5.1 — Control flow introduction

BY ALEX ON JUNE 20TH, 2007 | LAST MODIFIED BY ALEX ON APRIL 13TH, 2018

When a program is run, the CPU begins execution at the top of main(), executes some number of statements, and then terminates at the end of main(). The sequence of statements that the CPU executes is called the program's **execution path** (or path, for short). Most of the programs you have seen so far have been **straight-line programs**. Straight-line programs have **sequential flow** -- that is, they take the same path (execute the same statements) every time they are run (even if the user input changes).

However, often this is not what we desire. For example, if we ask the user to make a selection, and the user enters an invalid choice, ideally we'd like to ask the user to make another choice. This is not possible in a straight-line program. Alternatively, there are cases where we need to do something a number of times, but we don't know how many times at compile time. For example, if we wanted to print all of the integers from 0 to some number the user entered, we couldn't do that until we know what number the user entered.

Fortunately, C++ provides **control flow statements** (also called *flow control statements*), which allow the programmer to change the CPU's path through the program. There are quite a few different types of control flow statements, so we will cover them briefly here, and then in more detail throughout the rest of the chapter.

Halt

The most basic control flow statement is the **halt**, which tells the program to quit running immediately. In C++, a halt can be accomplished through use of the exit() function that is defined in the cstdlib header. The exit function takes an integer parameter that is returned to the operating system as an exit code, much like the return value of main().

Here is an example of using exit():

```cpp
#include <cstdlib> // needed for exit()
#include <iostream>

void cleanup
{
    // code here to do any kind of cleanup required
}

int main()
{
    std::cout << 1;
    cleanup();

    exit(0); // terminate and return 0 to operating system

    // The following statements never execute
    std::cout << 2;
    return 0;
}
```

Note that exit() works no matter what function it's called from (even a function other than main). Also note that exit() terminates the program immediately with minimal cleanup. Therefore, before calling exit(), you should consider whether any manual cleanup is required (e.g. saving the user's game to disk).

Most often, exit() is used to immediately terminate the program when some catastrophic, unrecoverable error occurs.

Jumps

The next most basic flow control statement is the jump. A **jump** unconditionally causes the CPU to jump to another statement. The *goto*, *break*, and *continue* keywords all cause different types of jumps -- we will discuss the difference between these in upcoming sections.

Function calls also cause jump-like behavior. When a function call is executed, the CPU jumps to the top of the function being called. When the called function ends, execution returns to the statement after the function call.

Conditional branches

A **conditional branch** is a statement that causes the program to change the path of execution based on the value of an expression. The most basic conditional branch is an *if statement*, which you have seen in previous examples. Consider the following program:

```
int main()
{
    // do A
    if (expression)
        // do B
    else
        // do C

    // do D
}
```

This program has two possible paths. If expression evaluates to true, the program will execute A, B, and D. If expression evaluates to false, the program will execute A, C, and D. As you can see, this program is no longer a straight-line program -- its path of execution depends on the value of expression.

The *switch* keyword also provides a mechanism for doing conditional branching. We will cover if statements and switch statements in more detail in an upcoming section.

Loops

A **loop** causes the program to repeatedly execute a series of statements until a given condition is false. For example:

```
int main()
{
    // do A
    // loop on B, 0 or more times
    // do C
}
```

This program might execute as ABC, ABBC, ABBBC, ABBBBC, or even AC. Again, you can see that this program is no longer a straight-line program -- its path of execution depends on how many times (if any) the looped portion executes.

C++ provides 3 types of loops: *while*, *do while*, and *for* loops. C++11 added support for a new kind of loop called a *for each* loop. We will discuss loops at length toward the end of this chapter, except for the *for each* loop, which we'll discuss a little later.

Exceptions

Finally, **exceptions** offer a mechanism for handling errors that occur in a function. If an error occurs in a function that the function cannot handle, the function can trigger an exception. This causes the CPU to jump to the nearest block of code that handles exceptions of that type.

Exception handling is a fairly advanced feature of C++, and is the only type of control flow statement that we won't be discussing in this section. We discuss exceptions in chapter 14.

Conclusion

Using program flow statements, you can affect the path the CPU takes through the program and control under what conditions it will terminate. Prior to this point, the number of things you could have a program do was fairly limited. Being able to control the flow of your program makes any number of interesting things possible, such as displaying a menu repeatedly until the user makes a valid choice, printing every number between x and y, or determining the factors of a number.

Once you understand program flow, the things you can do with a C++ program really open up. No longer will you be restricted to toy programs and academic exercises -- you will be able to write programs that have real applications. This is where the real fun begins. So let's get to it!

5.2 -- If statements

Index