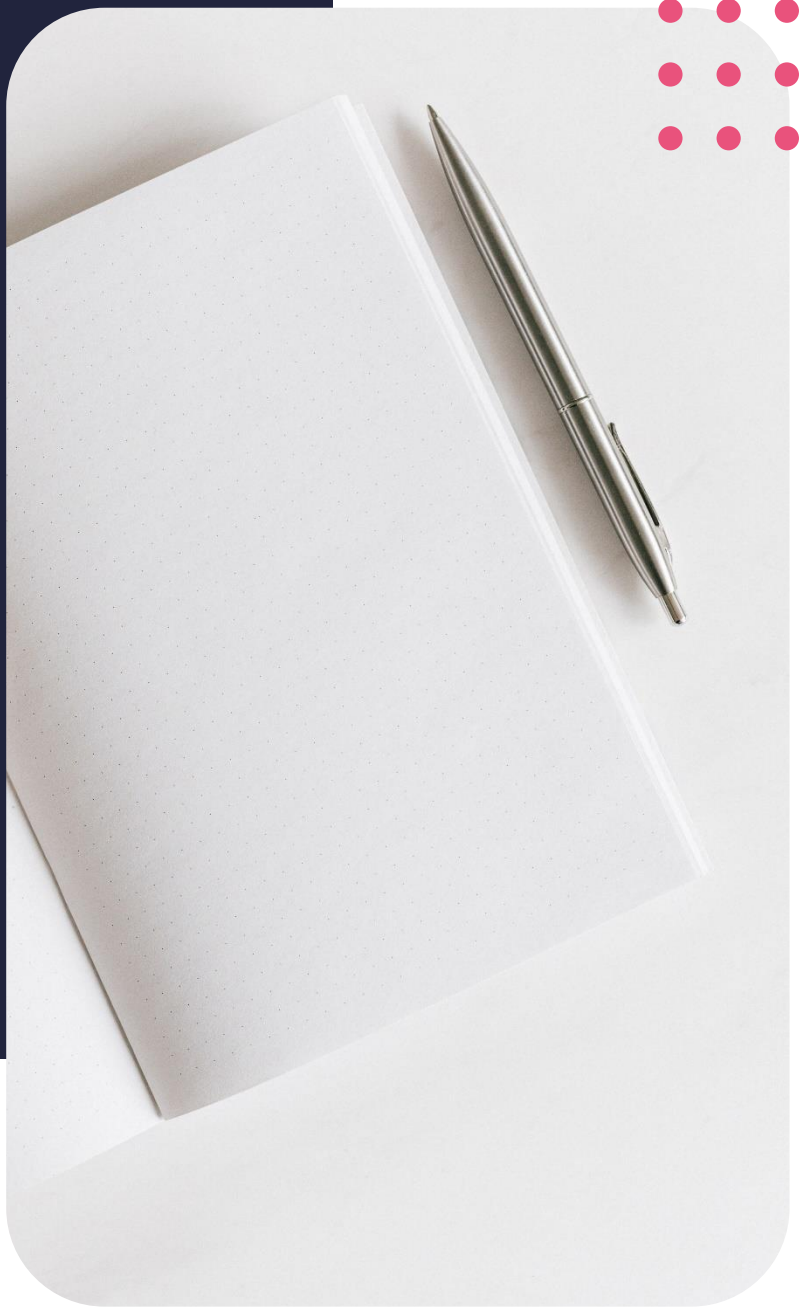
Diploma in Computer Science

Typecasting & Advanced Data Structures

Contents



Lesson outcomes

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

Explore structures and unions

Define typecasting and type conversion

Recall data types and their uses

Structures

You recall from our lesson on data types that structures are a data type that is built into C. A structure is a composite data type or record declaration that defines a physically grouped list of variables under one name. It allows this grouped data to be accessed using just one pointer. There are a number of ways in which structures are useful.

What's inside a structure?

Let's start by taking a look at what a structure consists of.

One interesting thing about a structure is that the variables are physically grouped in the same block of memory. This sounds pretty familiar and very similar to arrays. Both actually work the same way, except that in an array, you can only store one data entity whereas in a structure, you can store a number of different data types to form a record.

To jog your memory a bit we'll look at the syntax of a struct now.

The computer allocates enough space according to the data type that you specify. You can access any of the individual components using the dot operator [.]. Variables of this type can be passed to functions returned from functions and used in comparison operations just like you do with any ordinary variable.

Let's take a look at this example:

```
struct string {
 int length;
 char *data;
};
```

Here, a struct of type string is declared. You could equally make it an int, float or char. In this snippet, the computer allocates enough space to hold an int and a char. As mentioned earlier, you can access the individual components using the dot operator.

Here's a more complete version of the code snippet:

```
struct string {
int length;
 char *data;
 };
int main(int argc, char **argv) {
 struct string s;
s.length = 4;
s.data = "This is a long string!";
```

```
puts(s.data);
return 0;
```

In this code snippet we have declared a variable S that is going to allow us to access data integers within the struct. Using the dot notation, we could access, for example, the length entity using s.length.

Unions

C provides quite a lot of ways for you to structure your data to preserve memory. One other method of doing this is called a union. A union behaves in much the same way as a structure with a subtle difference as we are going to see shortly.

Unions are a composite data type that is closely related to structs. In a union, members of different data types are declared, but they all occupy the same region of storage. This technically means they share the storage and cannot all exist at the same time. As such, only one member is valid at a given time. This is the key difference between a struct and a union: all the members on a struct can coexist, while those of a union cannot. The size of a union is the size of the largest member of the union.

The basic syntax of a union looks like this:

```
Union [union tag] {
   Member definition;
   Member definition;
   ...
   Member definition;
} [one or more union variables];
```

The variables at the end of the union declaration are optional, just like they are optional in a struct. This looks eerily like a struct, in fact, we just literally plucked out the keyword struct and replaced it with the keyword union! The natural question that would follow in this instance would be “So why on earth do we need something that looks almost exactly like another data structure?”. There are quite a number of reasons why you would want to use a union. The biggest reason being unions save space.

Imagine a scenario where a data component in your code can be one of a number of data types at a time. Instead of creating separate variables for each of the instances or creating a structure with all the possible data types for the data item, you could simply create a union that only occupies at most the size of the size of the largest member. This saves quite a bit of memory, and this quickly adds up if you are working with particularly large sets of data.

Typecasting

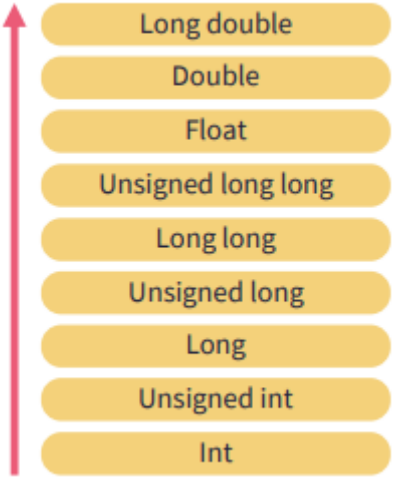
Still on our path of exploring clever ways of utilising memory and handling variables, we are going to look at another concept called typecasting. Typecasting is a way of changing an object from one data type to another. This is a mechanism in C and many other programming languages that makes sure that operations are always carried out correctly. To understand what typecasting is really about, we’ll look at variables and what assigning a data type to a variable means.

A few steps back...

We need to take a few steps back so we can better understand what we're trying to get at. When we talked about variables and constants, we mentioned that each of them has a data type associated with it. Using this data type, the computer can tell what kind of data is stored in the variable or constant. When variables and constants of different types put together in an expression, they are converted to the same data type.

Type conversion

The process of converting one predefined type into another is called type conversion. An immediate result of doing this is the prevention of unwanted consumption of memory. This is because the variable type used to store the value of the operation consumes only the amount of memory its needs and was actually introduced to improve memory management efficiency. Values should always be typecast to a data type that makes sense. It wouldn't be reasonable to convert, say, a character to a float, but it makes sense to convert a float to an integer.



Here is a diagram showing the data type hierarchy. This hierarchy is what the compiler follows when typecasting. A data type can either be promoted or demoted. Promotion occurs when a data type that is lower in the hierarchy is converted to a data type that is higher up in the hierarchy. This is almost never a problem , since smaller data types fit in larger data types. When a data type that is in a higher position in the hierarchy is converted to a data type that is lower in the hierarchy this is called demotion. This can pose problems of data loss, as larger data types need more memory to store data than the smaller types.

In some instances, typecasting is an automatic process, or it can be done manually. Two methods of type conversion are supported in C. They are explicit type conversion and implicit type conversion. Explicit type conversion is done by the programmer by posing the data type of the expression in a specific type. In this process, data in a value of a higher type is converted into a lower type of value. Let's take a closer look at explicit type conversion.

Explicit type conversion

Explicit type conversion is when the programmer poses the data type of the expression n a specific type so that data of a higher value type is converted into a lower type value.

Let's take a look at third example:

```
#include<stdio.h>
void main()
{
float num=3.141592654;
int integer;
integer = (int) num;
printf("Explicit value is %d",integer);
return 0;
}
```

Here, we declared the variable num as a float so that it can hold the value of π. We then declared an additional variable that we named integer, with the data type int. The next line is what we are most interested in. We cast the data type int onto num. Since num carries the data type float, the number that is stored there has a fractional part. When we cast the data type int onto it, the fractional part of the number is discarded, since int cannot hold any value that is not a whole number. When we print the value of num, despite having declared it as a float and initialised it with the value of π, it will return 3 , which is an integer, and the fractional part has been discarded. We could turn this into a pretty useful programme that rounds off numbers to the nearest whole number.

Look at this example:

```
#include<stdio.h>
void main()
{
float num=3.141592654;
int integer;
integer = (int) (num+0.5);
printf("Explicit value is %d",integer);
return 0;
}
```

Let's take a look at this code snippet which we have modified a bit so that it can round off decimal numbers. The function will now round off the provided number more accurately. You may be

Wondering what just happened there. Well, if you remember from mathematics, if the 1st decimal place of a number is below 0.5, then we round the number downwards. If the 1st decimal place of a

Number is above 0.5, then we round the number upwards. Now in our first code snippet, the computer wouldn't be able to differentiate between 3.141592654 and 3.641592654. Adding 0.5 will cause our floating-point number to jump into the next interval, so when we eventually discard the decimal part of the number when we cast it to int, we get an accurate value. Genius! Now notice that we had to put the addition operation within brackets. If you remember from our lesson on operator precedence,

Anything in parentheses is always operated on first. This means that 0.5 is added then the type int is cast on to our variable. If we left out the parentheses, it would mean that the cast will be done first, then the value 0.5 would be added. Now this is a problem because our variable is now of the type int, so adding a decimal value would not have any effect as it can no longer be stored in the variable and this would make our results inaccurate.

Implicit type conversion

In implicit type conversion, data is automatically converted into another type without the programmer’s involvement. The compiler is the one that carries out this task, but there's a catch. So, type conversion only occurs if both of the data types are compatible with each other.

Key differences...

The key differences between typecasting and type conversion is that typecasting refers to the conversion of a variable from one data type to another by the user. Type conversion is an automatic process carried out by the compiler. Typecasting is generally used when both of the data types in question are not exactly compatible with each other. Type conversion on the other hand mandates that both of the data types be compatible with each other. Typecasting requires the casting operator () , but type conversion does not require this. Typecasting is done while still writing the code while type conversion occurs during compilation.

Implicit typecasting

Okay, so basically typecasting results in conversion. Now that we've cleared the air let's take a close look at a type of casting that is better known as implicit typecasting. This basically means that the conversion of the data type does not alter the original value of the data. This is essential when the accuracy of data is of primary importance. This type of data type conversion happens automatically when a value is copied into a compatible data type.

A number of rules apply to this type of conversion:

If the variables in question are of two different data types, then the computer compares the two operands and the operand with a lower data type will be automatically converted into a higher data type. This uses the data type hierarchy mentioned earlier. Basically, the computer uses this pretty long and self-explanatory if statement: All short and char are automatically converted to int, then,

If either of the operand is of type long double, then others will be converted to long double and

Result will be long double. Else, if either of the operand is double, then others are converted to double.

Else, if either of the operand is float, then others are converted to float. Else, if either of the operand is unsigned long int, then others will be converted to unsigned long int. Else, if one of the operands is long int, and the other is unsigned int, then if a long int can represent all values of an unsigned int, the unsigned int is converted to long int. Otherwise, both operands are converted to unsigned long int.

Else, if either operand is long int then other will be converted to long int Else, if either operand is unsigned int then others will be converted to unsigned int. The final result of whatever expression that was worked on is converted to the data type of the variable on the left side of the assignment operator before assigning the value to it.

Implicit conversion

As mentioned before, this is done automatically and therefore does not require any special keywords or any special input from the programmer. This type of conversion will always happen whenever the compiler encounters compatible data types if the conversion causes data loss, such as in the example that we looked at previously, where we wrote a programme that runs of decimal numbers, the process will not happen automatically. Typically, if your code attempts to do this, you will receive a warning during compilation. If you are stubborn and want to do this anyway, then you would have to resort to explicit typecasting.

Built-in type conversion

Another way in which we can alter a variable's data type is with a built-in type conversion functions.

C provides a number of built-in functions to perform data type conversion. There are a number of functions that are generally targeted at strings. The standard library header file (stdlib.h) provides this functionality. The built-in functions include the following:

atof() Converts a string to float

atoi() Converts a string to int

atol() Converts a string to long

itoa() Converts int to string

ltoa() Converts long to string

Let's look at them individually.

atof() converts variables of the data type string to the float data type. The general syntax looks like this:

```
double atof (const char* string);
```

"String" represents the parameter that is passed to the function which will be of the data type string.

Pretty obvious but doesn't hurt to mention it anyway. The function returns the converted floating-point number as a value of data type double. If the conversion could not be performed, for example, when an invalid parameter is supplied, the function will return 0.

The atoi() function converts variables of the data type string to an integer. Its general syntax looks like this:

```
int atoi(const char* string)
```

Again here string represents the parameter that is passed to the function. The function will return the converted value as an integer value. If the conversion could not be completed, the function returns a 0.

atol()
The atoi() function converts variables of the data type string to an long. Its general syntax looks like this:

```
int atol(const char *string)
```

The same rules in the functions that we just looked at apply here string represents the parameter that is passed to the function.

The function will return the converted value as a long value. If the conversion could not be completed, the function returns a 0.

It's awesome to be able to convert strings to other data types, but what if we want to convert other data types to Strings? The remainder functions on our list do exactly that. Let's have a brief look at each of them.

ltoa()
this function convert a variable of an integer data type to a string what stop its basic syntax looks like this:

```
char *  itoa ( int value, char * str, int base );
```

Finally refers to the value that is to be converted to a string. Str represents the array in memory with this result is going to be stored as a null terminated string. Base the first of the new medical base that is used to represent value as a string. The function will return a pointer to the insulting null terminated string.

Typedef

The keywords typedef is used in C to assign alternative names to existing data types. It's typically used on user defined data types when data type names become slightly complicated. You could think of this as sort of giving nicknames to user defined data types. This is essentially creating an alternative name for a data type, commonly referred to as an alias for another data type. I would want to explicitly note that typedef does not create a new type except in the case of a qualified typedef of an array type. Here the types of qualifiers are transferred to the array element type. The reason why we are mentioning it now is because typedef is typically used to simplify the syntax of complex data structures that consist of unions and structs. The basic syntax of it typedef declaration is pretty simple. The basic syntax looks like this:

```
typedef type-definition identifier
```

typedef vs #define

#define is a C-directive which is also used to define the aliases for various data types, much in the same way as we described typedef. There are a number of key differences between the two.

Typedef is limited to assigning symbolic names to data types while #define can be used to define aliases for data types as well as values

Typedef is handled by the compiler while #define statements are processed by the pre-processor. If you recall from our first module, this has implications in the way that errors are handled.

Demo

Our demo is quite extensive and it's going to take quite a bit of time, so we saved it for last. Be sure to follow along so that you can get a better understanding of what is really going on. Remember you can always go back to the recording to look at what we have done in the lesson at your own pace. Enough chitchat let's get coding!

Conclusion

In this lesson, we took off from where we left off last time and had another look at structs. To really cement this, we had a lengthy example that really highlighted what we can do using structurers. This is critical since data structures in general, not just structs, are at the core of data processing in computing and Computer Science. In our next lesson, we look at libraries and header files, exploring what they really contain and how you as a programmer can make the most out of that rich treasure trove. We are nearing the end of our third module too.

References

Professor, A., Etuari, M., Asst, O., Mr, B. and Naik (n.d.). LECTURE NOTE on PROGRAMMING IN “C” COURSE CODE: MCA 101. [online] Available at: https://www.vssut.ac.in/lecture_notes/lecture1424354156.pdf

Dot Net Tutorials. (2020). Type Casting in C Program with Examples - Dot Net Tutorials. [online] Available at: https://dotnettutorials.net/lesson/type-casting-in-c/

DataFlair. (2019). Typecasting in C/C++ | Uncover Difference between Typecasting & Type Conversion - DataFlair. [online] Available at: https://data-flair.training/blogs/typecasting-in-c-cpp/

Vineet Choudhary (2016). Type Casting - C Programming. [online] Developer Insider. Available at: https://developerinsider.co/type-casting-c-programming/

Utsa.edu. (2021). 2-dimensional Arrays in C. [online] Available at: http://www.cs.utsa.edu/~wagner/CS2073/fraction/fractions.html

Kenneth Leroy Busbee and Braunschweig, D. (2018). Data Type Conversions. [online] Rebus.community. Available at: https://press.rebus.community/programmingfundamentals/chapter/data-type-conversions/

Engineering LibreTexts. (2019). 4.7: Data Type Conversions. [online] Available at: https://eng.libretexts.org/Bookshelves/Computer_Science/Book3A_Programming_Fundamentals_-_A_Modular_Structured_Approach_using_C___(Busbee)/04%3A_Data_and_Operators/4.07%3A_Data_Type_Conversions