1.6 — Uninitialized variables and undefined behavior

Uninitialized variables

Unlike some programming languages, C/C++ does not automatically initialize most variables to a given value (such as zero). When a variable that is not initialized is given a memory address to use to store data, the default value of that variable is whatever (garbage) value happens to already be in that memory address! A variable that has not been given a known value (through initialization or assignment) is called an **uninitialized variable**.

Nomenclature

Many readers expect the terms “initialized” and “uninitialized” to be strict opposites, but they aren’t quite! In common language, “initialized” means the object was provided with an initial value at the point of definition. “Uninitialized” means the object has not been given a known value yet (through any means, including assignment). Therefore, an object that is not initialized but is then assigned a value is no longer *uninitialized* (because it has been given a known value).

To recap:

- Initialized = The object is given a known value at the point of definition.
- Assignment = The object is given a known value beyond the point of definition.
- Uninitialized = The object has not been given a known value yet.

Relatedly, consider this variable definition:

```
int x;
```

In lesson 1.4 -- Variable assignment and initialization, we noted that when no initializer is provided, the variable is default-initialized. In most cases (such as this one), default-initialization performs no actual initialization. Thus we’d say `x` is uninitialized. We’re focused on the outcome (the object has not been given a known value), not the process.

As an aside…

This lack of initialization is a performance optimization inherited from C, back when computers were slow. Imagine a case where you were going to read in 100,000 values from a file. In such case, you might create 100,000 variables, then fill them with data from the file.

If C++ initialized all of those variables with default values upon creation, this would result in 100,000 initializations (which would be slow), and for little benefit (since you're overwriting those values anyway).

For now, you should always initialize your variables because the cost of doing so is minuscule compared to the benefit. Once you are more comfortable with the language, there may be certain cases where you omit the initialization for optimization purposes. But this should always be done selectively and intentionally.

Using the values of uninitialized variables can lead to unexpected results. Consider the following short program:

```
#include <iostream>

int main()
{
    // define an integer variable named x
    int x; // this variable is uninitialized because we haven't given it a value

    // print the value of x to the screen
    std::cout << x << '\n'; // who knows what we'll get, because x is uninitialized

    return 0;
}
```

In this case, the computer will assign some unused memory to *x*. It will then send the value residing in that memory location to *std::cout*, which will print the value (interpreted as an integer). But what value will it print? The answer is "who knows!", and the answer may (or may not) change every time you run the program. When the author ran this program in Visual Studio, *std::cout* printed the value `7177728` one time, and `5277592` the next. Feel free to compile and run the program yourself (your computer won't explode).

Warning

Some compilers, such as Visual Studio, *will* initialize the contents of memory to some preset value when you're using a debug build configuration. This will not happen when using a release build configuration. Therefore, if you want to run the above program yourself, make sure you're using a *release build configuration* (see lesson 0.9 -- Configuring your compiler: Build configurations for a reminder on how to do that). For example, if you run the above program in a Visual Studio debug configuration, it will consistently print -858993460, because that's the value (interpreted as an integer) that Visual Studio initializes memory with in debug configurations.

Most modern compilers will attempt to detect if a variable is being used without being given a value. If they are able to detect this, they will generally issue a compile-time warning or error. For example, compiling the above program on Visual Studio produced the following warning:

```
c:\VCprojects\test\test.cpp(11) : warning C4700: uninitialized local variable
'x' used
```

If your compiler won't let you compile and run the above program (e.g. because it treats the issue as an error), here is a possible solution to get around this issue:

```
#include <iostream>

void doNothing(int&) // Don't worry about what & is for now, we're just using it to
trick the compiler into thinking variable x is used
{
}

int main()
{
    // define an integer variable named x
    int x; // this variable is uninitialized

    doNothing(x); // make the compiler think we're assigning a value to this variable

    // print the value of x to the screen (who knows what we'll get, because x is
uninitialized)
    std::cout << x << '\n';

    return 0;
}
```

Using uninitialized variables is one of the most common mistakes that novice programmers make, and unfortunately, it can also be one of the most challenging to debug (because the program may run fine anyway if the uninitialized variable happened to get assigned to a spot of memory that had a reasonable value in it, like 0).

This is the primary reason for the "always initialize your variables" best practice.

Undefined behavior

Using the value from an uninitialized variable is our first example of undefined behavior. **Undefined behavior** (often abbreviated **UB**) is the result of executing code whose behavior is not well-defined by the C++ language. In this case, the C++ language doesn't have any rules determining what happens if you use the value of a variable that has not been given a known value. Consequently, if you actually do this, undefined behavior will result.

Code implementing undefined behavior may exhibit *any* of the following symptoms:

- Your program produces different results every time it is run.
- Your program consistently produces the same incorrect result.
- Your program behaves inconsistently (sometimes produces the correct result, sometimes not).

- Your program seems like it's working but produces incorrect results later in the program.
- Your program crashes, either immediately or later.
- Your program works on some compilers but not others.
- Your program works until you change some other seemingly unrelated code.

Or, your code may actually produce the correct behavior anyway.

Author's note

Undefined behavior is like a box of chocolates. You never know what you're going to get!

C++ contains many cases that can result in undefined behavior if you're not careful. We'll point these out in future lessons whenever we encounter them. Take note of where these cases are and make sure you avoid them.

Rule

Take care to avoid all situations that result in undefined behavior, such as using uninitialized variables.

Author's note

One of the most common types of comment we get from readers says, "You said I couldn't do X, but I did it anyway and my program works! Why?".

There are two common answers. The most common answer is that your program is actually exhibiting undefined behavior, but that undefined behavior just happens to be producing the result you wanted anyway… for now. Tomorrow (or on another compiler or machine) it might not.

Alternatively, sometimes compiler authors take liberties with the language requirements when those requirements may be more restrictive than needed. For example, the standard may say, "you must do X before Y", but a compiler author may feel that's unnecessary, and make Y work even if you don't do X first. This shouldn't affect the operation of correctly written programs, but may cause incorrectly written programs to work anyway. So an alternate answer to the above question is that your compiler may simply be not following the standard! It happens. You can avoid much of this by making sure you've turned compiler extensions off, as described in lesson 0.10 -- Configuring your compiler: Compiler extensions.

Implementation-defined behavior and unspecified behavior

Implementation-defined behavior means the behavior of some syntax is left up to the implementation (the compiler) to define. Such behaviors must be consistent and documented, but different compilers may produce different results.

Let's look at a simple example of implementation-defined behavior:

```
#include <iostream>

int main()
{
	std::cout << sizeof(int) << '\n'; // print how many bytes of memory an int value takes

	return 0;
}
```

On most platforms, this will produce `4`, but on others it may produce `2`.

Related content

We discuss `sizeof()` in lesson 4.3 -- Object sizes and the sizeof operator.

Unspecified behavior is almost identical to implementation-defined behavior in that the behavior is left up to the implementation, but the implementation is not required to document the behavior.

We generally want to avoid implementation-defined and unspecified behavior, as it means our program may not work as expected if compiled on a different compiler (or even on the same compiler if we change project settings that affect how the implementation behaves!)

Best practice

Avoid implementation-defined and unspecified behavior whenever possible, as they may cause your program to malfunction on other implementations.

Related content

We show examples of unspecified behavior in lesson 6.1 -- Operator precedence and associativity.

Quiz time

Question #1

What is an uninitialized variable? Why should you avoid using them?

Show Solution

Question #2

What is undefined behavior, and what can happen if you do something that exhibits undefined behavior?

Show Solution