What are the basic data types in C?

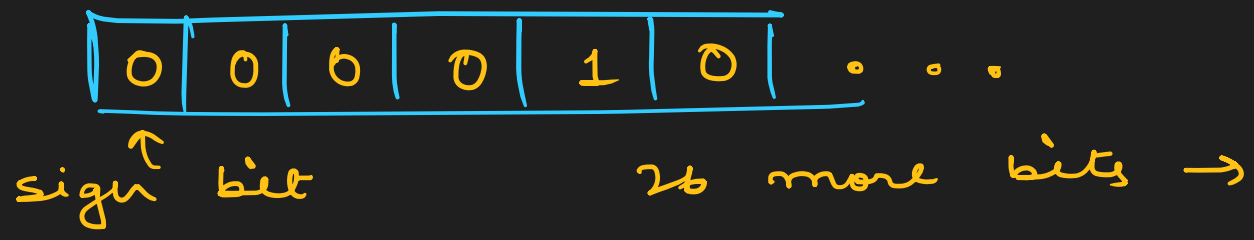
1. A program contains data, it may be numerical data for doing calculations or text data for displaying output.
2. C has the following basic data types:
    a. int
    b. char
    c. float
    d. double
    e. void

What is data type anyway?

1. We will go low level for a bit. Data types are abstractions created by a programming language so that we don't have to bother with bit level data, but under the hood every type of data is just 0s and 1s in the memory. A single 0 or 1 is not helpful, so we need a group of 0s and 1s. So, data types are just groups of 0s and 1s.
2. The amount of 0s and 1s in each data type may be different for C compilers of different architectures. What I am using below is what our modern computers generally use.

How does the int data type look like?

1. int is just 32 bits (or 4 bytes) of 0s and 1s which can store a 32-bit binary number. But it can only store 31-bit numbers because it uses the 1st bit to store the sign (i.e. whether the number is positive or negative). Negative sign is represented by 1 and non-negative sign is represented by 0.

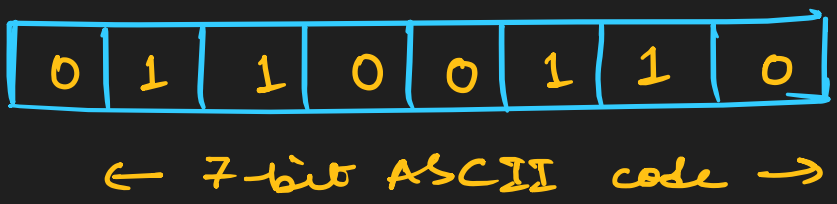


    Bonus challenge: Imagine that you are inspecting your program's memory and find that an integer starts with the above 6 bits, what is the minimum possible value of this integer?
2. An integer can also be unsigned. Just add unsigned before int when declaring the variable. Eg: unsigned int a = 2001; In the case of unsigned integers, we don't need a sign bit, so we can store 32-bit integers.

How does the char data type look like?

1. char is also simple; it is just an 8-bit integer.
2. Some American organization (IBM) prepared a chart of 128 characters used by American people. Then the American government standardized the chart and made the entire world use it. It is called the ASCII table.
3. So back to char again, char is an 8-bit integer representing the character code from 0 to 127. Why do we need 8 bits to store a 7-bit integer? Because CPU accesses memory byte after byte, so it is 8 bits or 1 byte. The 1st bit of a char

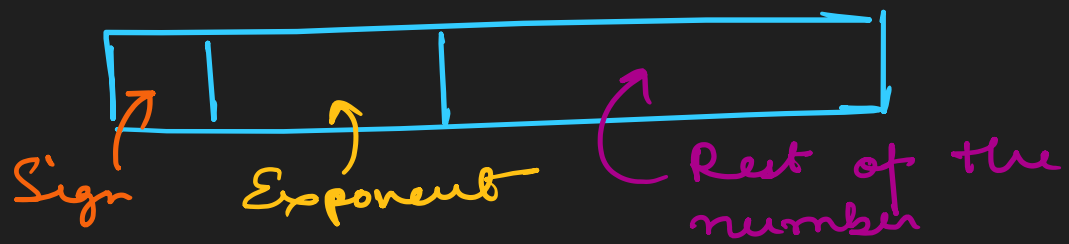
is the signed bit, just like an integer, but it doesn't make much sense to create a negative A.



Bonus challenge: Lookup the ASCII table on the Internet and try to figure out which American character this is.

How does the float data type look like?

1. Point numbers are more complicated than integers, so I won't go into too much details.
2. You must have studied about scientific notations of point numbers; it is the way of writing -3523.545 as $-3.523545 * 10^{3}$. Computers use a similar approach to store point numbers, but in binary.



Interestingly, not all point numbers can be represented by using binary number system, it is just like how we cannot represent point numbers like 1/3 in decimal number system, and the reason for why 0.1 + 0.2 is 0.30000000000000004!
I would've added some bonus challenge related to this, but I fear that will be too complicated (you may try and learn binary division and IEEE 754 floating point representation to analyse why so many programming languages mess up this simple calculation (if you really hate yourself or if you have nothing better to do)).

How is double data type different from float?

1. Not much different, just more bits for exponent and the rest of the numbers.
2. In fact, float is 32 bits or 4 bytes and double is 64 bits or 8 bytes.

What about void?

1. I don't think it is stored in memory.
2. It just means nothing, literally not structurally.

Can we finally start type casting now?

1. Sure, we have most of the tools needed to understand type casting.
2. Type casting means converting one data type into another.
3. In C, there are two types of type casting:
    a. Explicit casting
    b. Implicit casting

What is explicit casting?

explicit

/ɪkˈsplɪsɪt,ɛkˈsplɪsɪt/

adjective

programmers specify which type to convert data into

stated clearly and in detail, leaving no room for confusion or doubt.

1. If we want to convert a data into a type, we write (type)data.
2. Eg: (char)45 casts 45 (int) into '-' (char).

What is implicit casting?

implicit

/ɪmˈplɪsɪt/

we do not directly specify the conversion but the conversion is implied (?)

adjective

1. suggested though not directly expressed.

1. What do I mean by 'the conversion is implied'? It means that although we did not specify any type conversion, C still understands what to do because of the context the data is used in.
2. Eg: if we try to assign a char into an int, C will automatically convert char to int. And it is not just for assignment, for any other operation we may use, C will convert type of the operand to make the operation successful.
3. So, the code snippet: int i = ('a' + 'A') / 2, will store 81 in the variable a, while the code snippet: char c = ('a' + 'A') / 2 will store 'Q'.

What is the real difference between C data types anyway?

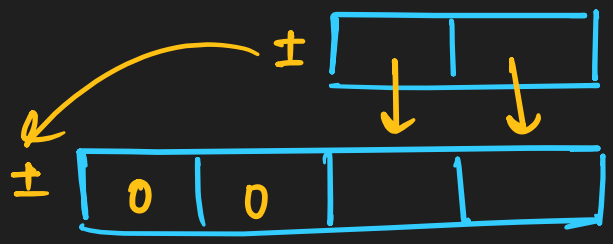
1. The only difference between C data types is how much space is required to store them and how they are represented.
2. int is simply binary numbers, char is just a smaller number, float is just a set of number and exponent, double is just a bigger float, and void is nothing.
3. There are also pointer types, but we will look at them when we study pointers.

Okay, but are there only these five types?

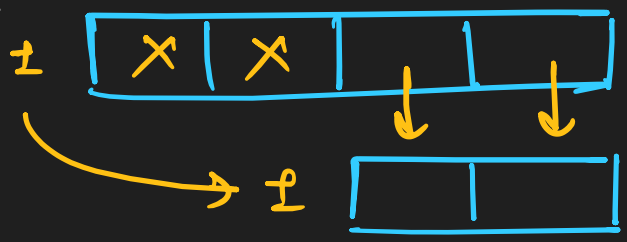
1. Unfortunately, there are more.
2. There are pointers, which are just bigger integers.
3. Then there are also modifiers.
4. long is one such modifier, which makes an already existing data type longer. Eg: long int is 8 bytes long (twice that of int)!
5. short is the opposite of long, it makes data types shorter. short int is 2 bytes long (the half of int), so it can be used for smaller numbers.
6. unsigned is useful. As we have seen so far, every data type has a sign bit; well unsigned types don't! If we define an unsigned int, then we can store 32-bit integers because there won't be any sign bit. The format specifier for unsigned is %u btw.
7. bool has also been newly introduced in C, it is available in newer C versions, but we can use it in older C versions too by including stdbool.h into our file.

Can we finally end type casting now?

1. Sure, but after one (or more) final thing(s).
2. When we are casting a smaller int into a bigger int, eg: short int into int, it copies the 2 bytes of the short to the last 2 bytes of the int and fills the bytes before it with 0s. The sign bit of short is copied to the sign bit of the int, and not the 16th bit.



3. But when we are doing the opposite, i.e. bigger int into smaller int, eg: int into short, it copies the last two bytes of the int into the short, and the first two bytes are lost. Just like the previous case, the sign bit of the int is copied to the sign bit of the short, so only 15 bits worth of value is only copied.



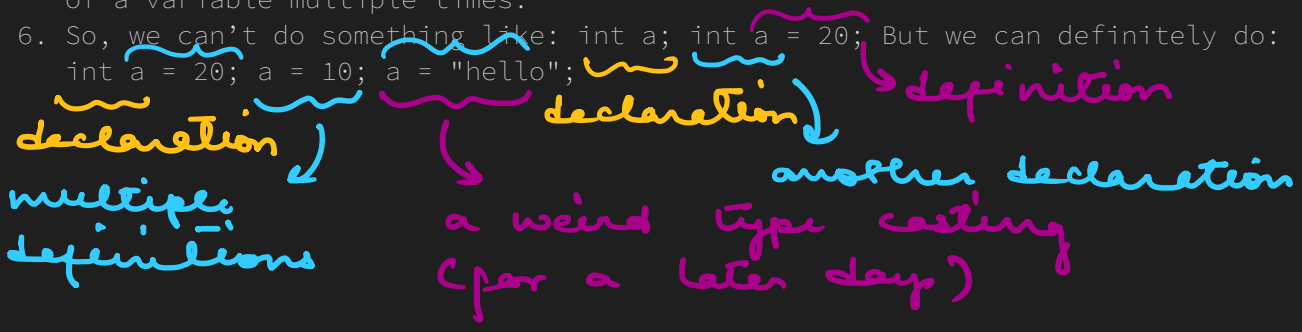
4. When we type cast int to char or char to int, it just converts from int to ASCII value and vice versa. Eg: 'A' to 65 and 97 to 'a' (refer to ASCII table maybe?).
5. Bonus challenge, what will be the content of the variable a defined as:
   ```
   char a = 0x4e;
   ```
6. Finally, when we type cast float to int, it removes the part after point and just becomes the integral part of it. And in the opposite case, when we are trying to cast int to float, it just adds a point after the int and adds 0s after the point. Eg: 1.999 become 1 (note that this is not rounding off, this is similar to floor) and 1 becomes 1.000 (thus the part after the floating point is lost if we do (float)(int)1.999 (why not try it yourself? I may be wrong.))
7. Bonus challenge! We have already seen how a float is represented in memory. If you are given that the exponent part is 0b101 and the rest of the number is 0b11010001001110, how can you find out what the integral part of the float is? I.e. how can you find out the result after type casting the float to int?

Now we know about data and types, but what are variables?

1. We understand now how data is stored in memory, but we don't need to remember the memory address where the data is stored (we do when we are using pointers) because we use variables for that.
2. Variables remember the starting memory address and the type of the data, so lookup is easy that way.
3. We just have to declare variables using the data type, like int age or float gender, and we can use it in our program.

While in that context, what is declaration and definition?

1. Declaration is when we just tell the compiler to create a location in memory for our data but don't tell what data to store.
2. We can do that by writing the variable type followed by the variable name like: int a; or float f; or char c; or unsigned u;
3. We don't specify any data along with it, but when we do, it is definition.
4. We use the assignment operator (=) to do that like: int a = 20; or float f = 1.12;
5. We can only declare a variable (or anything for that matter) once in a scope (what is a scope? We will discuss it some other day), but we can change the value of a variable multiple times.
6. So, we can't do something like: int a; int a = 20; But we can definitely do: int a = 20; a = 10; a = "hello";



When will we go to coding stuffs?

1. Right now, so let's look at something basic - a form of flow control with if and else.
2. What is flow control? Code is executed line by line, but sometimes we may want to jump from one line to another. Statements which allow us to jump from one line to another are called flow control statements. For now, we will look at if and else.

What are if and else statements?

1. if is followed by a condition enclosed within round brackets. Then it has a block of code within curly braces. The block of code is executed if the condition is true.

```
if (n % 2 == 0) {
      printf("%d is even!\n", n);
}
```

2. An if statement can be followed by an else statement. The if block is executed when the condition provided after the if statement is true. The following else statement does not have any condition, and is executed if the if statement before it is false.

```
if (n % 2 == 0) {
      printf("%d is even!\n", n);
} else {
      printf("%d is odd!\n", n);
}
```

3. In fact, the if condition may not be a condition at all, it can be an expression too.

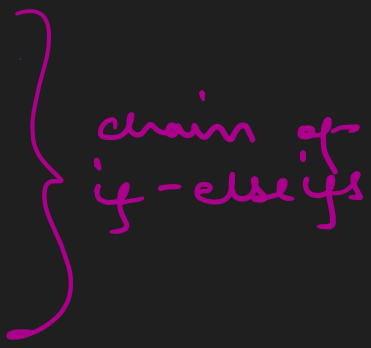
```
if (n % 2) {
      printf("%d is odd!\n", n);
} else {
      printf("%d is even!\n", n);
}
```

   In this example, if n is even, n % 2 will be 0 and 0 means false (anything other than 0 means true, even negative numbers), so if n % 2 is true, then n must be odd, else it must be even.

How to handle multiple conditions?

1. We can insert else if statements after if statement for that.

```
if (dogPerson) {
        printf("woof\n");
} else if (catPerson) {
        printf("meow\n");
} else if (hamsterPerson) {
        printf("squeak\n");
} else if (dinosaurPerson) {
        printf("rawrrrr\n");
} else {
        printf("you like some weird animal\n");
}
```

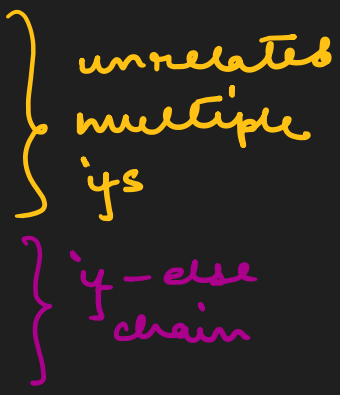


2. When we have statements like this, C sequentially checks each condition; when it finds a true condition, it executes the block of code after it and gets out of the if-else structure.
3. Let's look at an example when a person likes dinosaurs.
    a. Does the person like dog? No.
    b. Does the person like cat? No.
    c. Does the person like hamster? No.
    d. Does the person like dinosaur? Yes!
        i. Print some dinosaur noises.
        ii. Come out of the if-else chain.
4. This is how C thinks when it encounters this code, it is skipping some lines of code based on the conditions. Go two questions back and see if you understand what I said about flow of control there better now.
5. If you really went to check, then that was also a flow of control. We generally read questions one by one, but I told you to jump two questions back; and after that you returned back to this place.
6. There are more ways of flow control. Switch-case, loops and jumps. But we will look at them some other day.

What if we just do multiple ifs?

1. We can do that.

```
if (dogPerson) {
        printf("woof\n");
} if (catPerson) {
        printf("meow\n");
} if (hamsterPerson) {
        printf("squeak\n");
} if (dinosaurPerson) {
        printf("rawrrrr\n");
} else {
        printf("you like some weird animal\n");
}
```



2. In if-elseif, the next condition is only checked if the current condition is not true (that's why it's *else* if), but in the case of multiple ifs, the next condition is checked even if the current if is true. Let's understand that with an example: consider a person likes dogs.
    a. Does the person like dogs? Yes.
        i. Print dog noises.
        ii. Proceed to the next if.
    b. Does the person like cats? No.
    c. Does the person like hamsters? No.

d. Does the person like dinosaurs? No.
    i. Execute the else part (this part shouldn't have been executed, but it still is because the multiple ifs are individual units and not a chain like if-elseifs).
    ii. Print about the weird tastes of the user.

What to do at home?

1. Take obtained marks and total marks (both floats) as input from the user. Given, a student passes an exam if he gets more than 40%, print a non-encouraging message if the student has failed the exam, and nothing if he has passed.
2. Bonus challenge: can you do the same thing but by using ternary operator (?:) instead of if-else? (Note: this hasn't been discussed formally, but I hope you can research that on your own ;).)

Another note: This recap has been extraordinarily long because I added a lot of things that was not covered in class like the memory representation and stuffs, they are only to give an intuition to how interesting things are under the hood. You don't need to understand all that to write simple code in C, but you can flex your big brains in front of other people if you do :D.