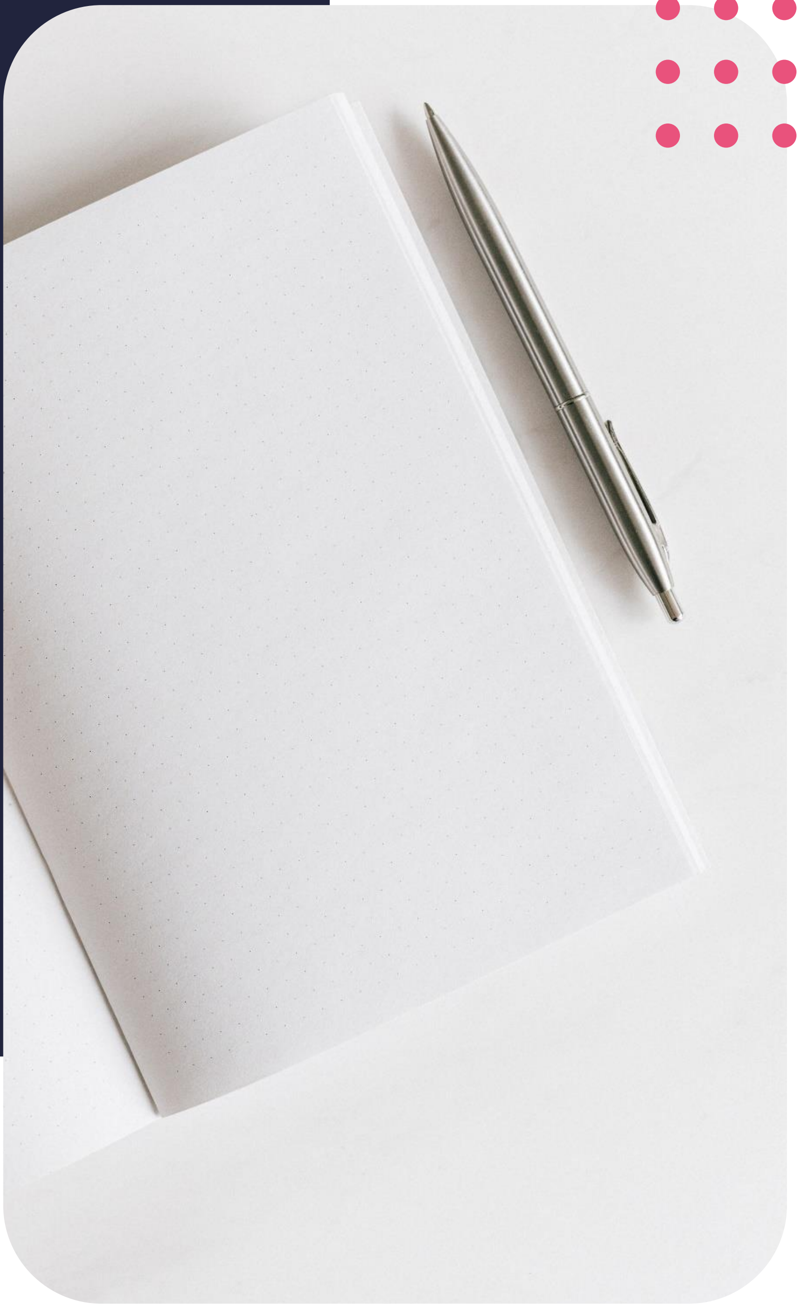
Diploma in Computer Science

Data types

Contents



Lesson outcomes

By the end of this lesson, you should be able to:

Describe the data types available in C

Understand the reasoning involved in choosing a data type

Appreciate the various data structures available in C

Understand how type qualifiers modify variables

In this lesson, we will look at the main tools in the C language, data types. These allow you to manipulate data. As we discussed in module 1, the whole point of having a computer is to be able to manipulate data. A computer wouldn't be of any use if it couldn't manipulate data, right? C provides a wealth of tools that allow you to manipulate data in many different ways. These come in many different configurations, in this lesson, we will look at the syntax used in these and how you use them according to C syntax. Let's jump in!

Coming up it our next lesson, we will learn all about variables and learn how to take advantage of the data types that we are learning about today. We will also look at a number of examples in which we will explore the power of variables and constants. With power comes responsibility; we will also look at the things that should be avoided when working with variables to make sure your code doesn't go haywire.

Data, data, and more data

There are several data types in C, and all of these are different from each other. We can even go as far as to say they are unique. As a result, it's important to make sure that you understand the existing data types, their abilities, and their limitations. Remember when we started talking about C programming, we mentioned that it is a unique language due to its close association with hardware, while at the same time offering a lot of the functionality of high-level languages. This is especially evident with data types. Data types depend entirely on the hardware that you're running your code on. An int on your laptop will be smaller than an int on a supercomputer, so knowing the limitations of the hardware you're working on is important. This is also why the data types are defined as being minimums- an int value, as you will learn, is at minimum -32767 to 32767: on certain machines, it will be able to store even more values that this.

There are four classes of data types, namely

- basic types,
- enumerated types,
- void and
- derived types.

Basic types

Basic data types are

arithmetic in nature. We can generally break basic data types into 2 categories,

- integer data types and
- floating-point data types.

Integer types

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes or (4bytes for 32 bit OS32-bit3372036854775808 to 9223372036854775807	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

This table summarises what you can do with integer data types. You will notice that some data types like the long data type have alternate specifications. Remember what we established in earlier lessons that C is closely associated with hardware? This is exactly what is going on here. C will allow you to create these data types with constraints that correspond to the hardware that you are running it on. You can easily get the maximum size of data types using the sizeof operator that you want to use.

This code will get the maximum sizes of data types on your particular pc

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <float.h>

int main(int argc, char** argv) {

    printf("CHAR_BIT    :   %d\n", CHAR_BIT);
    printf("CHAR_MAX    :   %d\n", CHAR_MAX);
    printf("CHAR_MIN    :   %d\n", CHAR_MIN);
    printf("INT_MAX     :   %d\n", INT_MAX);
    printf("INT_MIN     :   %d\n", INT_MIN);
    printf("LONG_MAX    :   %ld\n", (long) LONG_MAX);
    printf("LONG_MIN    :   %ld\n", (long) LONG_MIN);
    printf("SCHAR_MAX   :   %d\n", SCHAR_MAX);
    printf("SCHAR_MIN   :   %d\n", SCHAR_MIN);
    printf("SHRT_MAX    :   %d\n", SHRT_MAX);
    printf("SHRT_MIN    :   %d\n", SHRT_MIN);
    printf("UCHAR_MAX   :   %d\n", UCHAR_MAX);
    printf("UINT_MAX    :   %u\n", (unsigned int) UINT_MAX);
    printf("ULONG_MAX   :   %lu\n", (unsigned long) ULONG_MAX);
    printf("USHRT_MAX   :   %d\n", (unsigned short) USHRT_MAX);

    return 0;
}
```

You will see that all of these values will match what we have in this table. Results will have the alternate value for some systems, depending on your hardware.

In C, char values are stored in 1 byte, and are, in actual fact, numbers as

encoded using the ASCII encoding. Remember the ASCII code that we looked at in module 1? This is the same ASCII code that we use here. The ASCII code uses 8 bits, just like it shows in the table that we looked at earlier in this lesson. It is not a good practice to use the ASCII numeric value directly, but you should know that the characters

- 'A' to 'Z' are contiguous,
- 'a' to 'z' are contiguous, and
- '0' to '9' are contiguous.

This means that you can write code like ‘d-2’ and get a ‘b’! by the way, char, like we saw in previous lessons, holds characters- things like letters, punctuation, and spaces. The actual translation is described by the ASCII standard, you might want to check your module 1 notes for the ASCII table.

Like all other data types in C, the actual size depends on the hardware you’re working on. The smallest it is at least 8 bits, so you will have 0 to 127 to work with. Remember we also mentioned in module 1 that 0 in computing counts! Alternatively, you can use signed char to get at least -128 to 127.

Unsigned char effectively accommodates the use of Extended ASCII characters which represent most special characters such as the registered trademark sign ®, copyright sign ©, some European letters like è, é, and so on. Both char and unsigned char are stored internally as integers so they can effectively be operated on using arithmetic operators, like what we mentioned just now.

The int data type

represents all real numbers that are not fractions. But if you look at the maximum size of the int data type, it is pretty small. If it’s very likely that the variable will exceed the maximum possible value, it would be better to declare it as long int, so that it can accommodate more. When using int, there are no fractions, so

the value is absolute, but then the problem comes when you divide the value stored as int.

The fractional part will be discarded, and for that you would need the float data type that we are going to discuss in a bit. If, for example, you want to calculate pi and calculate 22/7 using the int data type, the result would be 3 and everything else would be discarded. This will result in a value that is wildly inaccurate.

Normally, the short integer type isn’t used because it is so small, but it’s just good to know that it exists. Like int, it can store -32768 to 32767. This is, however, where the party ends. Luckily, whenever you can use short int, you can well and equally substitute it with int.

But what if you want something bigger? Well, C won’t disappoint you. The long data type stores integers like int but gives a wider range of values and of course sits on more memory, so you might want to be careful with it in some applications.

Long stores at least 32 bits, giving it a range of -2,147,483,648 to 2,147,483,647. You can alternatively also use unsigned long for a range of 0 to 4,294,967,295.

You can go even bigger with the long long integer type, and yes, it is as big as it sounds! This is typically an overkill in probably any application, but well, things move do fast in computing, and it could be commonplace sooner than you think! And well, it’s there, and you can use it and get at least −9,223,372,036,854,775,807 to 9,223,372,036,854,775,807. You could even max it out with unsigned long long , which will give you at least 0 to 18,446,744,073,709,551,615! This will do a number to the system footprint of your program and it would be best to just avoid it. You probably wont need all that storage anyway

Boolean types

Advanced types

Advanced data types build on the abilities of basic data types. This gives you even more flexibility and allows you to do much more.

Arrays

An array is a collection of elements of the same data type. An array it is drawn from the dictionary meaning of the word, which is an impressive display or range of a particular type of thing, or an ordered series or arrangement. this is a pretty handy way in which you can remember how an array works. You could also think of an array as a pile of books that have different titles, but they're all books.

Instead of declaring individual variables, such as val0, val1, ..., and val10, we can just declare one array, for example vals and use vals[0], vals[1], and ..., vals[10] to represent individual variables. We would use an index to access any of the elements in the array. Arrays are created in a contiguous memory block. The lowest address corresponds to the first element and the highest address to the last element. To declare an array in C, we specify the type of the elements and the number of elements required using the syntax: (note to VO, mention all elements or pause when you display the following line of text)

```
datatype arrayName [ arraySize ];
```

The arraySize must be an integer constant greater than zero and type can be any valid C data type. For example, to declare a 15element array named initials of type char, we would write:

```
Char initials[15];
```

This array would be able to hold 15 characters. The number of elements that you will assign to the array cannot be larger than the size of the array that you declared. Remember when we said a computer starts counting from zero? This is particularly important in arrays, where the first item in the array is a zero. If you're to type `[initials[2]=E;`

the letter E will be assigned to the 3rd position in the array; remember we start counting from zero! An element is accessed specifying the array name and the position of the item in square brackets. This is called the item's index.

Pointers

A pointer is a variable whose value is the address of another variable, an actual direct address of the memory location. Think of it as how a road sign shows you where to find a place. The road sign is literally a pointer to the place. A pointer in C works in pretty much the same way, it's a road sign to a certain memory location. A pointer is declared using the syntax

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication, the only difference being that here the asterisk is being used to designate a variable as a pointer. For now, we aren't really worrying about pointers, we're just acquainting ourselves with the tools available in C before we start using them.

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

It is good programming good practice to always assign a NULL value to a pointer variable if you do not yet have the address to which it will point This is done at the time of variable declaration. At this point, the pointer is called a null pointer. You will see this term quite a lot when we start working with pointers.

A NULL pointer is a constant with a value of zero defined in several standard libraries.

In most operating systems, programs are not allowed to access memory at address 0 because that memory is critical system memory, reserved for the operating system. This memory address has a special job; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to be like a blank road sign, not pointing to anything.

Apart from being used in their basic form, pointers can be used in 5 other forms

Pointer arithmetic

There are four arithmetic operators that can be used in pointers: ++, --, +, -

- Pointer arrays

You can define arrays to hold a number of pointers.

- Pointers to pointers

C allows you to have pointer on a pointer and so on.

- Passing pointers to functions

Passing an argument by reference or by address enables the argument that has been passed to be changed in the function in which another function is being called. We will look at this in greater detail in another lesson

- Return pointer from functions

C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory. Pointers are a really powerful tool, but can easily create messy code if not used correctly

Unions

A union allows you to store(s) different data types in the same memory location A union is defined with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for different purposes.

The union statement defines a new data type with more than one member. Unions use the syntax

```
union [union tag] {
   member definition;
   member definition;
   ...
   member definition;
} [union variables];
```

The word union here is optional, and each member definition is the usual variable definition, such as int a; or char initial; or any other valid variable definition. At the end of the union's definition, before the final semicolon, we can specify one or more union variables.

This data type is particularly great if you are running on low memory constraints or you just want to write a memory efficient program. Think of this as a single seater couch. either you, your friend or your dog can sit on the couch, but all three of you cannot sit on the couch at the same time.

The memory occupied by a union will be large enough to hold the largest member of the union.

To access any member of a union, we use the member access operator (.). The member access operator is denoted by a full stop between the union variable name and the union member that we want to access Remember the keyword union? We use the keyword union to define variables of union type.

Structures

A struct is a composite data type declaration that defines a group of variables under one name in a block of memory, allowing these variables to be accessed via a single pointer or by the struct declared name, which returns the same address. The struct data type can be used for mixed-data-type records such as a file directory entry, things like file length, name, extension, physical address and so on.

The C struct

directly references a contiguous block of physical memory, usually limited in size by word-length boundaries. It corresponds to the similarly named feature available in some assemblers for Intel processors. Being a block of contiguous memory, each field within a struct is located at a certain fixed offset from the start.

Because the contents of a struct are stored in contiguous memory, we need to use the sizeof operator to get the number of bytes needed to store a particular type of struct, just as it can be used for basic data types. The alignment of particular fields in the struct is the same as the word size of the machine. This means that the implementation is hardware specific, as we mentioned before, C is very close to hardware, and the word size we are referring to here is the same word size that we looked at in module 1.

There are three ways to initialize a structure. For the struct type

```
// Declare the struct with integer members x, y *
struct point {
int     x;
int     y;
};
```

- Define a variable p of type point, and initialize its first two members in place

```
struct point p = { 1, 2 };
```

- Define a variable p of type point, and set members using designated initializers

```
struct point p = { .y = 2, .x = 1 };
```

If an initializer is given or if the object is statically allocated, omitted elements are initialized to 0.

A third way of initializing a structure is to

- copy the value of an existing object of the same type

Define a variable q of type point, and set members to the same values as those of p

```
struct point q = p;
```

A struct can also be assigned to another struct. A compiler might use memcpy() to perform such an assignment.

Pointers can be used to refer to a struct by its address. This is useful for passing structs to a function. The pointer can be dereferenced using the * operator. The -> operator dereferences the pointer to struct (left operand) and then accesses the value of a member of the struct (right operand).

Type qualifiers

Type qualifiers are used to modify the properties of a variable. a type qualifier is a keyword that is applied to a type, resulting in a qualified type. In simpler English, a type qualifier will tell the compiler how a particular data type is going to be used. Let's take a closer look.

Constant

Constants are also like normal variables. The only difference is their values can't be modified by the program once they are defined. Essentially, this is a fixed value stored in a specific portion of memory They are also called as literals. They may be belonging to any of the data type.

Volatile

When a variable is defined as volatile, the program may not change the value of the variable explicitly, but the variable value might keep on changing without any explicit assignment by the program. These types of qualifiers are called volatile. For example, if a global variable's address is passed to clock routine of the operating system to store the system time, the value in this address will keep on changing without any assignment by the program.

Restrict

The restrict keyword is mainly used in pointer declarations as a type qualifier for pointers.

It doesn't add any new functionality. It is only a way for programmer to instruct the compiler to perform optimisations. When we use restrict with a pointer ptr, it tells the compiler that ptr is the only way to access the object that it points to. The compiler doesn't need to add any additional checks.

You can't however, use the restrict and volatile qualifiers at the same time. Expect some really wild and unexpected behaviour if you do so!

Choosing the right data type

C data types allow you to be very specific with the type of data your variables will hold and how they will be manipulated. This gives you a lot of power over your code, but it's important to pick the right one.

In general, you should pick the minimum for your task. If you know you'll be counting from integer 1 to 10, you don't need a long and you definitely don't need a double. If you know that you will never have negative values, you'd look into using the unsigned variants of the data types.

By providing this functionality rather than doing it automatically, C is able to produce very light and efficient code. Congratulations, we have just discovered the secret! It is, however, up to you as the programmer to understand the abilities and limitations and choose appropriate data types.

As we saw earlier in the lesson, we can use the sizeof() operator to check the size of a variable. Like we said before, and will say again for the sake of emphasis, some data types are dependent on hardware and it's important to choose these carefully. The sizeof operator allows you to see if the data type that you have chosen is up to task for your requirements.

Conclusion

In this lesson, we looked at the various data types available in C. data is the main reason why we use computers in the first place, and it is important that the computer allows us to manipulate it in the most flexible way possible. The data types that we discussed today allow us to manipulate data in ways that are literally limited by our imagination. In our next lesson, we will look at variables in greater detail and see how we can use them in conjunction with the data types that we learnt about today. That's it from me today, make sure you practise your coding, and I will see you next time!

References

Embeddedgurus.com. (2016). Efficient C Tips #1 – Choosing the correct integer size « Stack Overflow. [online] Available at: https://embeddedgurus.com/stack-overflow/2008/06/efficient-c-tips-1-choosing-the-correct-integer-size/

freeCodeCamp.org (2020). Data Types in C - Integer, Floating Point, and Void Explained. [online] freeCodeCamp.org. Available at: https://www.freecodecamp.org/news/data-types-in-c-integer-floating-point-and-void-explained/

Lmu.edu. (2017). Ray Toal. [online] Available at: https://cs.lmu.edu/~ray/

Uic.edu. (2020). C Programming Course Notes - Character Strings. [online] Available at: https://www.cs.uic.edu/~jbell/CourseNotes/C_Programming/CharacterStrings.html

Uic.edu. (2020). C Programming Course Notes - Character Strings. [online] Available at: https://www.cs.uic.edu/~jbell/CourseNotes/C_Programming/CharacterStrings.html

COMPUTER SCIENCE. (2017). PROGRAMMING LANGUAGE THEORY. [online] Available at: https://computingstudy.wordpress.com/programming-language-theory/

Datavail. (2017). On the Importance of Choosing the Correct Data Type. [online] Available at: https://www.datavail.com/blog/on-the-importance-of-choosing-the-correct-data-type/#:~:

Lmu.edu. (2015). computerscience. [online] Available at: https://cs.lmu.edu/~ray/notes/computerscience/

Lmu.edu. (2020). c. [online] Available at: https://cs.lmu.edu/~ray/notes/c/

Lmu.edu. (2020). CMSI 585: Programming Language Semantics. [online] Available at: https://cs.lmu.edu/~ray/classes/pls/

Ntu.edu.sg. (2011). C Basics - C Programming Tutorial. [online] Available at: https://www3.ntu.edu.sg/home/ehchua/programming/cpp/c1_Basics.html

Utah.edu. (2020). Programming - Topics. [online] Available at: https://www.cs.utah.edu/~germain/PPS/Topics/index.html