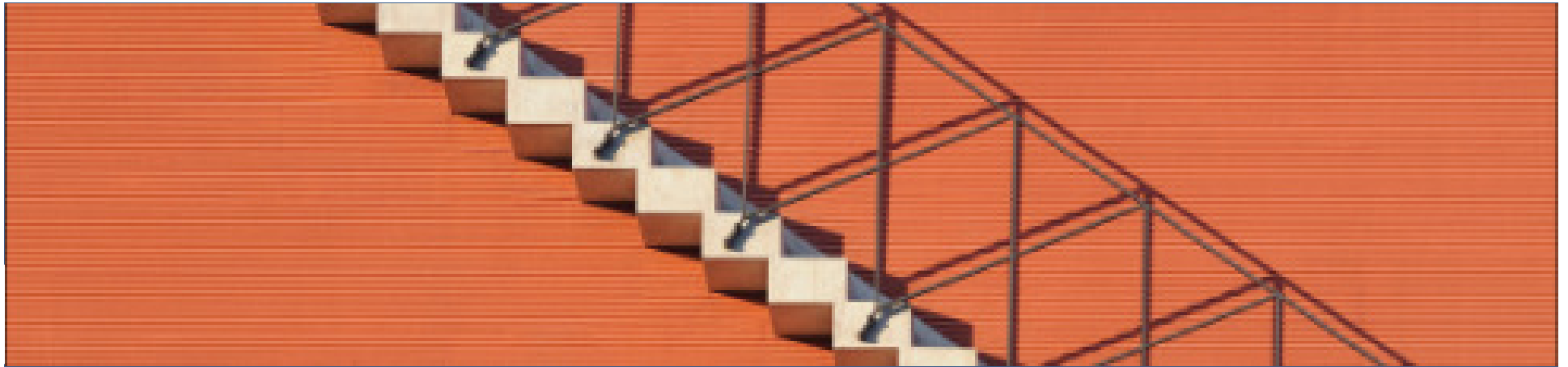




Module 2

Modular Design Using Functions

Objectives

In this chapter, you will:

- Learn about standard (predefined) functions
- Learn about user-defined functions
- Examine value-returning functions
- Construct and use a value-returning, user-defined function
- Construct and use void functions
- Understand value and reference parameters

Objectives (cont'd.)

- Learn about the scope of identifiers
- Understand local and global identifiers
- Learn about static variables
- Learn about function overloading
- Explore functions with default parameters

Predefined Functions

- In C++, a function is similar to that of a function in algebra
 - It has a name
 - It does some computation
- Some of the predefined mathematical functions are:

`sqrt(x)`

`pow(x, y)`

`floor(x)`

Predefined Functions (cont'd.)

- Predefined functions are organized into separate libraries
 - I/O functions are in `iostream` header
 - Math functions are in `cmath` header
- To use predefined functions, you must include the header file using an `include` statement

User-Defined Functions

- Value-returning functions: have a return type
 - Return a value of a specific data type using the `return` statement
- Void functions: do not have a return type
 - *Do not* use a `return` statement to return a value

Syntax: Value-Returning Function

- Syntax:

```
functionType functionName(formal parameter list)
{
    statements
}
```

- **functionType** is also called the data type or return type

**Let us now look
at workshop 2
question**



Syntax: Formal Parameter List

```
dataType identifier, dataType identifier, ...
```

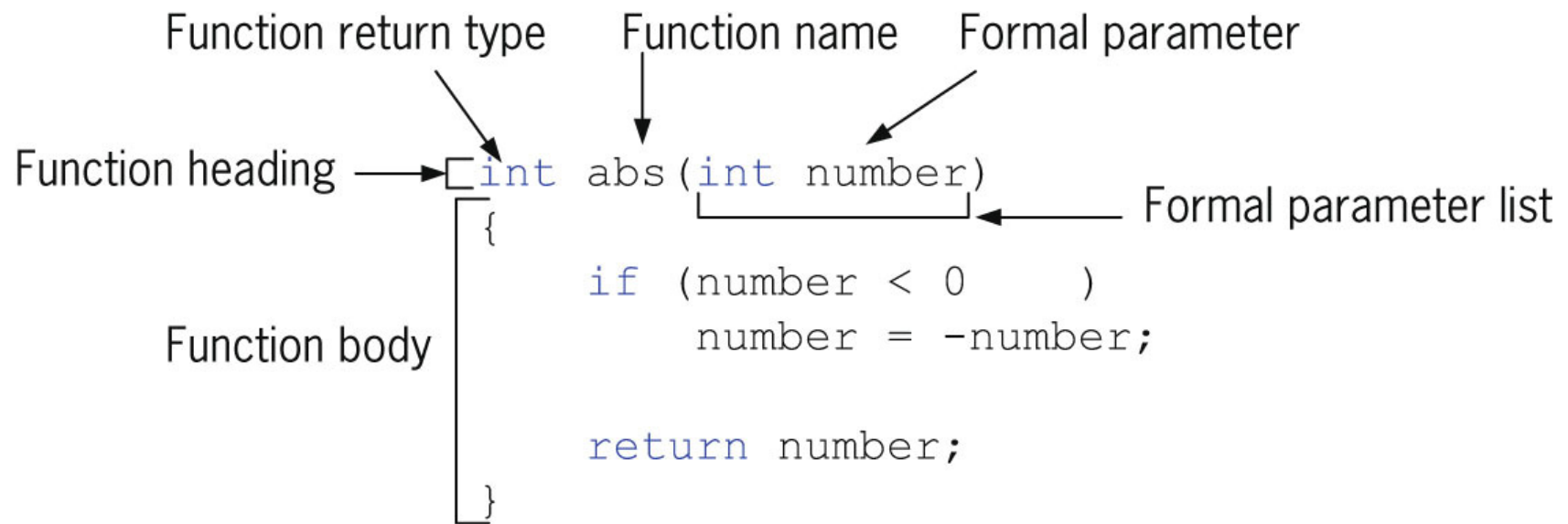


FIGURE 6-1 Various parts of the function `abs`

Function Call

- Syntax to call a value-returning function:

```
functionName(actual parameter list)
```

Syntax: Actual Parameter List

- Syntax of the actual parameter list:

```
expression or variable, expression or variable, ...
```

- Formal parameter list can be empty:

```
functionType functionName()
```

- A call to a value-returning function with an empty formal parameter list is:

```
functionName()
```

return Statement

- Function returns its value via the `return` statement
 - It passes this value outside the function

Syntax: `return` Statement

- Syntax:

```
return expr;
```

- In C++, `return` is a reserved word
- When a `return` statement executes
 - Function immediately terminates
 - Control goes back to the caller
- When a `return` statement executes in the function `main`, the program terminates

Syntax: return Statement (cont'd.)

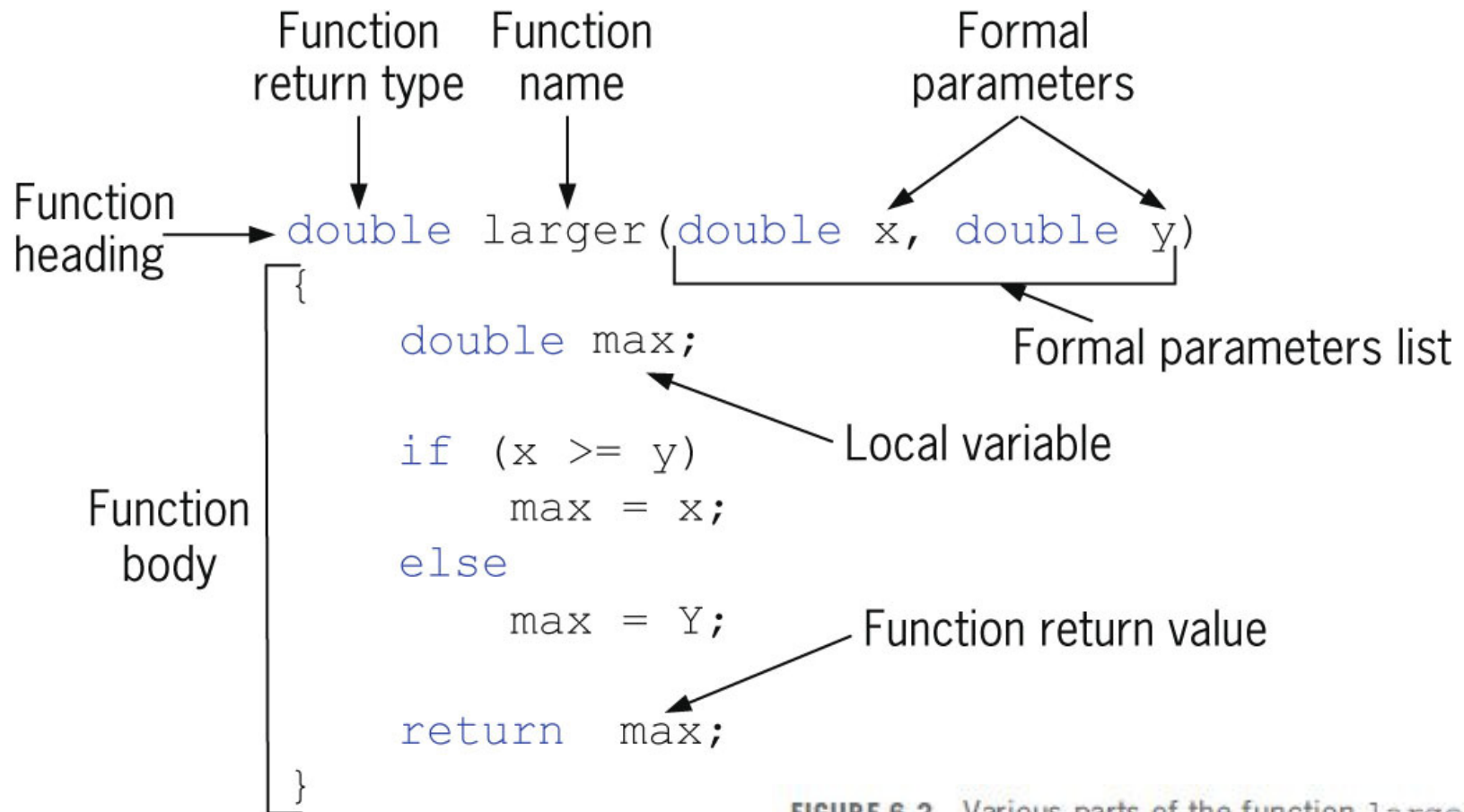


FIGURE 6-2 Various parts of the function `larger`

Function Prototype

- Function prototype: function heading without the body of the function
- Syntax:

```
functionType functionName (parameter list);
```

- Not necessary to specify the variable name in the parameter list
- Data type of each parameter must be specified

Void Functions

- User-defined void functions can be placed either before or after the function `main`
- If user-defined void functions are placed after the function `main`
 - The function prototype must be placed before the function `main`
- Void function does not have a return type
 - `return` statement without any value is typically used to exit the function early

Void Functions (cont'd.)

- Formal parameters are optional
- A call to a void function is a stand-alone statement
- Void function definition syntax:

```
void functionName(formal parameter list)
{
    statements
}
```

Void Functions (cont'd.)

- Formal parameter list syntax:

```
dataType& variable, dataType& variable, ...
```

- Function call syntax:

```
functionName(actual parameter list);
```

- Actual parameter list syntax:

```
expression or variable, expression or variable, ...
```

Void Functions (cont'd.)

- Value parameter: a formal parameter that receives a copy of the content of corresponding actual parameter
- Reference parameter: a formal parameter that receives the location (memory address) of the corresponding actual parameter

Value Parameters

- If a formal parameter is a value parameter:
 - The value of the corresponding actual parameter is copied into it
 - Formal parameter has its own copy of the data
- During program execution
 - Formal parameter manipulates the data stored in its own memory space

Reference Variables as Parameters

- If a formal parameter is a reference parameter
 - It receives the memory address of the corresponding actual parameter
- During program execution to manipulate data
 - Changes to formal parameter will change the corresponding actual parameter

Reference Variables as Parameters (cont'd.)

- Reference parameters are useful in three situations:
 - Returning more than one value
 - Changing the actual parameter
 - When passing the address would save memory space and time

Value and Reference Parameters and Memory Allocation

- When a function is called
 - Memory for its formal parameters and its local variables is allocated in the function data area
- For a value parameter, the actual parameter's value is copied into the formal parameter's memory cell
 - Changes to the formal parameter do not affect the actual parameter's value

Value and Reference Parameters and Memory Allocation (cont'd.)

- For a reference parameter, the actual parameter's address passes to the formal parameter
 - Both formal and actual parameters refer to the same memory location
 - During execution, changes made to the formal parameter's value permanently change the actual parameter's value

Reference Parameters and Value-Returning Functions

- Can also use reference parameters in a value-returning function
 - Not recommended
- By definition, a value-returning function returns a single value via `return` statement
- If a function needs to return more than one value, change it to a void function and use reference parameters to return the values

Static and Automatic Variables

- Automatic variable: memory is allocated at block entry and deallocated at block exit
 - By default, variables declared within a block are automatic variables
- Static variable: memory remains allocated as long as the program executes
 - Global variables declared outside of any block are static variables

Static and Automatic Variables (cont'd.)

- Can declare a static variable within a block by using the reserved word `static`
- Syntax:

```
static dataType identifier;
```

- Static variables declared within a block are local to the block
 - Have same scope as any other local identifier in that block

// static versus non-static (auto) variable

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

```
void autoVariable ();
void staticVariable ();
```

```
int main ()
{
    for (int i = 1; i <= 5; i++)
        autoVariable ();
    cout << endl;

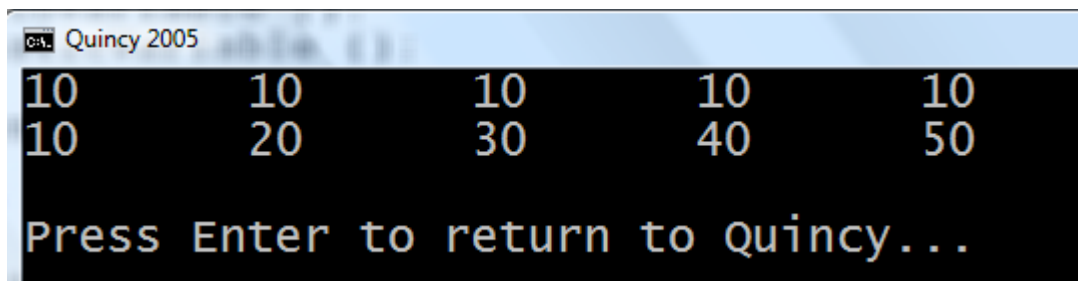
    for (int i = 1; i <= 5; i++)
        staticVariable ();
    cout << endl;
}
```

```
void autoVariable ()
{
    int x = 10;

    cout << x << "\t";
    x += 10;
}
```

```
void staticVariable ()
{
    static int x = 10;

    cout << x << "\t";
    x += 10;
}
```



```
Quincy 2005
10      10      10      10      10
10      20      30      40      50

Press Enter to return to quincy...
```

Function Overloading: An Introduction

- In a C++ program, several functions can have the same name
- Function overloading: creating several functions with the same name
- Function signature: the name and formal parameter list of the function
 - Does *not* include the return type of the function

Function Overloading (cont'd.)

- Two functions are said to have different formal parameter lists if both functions have either:
 - A different number of formal parameters
 - If the number of formal parameters is the same, but the data type of the formal parameters differs in at least one position
- Overloaded functions must have different function signatures

Function Overloading (cont'd.)

- The parameter list supplied in a call to an overloaded function determines which function is executed

```
int max (int, int);
```

```
int max (int, int, int);
```

Functions with Default Parameters

- In a function call, the number of actual and formal parameters must be the same
 - C++ relaxes this condition for functions with default parameters
- Can specify the value of a default parameter in the function prototype
- If you do not specify the value for a default parameter when calling the function, the default value is used

Functions with Default Parameters (cont'd.)

- All default parameters must be the rightmost parameters of the function
- If a default parameter value is not specified:
 - You must omit all of the arguments to its right
- Default values can be constants, global variables, or function calls
- Cannot assign a constant value as a default value to a reference parameter

// Default Parameters

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

```
int defaultPara (int = 10, int = 20);
```

```
int main ()
{
```

```
    cout << defaultPara () << endl;
    cout << defaultPara (100) << endl;
    cout << defaultPara (100, 200) << endl;
```

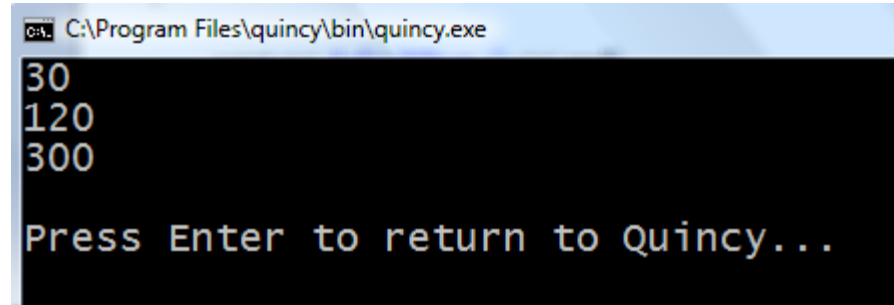
```
}
```

```
int defaultPara (int x, int y)
```

```
{
```

```
    return x + y;
```

```
}
```



A screenshot of a Windows command prompt window. The title bar shows the file path "C:\Program Files\quincy\bin\quincy.exe". The window has a black background with white text. It displays the output of the program: "30", "120", and "300" on three separate lines. Below the output, it says "Press Enter to return to Quincy...".

Use of rand () function

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
    srand(time(NULL));
    cout << rand() << endl << rand() << endl;
    return 0;
}
```

Factorial-Recursive Function

```
long factorial(int n)
{
    if (n == 0)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Base case:

if $n = 0$, $0! = 1$

General case

$n! = n * (n - 1)!$

**Let us now look
at workshop 2
question**

