2.1 — Fundamental variable definition, initialization, and assignment

BY ALEX ON JUNE 4TH, 2007 | LAST MODIFIED BY ALEX ON JUNE 18TH, 2018

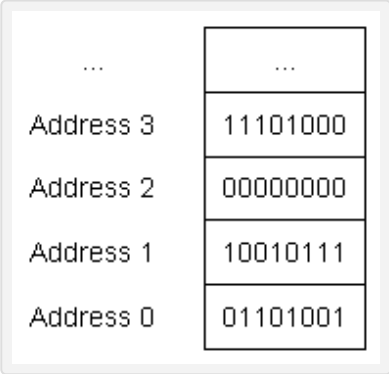
Addressing memory

This lesson builds directly on the material in the section "**1.3 -- A first look at variables**".

In the previous lesson on variables, we talked about the fact that variables are names for a piece of memory that can be used to store information. To recap briefly, computers have random access memory (RAM) that is available for programs to use. When a variable is defined, a piece of that memory is set aside for that variable.

The smallest unit of memory is a **binary digit (bit)**, which can hold a value of 0 or 1. You can think of a bit as being like a traditional light switch -- either the light is off (0), or it is on (1). There is no in-between. If you were to look at a random segment of memory, all you would see is …011010100101010… or some combination thereof. Memory is organized into sequential units called **memory addresses** (or addresses for short). Similar to how a street address can be used to find a given house on a street, the memory address allows us to find and access the contents of memory at a particular location. Perhaps surprisingly, in modern computers, each bit does not get its own address. The smallest addressable unit of memory is known as a **byte**. The modern standard is that a byte is comprised of 8 sequential bits. Note that some older or non-standard machines may have bytes of a different size -- however, we generally need not worry about these. For these tutorials, we'll assume a byte is 8 bits.

The following picture shows some sequential memory addresses, along with the corresponding byte of data:



Because all data on a computer is just a sequence of bits, we use a **data type** (often called a "type" for short) to tell us how to interpret the contents of memory in some meaningful way. You have already seen one example of a data type: the integer. When we declare a variable as an integer, we are telling the compiler "the piece of memory that this variable addresses is going to be interpreted as a non-fractional number".

When you assign a value to a data type, the compiler and CPU take care of the details of encoding your value into the appropriate sequence of bits for that data type. When you ask for your value back, your number is "reconstituted" from the sequence of bits in memory.

There are many other data types in C++ besides the integer, most of which we will cover shortly. As shorthand, we typically refer to a variable's "data type" as its "type".

Fundamental data types

C++ comes with built-in support for certain data types. These are called **fundamental data types** (in the C++ specification), but are often informally called **basic types**, **primitive types**, or **built-in types**.

Here is a list of the fundamental data types, some of which you have already seen:

Category	Types	Meaning	Example	Notes
boolean	bool	true or false	true	
character	char, wchar_t, char16_t, char32_t	a single ASCII character	'c'	char16_t, char32_t introduced in C++11
floating point	float, double, long double	a number with a decimal	3.14159	
integer	short, int, long, long long	a whole number	64	long long introduced in C99/C++11

void	no type	void	n/a	

This chapter is dedicated to exploring these basic data types in detail.

Defining a variable

In the “basic C++” section, you already learned how to define an integer variable:

```
int nVarName; // int is the type, nVarName is the name of the variable
```

To define variables of other data types, the idea is exactly the same:

```
type varName; // type is the type (eg. int), varName is the name of the variable
```

In the following example, we define 5 different variables of 5 different types.

```
bool bValue;
char chValue;
int nValue;
float fValue;
double dValue;
```

Note that void has special rules about how it can be used, so the following won’t work:

```
void vValue; // won't work, void can't be used as a type for variable definitions
```

Variable initialization

When a variable is defined, you can immediately give that variable a value. This is called **variable initialization** (or initialization for short).

C++ supports three basic ways to initialize a variable. First, we can do **copy initialization** by using an equals sign:

```
int nValue = 5; // copy initialization
```

(Note for advanced users: The equals sign used here for copy initialization is part of the initialization syntax, and is not considered a use of the assignment operator that gets invoked when doing copy assignment)

Second, we can do a **direct initialization** by using parenthesis.

```
int nValue(5); // direct initialization
```

Even though direct initialization form looks a lot like a function call, the compiler keeps track of which names are variables and which are functions so that they can be resolved properly.

Direct initialization can perform better than copy initialization for some data types, and comes with some other benefits once we start talking about classes. It also helps differentiate initialization from assignment. Consequently, we recommend using direct initialization over copy initialization.

Rule: Favor direct initialization over copy initialization

Uniform initialization in C++11

Because C++ grew organically, the copy initialization and direct initialization forms only work for some types of variables (for example, you can’t use either of these forms to initialize a list of values).

In an attempt to provide a single initialization mechanism that will work with all data types, C++11 adds a new form of initialization called **uniform initialization** (also called brace initialization):

```
int value{5};
```

Initializing a variable with an empty brace indicates default initialization. Default initialization initializes the variable to zero (or empty, if that’s more appropriate for a given type).

```
int value{}; // default initialization to 0
```

Uniform initialization has the added benefit of disallowing “narrowing” type conversions. This means that if you try to use uniform initialization to initialize a variable with a value it can not safely hold, the compiler will throw an warning or error. For example:

```
int value{4.5}; // error: an integer variable can not hold a non-integer value
```

Rule: If you’re using a C++11 compatible compiler, favor uniform initialization

Variable assignment

When a variable is given a value after it has been defined, it is called a **copy assignment** (or assignment for short).

```
int nValue;
nValue = 5; // copy assignment
```

C++ does not provide any built-in way to do a direct or uniform assignment.

Uninitialized variables

A variable that is not initialized is called an **uninitialized variable**. In C++, a fundamental variable that is uninitialized will have a garbage value until you assign a valid one. We discuss this in previous lesson **A first look at variables, initialization, and assignment**.

Side note: C++ also has other non-fundamental types, such as pointers, structs, and classes. Some of these do not initialize by default, and some of them do. We’ll explore these types in future lessons. For now, it’s safer to assume all types do not initialize by default.

Rule: Always initialize your fundamental variables, or assign a value to them as soon as possible after defining them.

Defining multiple variables

It is possible to define multiple variables *of the same type* in a single statement by separating the names with a comma. The following 2 snippets of code are effectively the same:

```
int a, b;
```

```
int a;
int b;
```

You can also initialize multiple variables defined on the same line:

```
int a = 5, b = 6;
int c(7), d(8);
int e{9}, f{10};
```

There are three mistakes that new programmers tend to make when defining multiple variables in the same statement.

The first mistake is giving each variable a type when defining variables in sequence. This is not a bad mistake because the compiler will complain and ask you to fix it.

```
int a, int b; // wrong (compiler error)

int a, b; // correct
```

The second error is to try to define variables of different types on the same line, which is not allowed. Variables of different types must be defined in separate statements. This is also not a bad mistake because the compiler will complain and ask you to fix it.

```
int a, double b; // wrong (compiler error)

int a; double b; // correct (but not recommended)

// correct and recommended (easier to read)
int a;
double b;
```

The last mistake is the dangerous case. In this case, the programmer mistakenly tries to initialize both variables by using one initialization statement:

```
int a, b = 5; // wrong (a is uninitialized!)
```

```
int a= 5, b= 5; // correct
```

In the top statement, variable “a” will be left uninitialized, and the compiler may or may not complain. If it doesn’t, this is a great way to have your program intermittently crash and produce sporadic results.

The best way to remember that this is wrong is to consider the case of direct initialization or uniform initialization:

```
int a, b(5);
int c, d{5};
```

This makes it seem a little more clear that the value 5 is only being assigned to variable b.

Because defining multiple variables on a single line AND initializing them is a recipe for mistakes, we recommend that you only define multiple variables on a line if you’re not initializing any of them.

Rule: Avoid defining multiple variables on a single line if initializing any of them.

Where to define variables

Older C compilers forced users to define all of the variables in a function at the top of the function:

```
int main()
{
    // all variable up top
    int x;
    int y;

    // then code
    std::cout << "Enter a number: ";
    std::cin >> x;

    std::cout << "Enter another number: ";
    std::cin >> y;

    std::cout << "The sum is: " << x + y << std::endl;
    return 0;
}
```

This style is now obsolete. C++ compilers do not require all variables to be defined at the top of a function. The proper C++ style is to define variables as close to the first use of that variable as you reasonably can:

```
int main()
{
    std::cout << "Enter a number: ";
    int x; // we need x on the next line, so we'll declare it here, as close to its first use as we rea
sonably can.
    std::cin >> x; // first use of x

    std::cout << "Enter another number: ";
    int y; // we don't need y until now, so it gets declared here
    std::cin >> y; // first use of y

    std::cout << "The sum is: " << x + y << std::endl;
    return 0;
}
```

This has quite a few advantages.

First, variables that are defined only when needed are given context by the statements around them. If x were defined at the top of the function, we would have no idea what it was used for until we scanned the function and found where it was used. Defining x amongst a bunch of input/output statements helps make it obvious that this variable is being used for input and/or output.

Second, defining a variable only where it is needed tells us that this variable does not affect anything above it, making our program easier to understand and requiring less scrolling.

Finally, it reduces the likelihood of inadvertently leaving a variable uninitialized, because we can define and then immediately initialize it with the value we want it to have.

Most of the time, you'll be able to declare a variable on the line immediately preceding the first use of that variable. However, you will occasionally encounter a case where this is either not desirable (due to performance reasons), or not possible (because the variable will get destroyed and you need it later). We'll see examples of these cases in future chapters.

Rule: Define variables as close to their first use as you reasonably can.


2.2 -- Void


Index


1.12 -- Chapter 1 comprehensive quiz

177 comments to 2.1 — Fundamental variable definition, initialization, and assignment

« Older Comments 1 2 3

yugin
July 8, 2018 at 9:37 pm · Reply

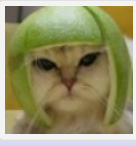
Is it merely a coincidence that "int" is used for both initialising integers and functions, or is there some deeper meaning or reason? Also, is it correct to say that when the compiler sees

```
1 int something(<value>)
```

, it knows we're talking about a variable, while when it sees

```
1 int something(<type> <value>)
```

, it instead recognises it as a function?



Alex
July 9, 2018 at 4:44 pm · Reply

> Is it merely a coincidence that "int" is used for both initialising integers and functions, or is there some deeper meaning or reason?

int has nothing to do with initialization. int defines a variable as an integer type, or in the case of a function, that the function either returns or accepts an integer type.

> Also, is it correct to say that when the compiler sees

```
1 int something(<value>)
```

, it knows we're talking about a variable, while when it sees