# FUNCTIONS

Chapter 6, Part 1

# TOPICS

Why Use Functions?

Declaring Functions: Function Prototypes

Calling Functions

- The Call Stack
- Calling Functions with Arguments

Reducing the Overhead of Function Calls

- References and Default Parameters
- Inline Functions
- Function Overloading
- Function Templates

# Why Use Functions?

# DIVIDE AND CONQUER!

Best way to develop complex programs is to use a modular approach

Why is using modules beneficial?

1.

2.

3.

## USING MODULES IS BENEFICIAL

1. Divide and conquer approach makes designing program easier

2. Reusable modules avoid problems with repeating same code in multiple places.

3. Easier to debug and maintain

# Program Components in C++

Typical C++ program is a combination of **user-defined classes and functions**, and the classes and functions in the **C++ Standard Library**

Many techniques for working with functions in C++

## NAMING A FUNCTION IN C++

Should perform a **single, well-defined task**

Name should describe that task in a meaningful way

Follow the verbNoun style:

**calculateAverage()**
**displayOutput()**
**countData()**

If a function can't easily be named using a verbNoun style, what could be the problem?

# DECLARING FUNCTIONS: FUNCTION PROTOTYPES

## VARIABLES MUST BE KNOWN TO COMPILER BEFORE USE VIA DECLARATION

Example:

```
int x{0};
y = x;
```

This would not compile if **y** has not been declared yet.

```
int x{0};
int y;
y = x;
```

This would compile, since **y** has been declared before use.

# Functions Also Need to Be Known to Compiler Before Use

```
int functionA (int x) {
    return sqrt(x);
}

void functionB () {
    cout << functionA(9);
}
```

If **functionA** appears **BEFORE functionB** in the source code file, then this is fine.

# Defining a Function Before It's Used

```
void functionB () {
    cout << functionA(9);
}

int functionA (int x) {
    return sqrt(x);
}
```

But if the order is changed, **functionA** is no longer known to **functionB**, and this would not compile.

# Function Prototypes Act Like Function Declarations

Describes function without revealing its implementation

Tells compiler the function's **name**, **return type**, and **parameters** (number, order, type)

Compiler **uses prototype** to check that:

- Function **definition matches**
- Function call uses **correct arguments** (number, order, and type)
- Function call **handles return value** correctly (if used)

Using prototypes, functions can then be defined in any order.

# REVIEW: PARTS OF A FUNCTION – DEFINITION

```
int findMax ( int x, int y )
{
   //statements
}
```

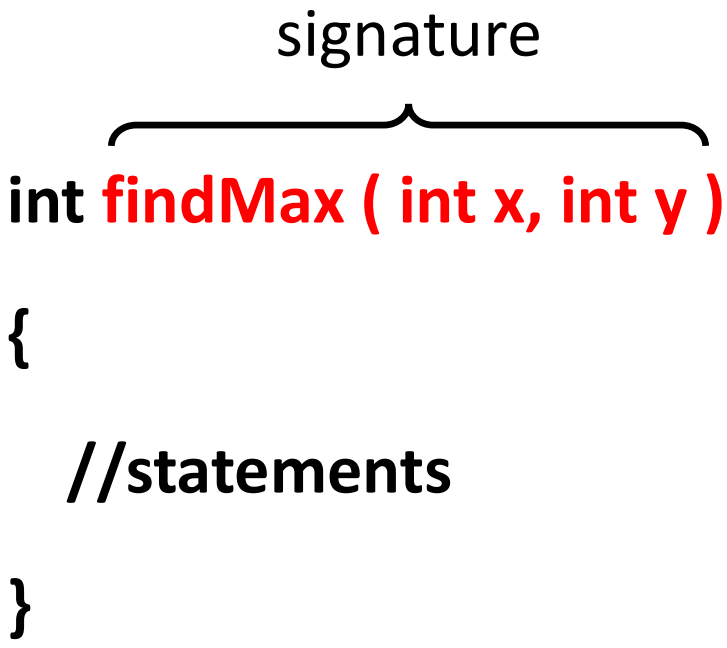
definition

# REVIEW: PARTS OF A FUNCTION – HEADER

```
int findMax ( int x, int y )
{
   //statements
}
```

header

# REVIEW: PARTS OF A FUNCTION – SIGNATURE



# FUNCTION PROTOTYPE FORMAT

General format:

**returnDataType functionSignature ;**

semicolon

Examples:

**int findMax ( int x, int y );**

**int findMax (int, int);**

Common to omit parameter names in prototypes

# Function Headers, Signatures and Prototypes

Return type **is** part of **header**

Return type **not** part of **signature**

Return type **is** part of **prototype**

## FUNCTION PROTOTYPES REQUIRED

Prototype must be located above where the function is used

Unless function defined before it's used

- But then the order of the functions in a file is important

**#include** statements

- Copy in prototypes for functions defined in that header

fig06_03.cpp – maximum function with a function prototype

- Function maximum takes three arguments & returns the largest

# SCOPE OF FUNCTIONS

What is the scope of a function?

Functions in same scope must have **unique signatures**

# CALLING FUNCTIONS

## TRANSFER OF CONTROL

Calling code (aka client code) **calls** a function

When function completes its task:

- **Always returns control** of program to calling code
- **May or may not return a result** back to calling code
- This is the **transfer of control**

Calling code does not need to know **how** function performed its task

- What is this concept called, and why is it a good thing?

# The Call Stack

# FUNCTION CALL STACK

When a program calls a function, the function must know how to return to the line of code where it was called

Address of the calling code is pushed onto the **function call stack**

If a series of function calls occurs, each successive return address is pushed onto the stack.

# A Stack is a Data Structure

Analogous to a pile of dishes

- A dish is placed on the pile at the top (pushing the dish onto the stack).
- A dish is removed from the pile from the top (popping the dish off the stack).

**Last-in, first-out (LIFO) data structure**

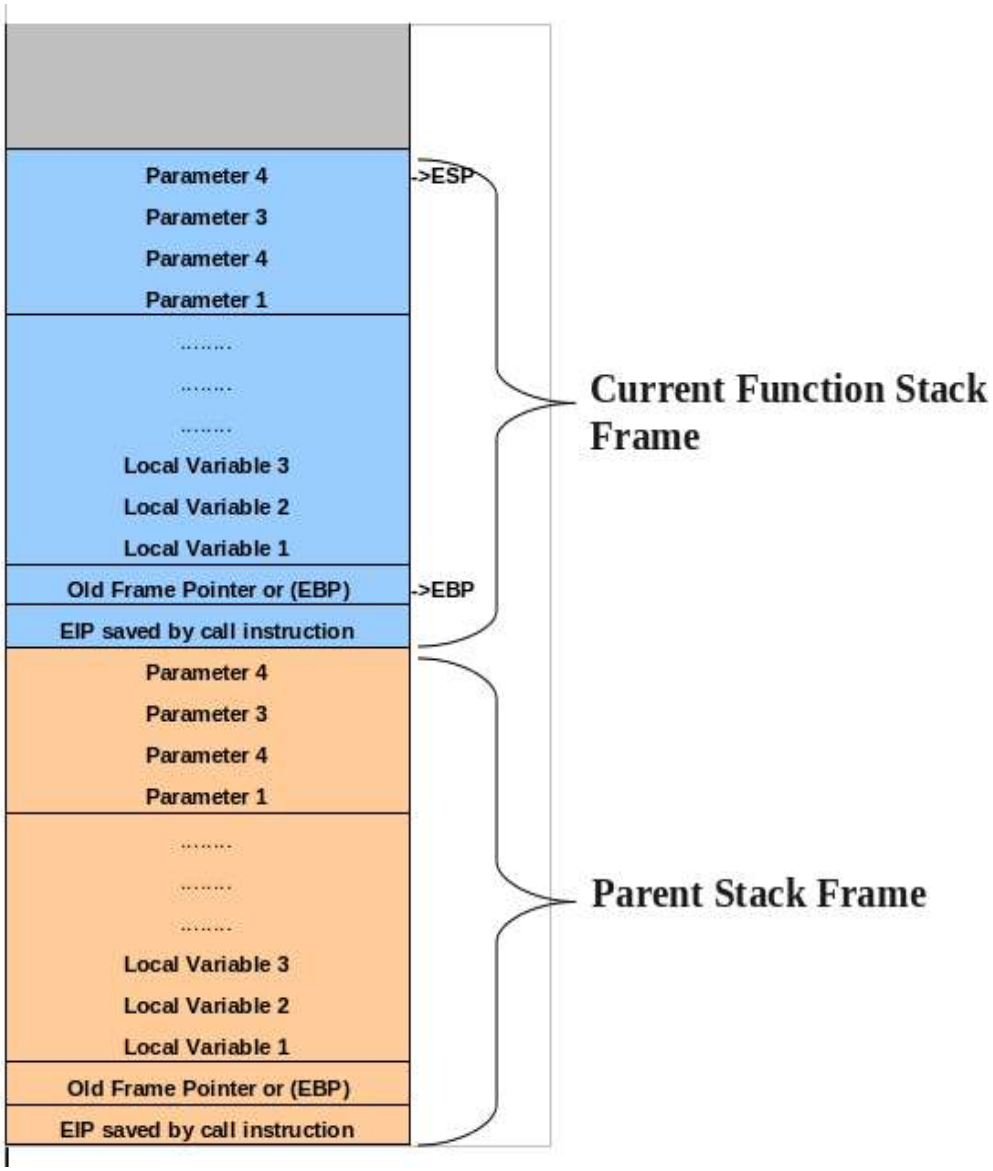
The last item pushed (added) on the stack is the first item popped (removed) from the stack.

## FUNCTION STACK FRAMES

Stack also contains memory for local variables used in each invocation of a function.

Stored as a portion of the program-execution stack known as the **activation record or stack frame** of the function call.

# EXAMPLE


Parameter 4
Parameter 3
Parameter 4
Parameter 1
->ESP
.....
.....
.....
Local Variable 3
Local Variable 2
Local Variable 1
Old Frame Pointer or (EBP)
->EBP
EIP saved by call instruction
Current Function Stack Frame
Parameter 4
Parameter 3
Parameter 4
Parameter 1
.....
.....
.....
Local Variable 3
Local Variable 2
Local Variable 1
Old Frame Pointer or (EBP)
EIP saved by call instruction
Parent Stack Frame

## STACK FRAMES

When a function is called, a stack frame for that function call is pushed onto stack.

When function returns to its caller, its stack frame is popped off the stack and those local variables are no longer known to the program.

If more function calls occur than can have their activation records stored on the program-execution stack, an error known as a stack overflow occurs.

Fig06_12.cpp – square function to demo stack frames

# CALLING FUNCTIONS WITH ARGUMENTS

# Review: Parameters vs. Arguments

## Parameters

- Variable declarations (data type and name)
- Located in the function signature and in the prototype

## Arguments

- Actual data passed when function executes
- Located in the function call

# REVIEW: CONSISTENCY

Argument types must be **consistent with** parameter types

**Consistent with**

- Data type of argument is equal to or smaller in bytes than data type of parameter
- Int argument → int parameter – ok
- Int argument → double parameter – ok
- Double argument → int parameter – ?

## ARGUMENT COERCION

Forcing arguments to the types specified by parameters

Works if the argument type is consistent with the parameter type

Relies on promotion rules and implicit conversion of fundamental types

# ARGUMENT & PARAMETER CONSISTENCY – EXAMPLES

| | Prototype | Function call | Result |
|---|---|---|---|
| 1 | int findMax ( int, int ); | findMax ( 10, 8 ); | |
| 2 | int findMax ( int, int ); | findMax ( 10.0, 8.0 ); | |
| 3 | int findMax ( int, int ); | findMax ( 10.0, 10.5 ); | |
| 4 | double findMax ( double, double ); | findMax ( 10, 8 ); | |
| 5 | double findMax ( double, double ); | findMax ( 10.0, 8.0 ); | |
| 6 | double findMax ( double, double ); | findMax ( 10.0, 10.5 ); | |

# ARGUMENT & PARAMETER CONSISTENCY – RESULT OF EXAMPLE 1

| | Prototype | Function call | Result |
|---|---|---|---|
| 1 | int findMax ( int, int ); | findMax ( 10, 8 ); | 10 |
| 2 | int findMax ( int, int ); | findMax ( 10.0, 8.0 ); | |
| 3 | int findMax ( int, int ); | findMax ( 10.0, 10.5 ); | |
| 4 | double findMax ( double, double ); | findMax ( 10, 8 ); | |
| 5 | double findMax ( double, double ); | findMax ( 10.0, 8.0 ); | |
| 6 | double findMax ( double, double ); | findMax ( 10.0, 10.5 ); | |

# ARGUMENT & PARAMETER CONSISTENCY – RESULT OF EXAMPLE 2

| | Prototype | Function call | Result |
|---|---|---|---|
| 1 | int findMax ( int, int ); | findMax ( 10, 8 ); | 10 |
| 2 | int findMax ( int, int ); | findMax ( 10.0, 8.0 ); | 10 |
| 3 | int findMax ( int, int ); | findMax ( 10.0, 10.5 ); | |
| 4 | double findMax ( double, double ); | findMax ( 10, 8 ); | |
| 5 | double findMax ( double, double ); | findMax ( 10.0, 8.0 ); | |
| 6 | double findMax ( double, double ); | findMax ( 10.0, 10.5 ); | |

# ARGUMENT & PARAMETER CONSISTENCY – RESULT OF EXAMPLE 3

| | Prototype | Function call | Result |
|---|---|---|---|
| 1 | int findMax ( int, int ); | findMax ( 10, 8 ); | 10 |
| 2 | int findMax ( int, int ); | findMax ( 10.0, 8.0 ); | 10 |
| 3 | int findMax ( int, int ); | findMax ( 10.0, 10.5 ); | 10 |
| 4 | double findMax ( double, double ); | findMax ( 10, 8 ); | |
| 5 | double findMax ( double, double ); | findMax ( 10.0, 8.0 ); | |
| 6 | double findMax ( double, double ); | findMax ( 10.0, 10.5 ); | |

# ARGUMENT & PARAMETER CONSISTENCY – RESULT OF EXAMPLE 4

| | Prototype | Function call | Result |
|---|---|---|---|
| 1 | int findMax ( int, int ); | findMax ( 10, 8 ); | 10 |
| 2 | int findMax ( int, int ); | findMax ( 10.0, 8.0 ); | 10 |
| 3 | int findMax ( int, int ); | findMax ( 10.0, 10.5 ); | 10 |
| 4 | double findMax ( double, double ); | findMax ( 10, 8 ); | 10.0 |
| 5 | double findMax ( double, double ); | findMax ( 10.0, 8.0 ); | |
| 6 | double findMax ( double, double ); | findMax ( 10.0, 10.5 ); | |

# ARGUMENT & PARAMETER CONSISTENCY – RESULT OF EXAMPLE 5

| | Prototype | Function call | Result |
|---|---|---|---|
| 1 | int findMax ( int, int ); | findMax ( 10, 8 ); | 10 |
| 2 | int findMax ( int, int ); | findMax ( 10.0, 8.0 ); | 10 |
| 3 | int findMax ( int, int ); | findMax ( 10.0, 10.5 ); | 10 |
| 4 | double findMax ( double, double ); | findMax ( 10, 8 ); | 10.0 |
| 5 | double findMax ( double, double ); | findMax ( 10.0, 8.0 ); | 10.0 |
| 6 | double findMax ( double, double ); | findMax ( 10.0, 10.5 ); | |

# ARGUMENT & PARAMETER CONSISTENCY – RESULT OF EXAMPLE 6

| | Prototype | Function call | Result |
|---|---|---|---|
| 1 | int findMax ( int, int ); | findMax ( 10, 8 ); | 10 |
| 2 | int findMax ( int, int ); | findMax ( 10.0, 8.0 ); | 10 |
| 3 | int findMax ( int, int ); | findMax ( 10.0, 10.5 ); | 10 |
| 4 | double findMax ( double, double ); | findMax ( 10, 8 ); | 10.0 |
| 5 | double findMax ( double, double ); | findMax ( 10.0, 8.0 ); | 10.0 |
| 6 | double findMax ( double, double ); | findMax ( 10.0, 10.5 ); | 10.5 |

# Reducing the Overhead of Function Calls

# Calling Functions is Expensive

Calling a function uses memory

- Stack frame of function itself, plus additional memory if arguments are passed

Several techniques to reduce this overhead

We will see:

- Calling functions using references and default parameters
- Inline functions
- Function overloading
- Function templates

# Using References and Default Parameters

# Two Ways to Pass Arguments to Functions

## Pass-by-value

- A **copy of the data** is sent to the function
- Original data not modified
- May be inefficient when working with very large items, like objects

## Pass-by-reference

- A **reference to the original data** is sent to the function, not a copy
- Can be a performance advantage
- But may not protect original data from modification
- Can do this in C++ with **reference parameters** or **pointers** (chapter 8)

## REFERENCE PARAMETER

**Alias** for its corresponding argument

General format:

***dataType*& *variableName***

Read from right to left

# Reference Parameter Example

Example:

**int& count**

**count** is a **reference** to an **int**

May also see **&** associated with variable name

- Better to attach it to data type – current best practice

# Passing Arguments By Value and By Reference

Fig06_17.cpp

Note lines 16 and 21

No way to tell from these calls alone whether the functions can modify those arguments

- Must also look at the function prototypes

Note that **squareByReference** doesn't need to return a value…

## DEFAULT ARGUMENTS

Can specify **default values** for parameters in the prototype

- Used when the function is called repeatedly with the same data

Must be the **rightmost (trailing) argument(s)**

- And all arguments to right of a default must also be default

If default arguments omitted when calling function, default values used instead

Fig06_18.cpp

Simplifies function calls, but specifying all arguments can be clearer

# INLINE FUNCTIONS

# INLINE FUNCTIONS CAN REDUCE OVERHEAD

Function calls require the overhead of maintaining the call stack

**Inline functions reduce overhead**

- Advises compiler to generate a copy of the function's body code anywhere the function is called
- Makes program larger

Use qualifier **inline** before return type in function declaration

## INLINE FUNCTIONS CAN REDUCE OVERHEAD, BUT…

Compilers may or may not implement it the way you think

Modern compilers optimize code better than you can anyway

Don’t need separate prototype if definition appears before call

- First line of function declaration acts as the prototype

Fig06_16.cpp

# FUNCTION OVERLOADING

# Function Overloading – Definition

Can define functions with the **same name** in the same scope

But they **must have different signatures**

Often used to perform similar tasks on different data types

Can make programs easier to read and understand

Fig06_21.cpp – overloaded square functions

# How Compiler Differentiates between Overloaded Functions

Distinguished by their **signatures**

- Function name and parameters (in order)
- NOT return value

**Name mangling** (or name decoration)

- Compiler encodes each function with the types of its parameters

**Type-safe linkage**

- Ensures that the proper overloaded function is called and that the types of its arguments conform to the types of the parameters

# FUNCTION TEMPLATES

## FUNCTION TEMPLATES – DEFINITION

Overloaded functions usually use different logic

Templates perform overloading more compactly and conveniently if **the logic is the same**

You write **one function template definition**

- And it can be used for any appropriate argument type

maximum.h (Fig 6.22)

Separate **function template specifications automatically generated** to handle calls using each type of data

# FUNCTION TEMPLATE DEFINITIONS – FORMAT

**template < typename T >**

or

**template < class T >**

The template parameter **T** becomes a placeholder for a fundamental or user-defined type

# Function Template Definitions – Example

**T maximum( T value1, T value2, T value3 )**

Placeholder **T**

- Defines return type, parameters, and local variables within body of function maximum

When maximum is called, the **type of the arguments provided** is substituted for **T** throughout the template definition

Templates are a means of **code generation**

Fig06_23.cpp – uses maximum for ints, doubles and chars

# End of Functions, Part 1