition contains the statements that make up the function.

When creating a function, you must write its definition. All function definitions have the following parts:

- **Return type**: A function can send a value to the part of the program that executed it. The return type is the data type of the value that is sent from the function.

- **Name**: You should give each function a descriptive name. In general, the same rules that apply to variable names also apply to function names.

- **Parameter list**: The program can send data into a function. The parameter list is a list of variables that hold the values being passed to the function.

- **Body**: The body of a function is the set of statements that perform the function's operation. They are enclosed in a set of braces.

Figure 6-2 shows the definition of a simple function with the various parts labeled. The line in the definition that reads `int main()` is called the function header.

Functions

You already know that a function can return a value. The main function in all of the programs you have seen in this book is declared to return an int value to the operating system. The `return 0;` statement causes the value 0 to be returned when the main function finishes executing.

It isn't necessary for all functions to return a value, however. Some functions simply perform one or more statements which follow terminate. These are called void functions.

The `displayMessage` function, which follows is an example.

```c++
void displayMessage()
{
    cout << "Hello from the function displayMessage.\n";
    return 0;
}
```

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

Return type

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

6.2 Defining and Calling Functions

CONCEPT:
A function call is a statement that causes a function to execute. A function definition contains the statements that make up the function.

When creating a function, you must write its definition. All function definitions have the following parts:

- **Return type**: A function can send a value to the part of the program that executed it. The return type is the data type of the value that is sent from the function.
- **Name**: You should give each function a descriptive name. In general, the same rules that apply to variable names also apply to function names.
- **Parameter list**: The program can send data into a function. The parameter list is a list of variables that hold the values being passed to the function.
- **Body**: The body of a function is the set of statements that perform the function's operation. They are enclosed in a set of braces.

Figure 6-2 shows the definition of a simple function with the various parts labeled. The line in the definition that reads `int main()` is called the function header.

Functions

You already know that a function can return a value. The `main` function in all of the programs you have seen in this book is declared to return an `int` value to the operating system. The `return 0;` statement causes the value 0 to be returned when the `main` function finishes executing.

It isn't necessary for all functions to return a value, however. Some functions simply perform one or more statements which follows terminate. These are called void functions.

The `displayMessage` function, which follows is an example.

```cpp
void displayMessage() {
    cout << "Hello from the function displayMessage.\n";
    return 0;
}
```

Figure 6-2

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

| Return type: A function can send a value to the part of the program that executed it. The return type is the data type of the value that is sent from the function. |
| Name: You should give each function a descriptive name. In general, the same rules that apply to variable names also apply to function names. |
| Parameter list: The program can send data into a function. The parameter list is a list of variables that hold the values being passed to the function. |
| Body: The body of a function is the set of statements that perform the function's operation. They are enclosed in a set of braces. |

Figure 6-2 shows the definition of a simple function with the various parts labeled. The line in the definition that reads `int main()` is called the function header.

void Functions

You already know that a function can return a value. The `main` function in all of the programs you have seen in this book is declared to return an `int` value to the operating system. The `return 0;` statement causes the value 0 to be returned when the `main` function finishes executing.

It isn't necessary for all functions to return a value, however. Some functions simply perform one or more statements which follow and terminate. These are called `void` functions.

The `displayMessage` function, which follows, is an example.

```c
void displayMessage()
{
    cout << "Hello from the function displayMessage.\n";
    return 0;
}
```

```c
int main ()
{
    cout << "Hello World\n";
    return 0;
}
```
6.2 Defining and Calling Functions

CONCEPT:
A function call is a statement that causes a function to execute. A function definition contains the statements that make up the function. When creating a function, you must write its definition. All function definitions have the following parts:

- **Return type:** A function can send a value to the part of the program that executed it. The return type is the data type of the value that is sent from the function.
- **Name:** You should give each function a descriptive name. In general, the same rules that apply to variable names also apply to function names.
- **Parameter list:** The program can send data into a function. The parameter list is a list of variables that hold the values being passed to the function.
- **Body:** The body of a function is the set of statements that perform the function's operation. They are enclosed in a set of braces.

Figure 6-2 shows the definition of a simple function with the various parts labeled. The line in the definition that reads `int main()` is called the **function header**.

void Functions

You already know that a function can return a value. The main function in all of the programs you have seen in this book is declared to return an int value to the operating system. The `return 0;` statement causes the value 0 to be returned when the main function finishes executing.

It isn't necessary for all functions to return a value, however. Some functions simply perform one or more statements which follows terminate. These are called **void functions**.

The displayMessage function, which follows is an example.

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

void displayMessage()
The function's name is `displayMessage`. This name gives an indication of what the function does: It displays a message. You should always give functions names that reflect their purpose. Notice that the function's return type is `void`. This means the function does not return a value to the part of the program that executed it. Also notice the function has no return statement. It simply displays a message on the screen and exits.

Calling a Function

A function is executed when it is called. Function `main` is called automatically when a program starts, but all other functions must be executed by function call statements. When a function is called, the program branches to that function and executes the statements in its body. Let's look at Program 6-1, which contains two functions: `main` and `displayMessage`.

```cpp
#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;
}
```

Program Output

```
Hello from main.
Hello from the function displayMessage.
Back in function main again.
```
Flow of Control

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

int main()
{
    cout << "Hello from main.\n"
    displayMessage();
    cout << "Back in function main again.\n";
    return 0;
}
Functions may also be called in a hierarchical, or layered fashion. This is demonstrated by Program 6-4, which has three functions: `main`, `deep`, and `deeper`.

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

// Definition of function deeper
// This function displays a message.
void deeper()
{
    cout << "I am now inside the function deeper.\n";
}

// Definition of function deep
// This function displays a message.
void first()
{
    cout << "I am now inside the function first.\n";
}
void second()
{
    cout << "I am now inside the function second.\n";
}

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

- Modular programming
- **Function prototypes**
- Sending data to functions
- Global and local variables
- Static local variables
#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;
}

//*************************************
// Definition of function first. *
// This function displays a message. *
//*************************************
void first()
{
    cout << "I am now inside the function first.\n";
}

//*************************************
// Definition of function second. *
// This function displays a message. *
//*************************************
void second()
{
    cout << "I am now inside the function second.\n";
}
Today’s lecture

- Modular programming
- Function prototypes
- Sending data to functions
- Global and local variables
- Static local variables
Sending Data into a Function

• Can pass values into a function at time of call:
  \[ c = \text{pow}(a, b); \]

• Values passed to function are **arguments**

• Variables in a function that hold the values passed as arguments are **parameters**
#include <iostream>

using namespace std;

// Function Prototype
void showSum(int, int, int);

int main()
{
    int value1, value2, value3;

    // Get three integers.
    cout << "Enter three integers and I will display ";
    cout << "their sum: ";
    cin >> value1 >> value2 >> value3;

    // Call showSum passing three arguments.
    showSum(value1, value2, value3);
    return 0;
}

void showSum(int num1, int num2, int num3)
{
    cout << (num1 + num2 + num3) << endl;
}
In the function call in line 18, the variables `value1`, `value2`, and `value3` are passed as arguments: `showSum(value1, value2, value3);

When a function with multiple parameters is called, the arguments are passed to the parameters in order. This is illustrated in Figure 6-7.

The following function call will cause 5 to be passed into the `num1` parameter, 10 to be passed into `num2`, and 15 to be passed into `num3`:

`showSum(5, 10, 15);

However, the following function call will cause 15 to be passed into the `num1` parameter, 5 to be passed into `num2`, and 10 to be passed into `num3`:

`showSum(15, 5, 10);

**WARNING!**

Each parameter variable in a parameter list must have a data type listed before its name. For example, a compiler error would occur if the parameter list for the `showSum` function were defined as shown in the following header:

```cpp
void showSum(int num1, num2, num3) // Error!
```

A data type for all three of the parameter variables must be listed, as shown here:

```cpp
void showSum(int num1, int num2, int num3) // Correct
```

**NOTE:**

The function prototype must list the data type of each parameter.

**NOTE:**

Like all variables, parameters have a scope. The scope of a parameter is limited to the body of the function that uses it.

```cpp
void showSum(int num1, int num2, int num3) {
    cout << (num1 + num2 + num3) << endl;
}
```
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.
The `return` Statement

```c
int sum(int num1, int num2)
{
    int result;
    result = num1 + num2;
    return result;
}
```
When you call a value-returning function, you usually want to do something meaningful with the value it returns. Program 6-12 shows a function's return value being assigned to a variable. This is commonly how return values are used, but you can do many other things with them. For example, the following code shows a mathematical expression that uses a call to the `sum` function:

```
int x = 10, y = 15;
double average;
average = sum(x, y) / 2.0;
```

In the last statement, the `sum` function is called with `x` and `y` as its arguments. The function's return value, which is 25, is divided by 2.0. The result, 12.5, is assigned to `average`.

Here is another example:

```
int x = 10, y = 15;
cout << "The sum is " << sum(x, y) << endl;
```

This code sends the `sum` function's return value to `cout` so it can be displayed on the screen. The message "The sum is 25" will be displayed.

Remember, a value-returning function returns a value of a specific data type. You can use the function's return value anywhere that you can use a regular value of the same data type. This means that anywhere an `int` value can be used, a call to an `int` value-returning function can be used. Likewise, anywhere a `double` value can be used, a call to a `double` value-returning function can be used. The same is true for all other data types.

Let's look at another example. Program 6-13, which 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 `square` function is called in a mathematical statement. 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.
The `return` Statement

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

area = PI * square(radius);
```

Assuming the user has entered 10 as the radius, and this value is passed as an argument to the `square` function, the `square` function will return the value 100. Figure 6-12 illustrates how the value 100 is passed back to the mathematical expression in line 26. The value 100 will then be used in the mathematical expression.
Today’s lecture

- Modular programming
- Function prototypes
- Sending data to functions
- Global and local variables
- Static local variables
Local and Global Variables

#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;
}
Local and Global Variables

However, you can have a local variable or a parameter variable with the same name as a global variable, or a global constant. When you do, the name of the local or parameter variable shadows the name of the global variable or global constant. This means that the global variable or constant's name is hidden by the name of the local or parameter variable. For example, look at Program 6-20. This program has a global constant named `BIRDS`, set to 500. The `california` function has a local constant named `BIRDS`, set to 10000. When the program is executing in the `main` function, the global constant `BIRDS`, which is set to 500, is visible. The `cout` statement in lines 14 and 15 displays "In main there are 500 birds." (My apologies to folks living in Maine for the difference in spelling.) When the program is executing in the `california` function, however, the local constant `BIRDS` shadows the global constant `BIRDS`. When the `california` function accesses `BIRDS`, it accesses the local constant. That is why the `cout` statement in lines 27 and 28 displays "In california there are 10000 birds."

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

// Global constant.
const int BIRDS = 500;

// Function prototype
void california();

int main()
{
    cout << "In main there are " << BIRDS
         << " birds.\n";
    california();
    return 0;
}

//********************************************
// california function                       *
//********************************************

void california()
{
    const int BIRDS = 10000;
    cout << "In california there are " << BIRDS
         << " birds.\n";
}
```
Today’s lecture

- Modular programming
- Function prototypes
- Sending data to functions
- Global and local variables
- Static local variables
6.11 Static Local Variables

Static local variables are not destroyed when a function returns. They exist for the lifetime of the program, even though their scope is only the function in which they are defined.

Program 6-22 demonstrates some characteristics of static local variables:

In line 26 of Program 6-22, `statNum` is incremented in the `showStatic` function, and it retains its value between each function call. Notice that even though `statNum` is not explicitly initialized, it starts at zero. Like global variables, all static local variables are initialized to zero by default. (Of course, you can provide your own initialization value, if necessary.)

If you do provide an initialization value for a static local variable, the initialization only occurs 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. Program 6-23, which is a slight modification of Program 6-22, illustrates this point.

### Program 6-22

```cpp
#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++;
}
```

Program Output

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

A **Default argument** is an argument that is passed automatically to a parameter if the argument is missing on the function call.
Default parameters: must be declared in prototype

```cpp
void showArea(double = 20.0, double = 10.0);
```
Default parameters: declared in function header if no prototype

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

Show defaults in function header as comments

do double length; /* = 20.0*/;
do double width; /*= 10.08*/;
{
    double area = length * width;
    cout << "The area is " << area << endl;
}
Default parameters: setting default values for some parameters only

```c
void calcPay(int empNum, double payRate, double hours = 40.0);
```

// Illegal prototype
```c
void calcPay(int empNum, double hours = 40.0, double payRate);
```
// Illegal prototype
```c
void calcPay(double hours = 40.0, int empNum, double payRate);
```
Passing parameters by reference: reference variable

```cpp
// Function prototype
void doubleNum(int &);

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

int value = 4;
doubleNum(value);
cout << value << endl;
```
Overloading functions

» Two functions may have the same name as long as their (input) parameters lists are different

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

» The function is called, the compiler selects the appropriate function by comparing the types of the parameter list.
Overloading functions

» Convenient when we want to have multiple versions of a function

```c
int sum(int num1, int num2)
int sum(int num1, int num2, int num3)
int sum(int num1, int num2, int num3, int num4)
```