8.12 — Halts (exiting your program early)

The last category of flow control statement we'll cover in this chapter is the halt. A **halt** is a flow control statement that terminates the program. In C++, halts are implemented as functions (rather than keywords), so our halt statements will be function calls.

Let's take a brief detour, and recap what happens when a program exits normally. When the `main()` function returns (either by reaching the end of the function, or via a `return statement`), a number of different things happen.

First, because we're leaving the function, all local variables and function parameters are destroyed (as per usual).

Next, a special function called `std::exit()` is called, with the return value from `main()` (the `status code`) passed in as an argument. So what is `std::exit()`?

The std::exit() function

`std::exit()` is a function that causes the program to terminate normally. **Normal termination** means the program has exited in an expected way. Note that the term `normal termination` does not imply anything about whether the program was successful (that's what the `status code` is for). For example, let's say you were writing a program where you expected the user to type in a filename to process. If the user typed in an invalid filename, your program would probably return a non-zero `status code` to indicate the failure state, but it would still have a `normal termination`.

`std::exit()` performs a number of cleanup functions. First, objects with static storage duration are destroyed. Then some other miscellaneous file cleanup is done if any files were used. Finally, control is returned back to the OS, with the argument passed to `std::exit()` used as the `status code`.

Calling std::exit() explicitly

Although `std::exit()` is called implicitly when function `main()` ends, `std::exit()` can also be called explicitly to halt the program before it would normally terminate. When `std::exit()` is called this way, you will need to include the `cstdlib` header.

Here is an example of using `std::exit()` explicitly:

```
#include <cstdlib> // for std::exit()
#include <iostream>

void cleanup()
{
    // code here to do any kind of cleanup required
    std::cout << "cleanup!\n";
}

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

    std::exit(0); // terminate and return status code 0 to operating system

    // The following statements never execute
    std::cout << 2 << '\n';

    return 0;
}
```

This program prints:

```
1
cleanup!
```

Note that the statements after the call to `std::exit()` never execute because the program has already terminated.

Although in the program above we call `std::exit()` from function `main()`, `std::exit()` can be called from any function to terminate the program at that point.

One important note about calling `std::exit()` explicitly: `std::exit()` does not clean up any local variables (either in the current function, or in functions up the call stack). Because of this, it's generally better to avoid calling `std::exit()`.

Warning

The `std::exit()` function does not clean up local variables in the current function or up the call stack.

std::atexit

Because `std::exit()` terminates the program immediately, you may want to manually do some cleanup before terminating. In this context, cleanup means things like closing database or network connections, deallocating any memory you have allocated, writing information to a log file, etc…

As an aside…

When an application exits, modern OSes will generally clean up any memory that the application does not properly clean up itself. This leads to the question, “so why bother doing cleanup on exit?”. There are (at least) two reasons:

1. Cleaning up allocated memory is a “good habit” that you will need to use to avoid memory leaks while the application is running. Cleaning up in some cases and not others is inconsistent and can lead to errors. Not cleaning up memory properly can also impact the way certain tools like memory profilers behave (they may be unable to distinguish memory that you inadvertently aren’t intentionally cleaning up from memory that you intentionally aren’t cleaning up because you don’t have to).
2. There are other kinds of cleanup that may be necessary for your program to behave predictably. For example, if you write data to a file and then unexpectedly exit, that data may not have been flushed to the file yet, and may be lost when the program exits. Closing the file before shutting down helps ensure that all cached data will be written first.

In the above example, we called function `cleanup()` to handle our cleanup tasks. However, remembering to manually call a cleanup function before calling every call to exit() adds burden to the programmer.

To assist with this, C++ offers the `std::atexit()` function, which allows you to specify a function that will automatically be called on program termination via `std::exit()`.

Related content

We discuss passing functions as arguments in lesson 20.1 -- Function Pointers.

Here’s an example:

```
#include <cstdlib> // for std::exit()
#include <iostream>

void cleanup()
{
    // code here to do any kind of cleanup required
    std::cout << "cleanup!\n";
}

int main()
{
    // register cleanup() to be called automatically when std::exit() is called
    std::atexit(cleanup); // note: we use cleanup rather than cleanup() since we're
not making a function call to cleanup() right now

    std::cout << 1 << '\n';

    std::exit(0); // terminate and return status code 0 to operating system

    // The following statements never execute
    std::cout << 2 << '\n';

    return 0;
}
```

This program has the same output as the prior example:

```
1
cleanup!
```

So why would you want to do this? It allows you to specify a cleanup function in one place (probably in `main`) and then not have to worry about remembering to call that function explicitly before calling `std::exit()`.

A few notes here about `std::atexit()` and the cleanup function: First, because `std::exit()` is called implicitly when `main()` terminates, this will invoke any functions registered by `std::atexit()` if the program exits that way. Second, the function being registered must take no parameters and have no return value. Finally, you can register multiple cleanup functions using `std::atexit()` if you want, and they will be called in reverse order of registration (the last one registered will be called first).

For advanced readers

In multi-threaded programs, calling `std::exit()` can cause your program to crash (because the thread calling `std::exit()` will cleanup static objects that may still be accessed by other threads). For this reason, C++ has introduced another pair of functions that work similarly to `std::exit()` and `std::atexit()` called `std::quick_exit()` and `std::at_quick_exit()`.

`std::quick_exit()` terminates the program normally, but does not clean up static objects, and may or may not do other types of cleanup. `std::at_quick_exit()` performs the same role as `std::atexit()` for programs terminated with `std::quick_exit()`.

std::abort and std::terminate

C++ contains two other halt-related functions.

The `std::abort()` function causes your program to terminate abnormally. **Abnormal termination** means the program had some kind of unusual runtime error and the program couldn't continue to run. For example, trying to divide by 0 will result in an abnormal termination. `std::abort()` does not do any cleanup.

```
#include <cstdlib> // for std::abort()
#include <iostream>

int main()
{
    std::cout << 1 << '\n';
    std::abort();

    // The following statements never execute
    std::cout << 2 << '\n';

    return 0;
}
```

We will see cases later in this chapter (9.6 -- Assert and static_assert) where `std::abort` is called implicitly.

The `std::terminate()` function is typically used in conjunction with `exceptions` (we'll cover exceptions in a later chapter). Although `std::terminate` can be called explicitly, it is more often called implicitly when an exception isn't handled (and in a few other exception-related cases). By default, `std::terminate()` calls `std::abort()`.

When should you use a halt?

The short answer is "almost never". Destroying local objects is an important part of C++ (particularly when we get into classes), and none of the above-mentioned functions clean up local variables. Exceptions are a better and safer mechanism for handling error cases.

Best practice

Only use a halt if there is no safe way to return normally from the main function. If you haven't disabled exceptions, prefer using exceptions for handling errors safely.

Tip

Although explicit use of halts should be minimized, there are many other ways that a program can shut down unexpectedly. For example:

- The application could crash due to a bug (in which case the OS will shut it down).
- The user might kill the application in various ways.
- The user might turn off (or lose) power to their computer.
- The sun could go supernova and consume the earth in a giant ball of fire.

A well-designed program should be able to handle being shut down at any point with minimal repercussions.

As a common example of this, modern games often autosave game state and user settings periodically, so that if the game is unexpectedly shut down without saving, the user can continue later (using the prior autosave) without much lost progress.