William T. Doan

Another Trimesterly Review of CS 1436

Written for Dr. Brian Wescott Ricks and his 001 and 009 sections

October 2024

Dedication

This work is dedicated to Drs. Daniel Gibney and James Knox Willson III for their sage advice, guidance, and tutelage.

Contents

1

Managing the Execution of Multi-Function Programs and the Scope of Local Variables using the Stack

1.1 Introduction to a Stack

A stack is a last-in, first-out (LIFO) data structure.

- When an item is retrieved from a stack, the last item inserted (pushed) onto the stack is the first one retrieved (popped).
- Likewise, the first element inserted is the last one retrieved.

1.1.1 Visualizing the Stack

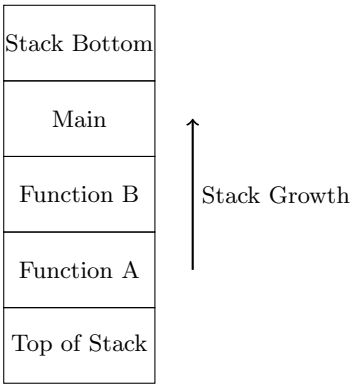

Fig. 1.1. Stack Frames during Function Calls

Exercises

1. Explain the difference between stack and heap memory allocation.

2. Draw a diagram illustrating the stack frames during the execution of nested function calls.
3. Why is it important to declare variables locally within functions?

1.2 When a Function is Called

- When a function is called, a **stack frame** containing the program's return address and the local variables of the function are pushed onto the stack.
- The stack frame of the currently executing function hides the stack frame of the calling function, including all the local variables of that function.
- The executing function has access to its local variables which are on the top frame of the stack.

1.3 When a Function Returns

- When the function ends execution, the return address is retrieved, and the stack frame is popped.
- Popping the stack destroys all the local variables of the function.
- The stack frame of the calling function is now on top of the stack, and all of its local variables are back in scope.

1.4 Global Variables

- Globals (constants and variables) are not stored on the stack.
- Global variables are created on the **heap**.
- The heap is a larger area of memory where access is less restrictive.

2

Making Decisions

2.1 Decision Statements

Decision statements (often called selection statements) allow us to specify alternate courses of execution based upon conditions that exist at run time.

In this chapter, we will learn about:

- Additional operators used in forming decisions:
  - Relational operators
  - Equality and inequality operators
  - Logical operators
  - The conditional operator
- Decision/selection statements:
  - `if`
  - `if/else`
  - `if/else if`
  - `switch`
  - The conditional statement

2.2 Relational Operators

Relational operators determine whether a specific relationship exists between two values.

Operator	Relationship Tested
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`==`	Equal to
`!=`	Not equal to

2.3 Relational Expressions

A relational expression is formed when two values or expressions are the operands of a relational operator.

Relational expressions have a value of data type `bool`. Recall `bool` is a small integer data type with two values: `true` (1) and `false` (0).

2.3.1 Examples

If `x` is 5 and `y` is 10, the following relational expressions have the value `true`:

- `x != y`
- `y > x`
- `x < y`
- `y >= x`
- `x <= y`

If `ch1` is `'W'` and `ch2` is `'M'`, the following relational expressions have the value `false`:

- `ch1 == ch2`
- `ch2 > ch1`
- `ch1 < ch2`
- `ch2 >= ch1`
- `ch1 <= ch2`

2.4 The if Statement

The `if` statement allows us to place a condition on the execution of a statement or block of statements.

2.4.1 Syntax

```
1 if (expression)
2 {
3     statement(s);
4 }
```

2.4.2 Example

```
if (score >= 70)
{
    passed = true;
}
```

Always use braces {} to enclose the block of statements, even if there is only one statement. This helps prevent errors and improves code readability.

2.4.3 Flowchart Example

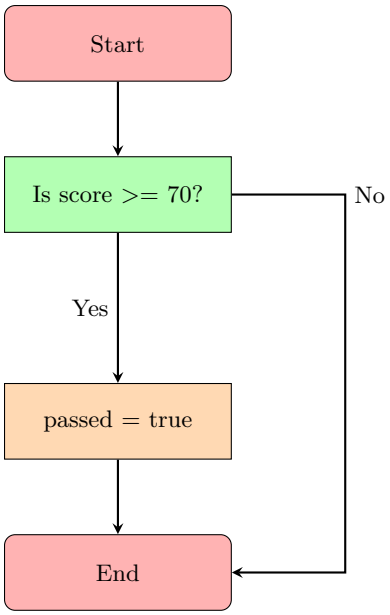

Fig. 2.1. Flowchart of an `if` Statement

2.5 The if/else Statement

The `if/else` statement provides two possible paths of execution. One for when the condition is true and another for when the condition is false.

2.5.1 Syntax

```
if (expression)
{
    statement(s); // if/true clause
}
else
{
    statement(s); // else/false clause
}
```

2.5.2 Example

```
if (hoursWorked <= 40.0)
{
    regularPay = hoursWorked * payRate;
    overtimePay = 0.0;
}
else // hoursWorked > 40
{
    regularPay = 40.0 * payRate;
    overtimePay = (hoursWorked - 40.0) * 1.5 * payRate;
}
```

2.6 Nested if Statements

An `if` or `if/else` statement can be nested inside another `if` statement.

2.6.1 Example

Suppose we are developing a banking program that determines whether a customer qualifies for a special interest rate on a loan. There are two conditions for qualification:

1. The customer must currently be employed.
2. The customer must have recently graduated from college (in the past two years).

```
if (isEmployed)
{
    if (yearsSinceGraduation <= 2)
    {
        cout << "You qualify for the special interest rate." << endl;
    }
}
```

2.7 Logical Operators

We can use logical operators to form more complex Boolean expressions from simpler expressions.

2.7.1 Operators

- **Logical NOT** (`!`): Reverses the truth of its operand.
- **Logical AND** (`&&`): Combines two expressions into one. Both sub-expressions must be true for the overall expression to be true.
- **Logical OR** (`||`): Combines two expressions into one. One or both sub-expressions must be true for the overall expression to be true.

2.7.2 Example of Logical AND

```
if (score >= 90 && score <= 100)
{
    grade = 'A';
}
```

2.7.3 Example of Logical OR

```
if (temperature < 0 || temperature > 100)
{
    cout << "Warning: Temperature is out of range!" << endl;
}
```

2.8 The switch Statement

The `switch` statement allows multi-way branching based on the value of an integer expression.

2.8.1 Syntax

```
switch (IntegerExpression)
{
    case ConstantExpression:
        statement(s);
        break;
    // more cases...
    default:
        statement(s);
}
```

2.8.2 Example

```
char grade = 'A';

switch (grade)
{
    case 'A':
    case 'a':
        cout << "Excellent work!" << endl;
        break;
    case 'B':
    case 'b':
        cout << "Good job!" << endl;
        break;
    default:
        cout << "Keep trying!" << endl;
}
```

3

Functions

3.1 Modular Programming

Functions are commonly used to break a problem down into small manageable pieces.

3.1.1 Advantages of Modular Programming

- Simplifies the process of writing programs
- Avoids redundancy
- Facilitates software reuse
- Improves the maintainability of programs

3.2 Defining and Calling Functions

When creating a function, you must write its definition. A function executes when it is called.

3.2.1 Function Definition

A function definition contains the statements that perform the task of the function.

3.2.2 Syntax

```
return_type function_name(parameter_list)
{
    // body of the function
}
```

3.3 Function Prototypes

A function prototype is a statement that declares a function, its return type, and the number and types of its parameters.

3.3.1 Syntax

```
return_type function_name(parameter_list);
```

3.3.2 Example

```
// Function prototype
void displayMessage();

// Function definition
void displayMessage()
{
    cout << "Hello from the function displayMessage." << endl;
}
```

3.4 Passing Arguments to Functions

When a function is called, the program may send values into the function.

3.4.1 Parameters and Arguments

- **Arguments**: Values passed to the function.
- **Parameters**: Variables in the function that receive the arguments.

3.4.2 Example

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

// Function call
displayValue(5);
```

3.5 Passing Multiple Arguments

Functions can accept multiple arguments, and they must be passed in the correct order.

3.5.1 Example

```
void showSum(int num1, int num2, int num3)
{
    int sum = num1 + num2 + num3;
    cout << "The sum is " << sum << endl;
}

// Function call
showSum(3, 4, 5);
```

3.6 Passing Data by Value

When an argument is passed to a function, it is passed by value by default.

- A copy of the argument's value is made for the function to use.
- Changes to the parameter do not affect the original argument.

3.6.1 Example

```
void changeMe(int myValue)
{
    myValue = 100;
    cout << "Inside changeMe, myValue is " << myValue << endl;
}

int main()
{
    int number = 12;
    cout << "Before calling changeMe, number is " << number << endl;
    changeMe(number);
    cout << "After calling changeMe, number is " << number << endl;
    return 0;
}
```

3.7 Returning a Value from a Function

A function can return a value back to the statement that called it.

3.7.1 Example

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

// Function call
int total = sum(5, 10);
```

3.8 Exercises

1. Write a function called `multiply` that takes two integers and returns their product.
2. Modify the `changeMe` function to use a reference parameter so that the original argument is changed.
3. Explain the difference between passing an argument by value and by reference.

3.9 Returning a Boolean Value

A function can return a boolean value, which is useful for decision-making.

3.9.1 Example

```
bool isEven(int number)
{
    return (number % 2 == 0);
}

// Function call
if (isEven(4))
{
    cout << "The number is even." << endl;
}
```

3.10 Default Arguments

A default argument is an argument that is passed automatically to a parameter when an argument is not provided in the function call.

3.10.1 Example

```
void displayMessage(string message = "Hello, World!")
{
    cout << message << endl;
}

// Function calls
displayMessage(); // Outputs: Hello, World!
displayMessage("Hi there!"); // Outputs: Hi there!
```

3.11 Using Reference Variables as Parameters

Reference variables allow a function to modify the argument passed to it.

3.11.1 Example

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

int main()
{
    int value = 5;
    doubleNumber(value);
    cout << "Value is now " << value << endl; // Outputs: Value is now
        10
    return 0;
}
```

4

Input Validation and Menus

4.1 Validating User Input

Input validation involves inspecting input data given to a program by the user and determining if it is acceptable.

- Ensure numbers/characters are within the range of acceptable values.
- Check that data entered is reasonable.
- Validate that a valid menu choice was selected.
- Prevent division by zero and other invalid operations.

4.1.1 Example

```
int getValidatedInput()
{
    int number;
    cout << "Enter a number between 1 and 10: ";
    cin >> number;

    while (number < 1 || number > 10)
    {
        cout << "Invalid input. Please enter a number between 1 and 10:
             ";
        cin >> number;
    }

    return number;
}
```

4.2 Menus

A menu-driven program allows the user to determine the course of action by selecting from a list of choices.

4.2.1 Menu-Driven Program Organization

1. Display the menu as a list of numbered or lettered choices.
2. Prompt the user to make a selection.
3. Test user input to determine which menu choice was selected.
4. Execute code that performs actions for the specific menu choice.

4.2.2 Example

```
void displayMenu()
{
    cout << "1. Add new record" << endl;
    cout << "2. Delete record" << endl;
    cout << "3. View record" << endl;
    cout << "4. Exit" << endl;
}

int main()
{
    int choice;
    do
    {
        displayMenu();
        cout << "Enter your choice: ";
        cin >> choice;

        switch (choice)
        {
            case 1:
                // Code to add new record
                break;
            case 2:
                // Code to delete record
                break;
            case 3:
                // Code to view record
                break;
            case 4:
                cout << "Exiting the program." << endl;
                break;
            default:
                cout << "Invalid choice. Please try again." << endl;
```

```
        }
    } while (choice != 4);

    return 0;
}
```

5

Overloading Functions and Stubs

5.1 Overloading Functions

C++ allows you to have multiple functions with the same name as long as their parameter lists are different.

5.1.1 Example

```
int max(int a, int b)
{
    return (a > b) ? a : b;
}

double max(double a, double b)
{
    return (a > b) ? a : b;
}

int max(int a, int b, int c)
{
    return max(max(a, b), c);
}
```

5.1.2 Usage

```
int x = max(3, 7); // Calls max(int, int)
double y = max(5.5, 2.3); // Calls max(double, double)
int z = max(1, 2, 3); // Calls max(int, int, int)
```

5.2 Stubs and Drivers

Stubs and drivers are helpful tools in developing, testing, and debugging programs that use functions.

5.2.1 Stubs

A stub is a dummy placeholder function that is called instead of the actual function it represents.

5.2.2 Example of a Stub

```
void complexFunction()
{
    // TODO: Implement this function
    cout << "complexFunction() called." << endl;
}
```

5.2.3 Drivers

A driver is a function that tests another function by calling it with test data.

5.2.4 Example of a Driver

```
void testIsEven()
{
    cout << "Testing isEven() function:" << endl;
    for (int i = -1; i <= 1; ++i)
    {
        cout << "isEven(" << i << ") = " << boolalpha << isEven(i) <<
            endl;
    }
}
```

6

Additional Topics

6.1 Comparing Characters and Strings

6.1.1 Comparing Characters

When two characters are compared, their ASCII values are compared.

6.1.2 Example

```
char letter1 = 'A';
char letter2 = 'B';

if (letter1 < letter2)
{
    cout << letter1 << " comes before " << letter2 << " in ASCII." <<
        endl;
}
```

6.1.3 Comparing Strings

C++ string objects can be compared using relational operators.

6.1.4 Example

```
string name1 = "Alice";
string name2 = "Bob";

if (name1 < name2)
{
    cout << name1 << " comes before " << name2 << " alphabetically." <<
         endl;
}
```

6.2 The Conditional Operator

The conditional operator `?:` provides a shorthand method of expressing a simple `if/else` statement.

6.2.1 Syntax

```
expression1 ? expression2 : expression3;
```

6.2.2 Example

```
int max = (a > b) ? a : b;
```