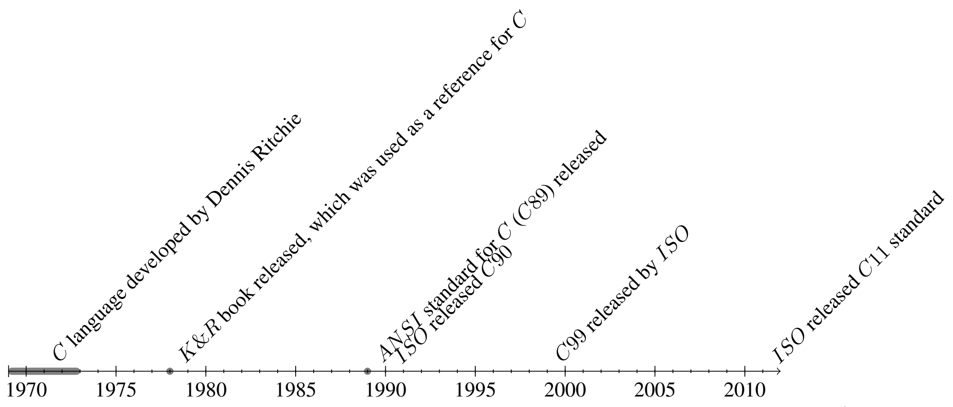
CHAPTER 2

DATA TYPES AND OPERATORS IN C

C is a just a language specification and C compilers (which translates C code to machine code) are made by different organizations/individuals. So, in order to make a C program work across multiple compilers, C standard is important.

2.1 A little about C standard



Till $ANSI$ standard which is almost the same as $C90$ (also called $C89/C90$) was published, the standard specified in $K\&R$ book was used as a reference for C standard. Each new standard aimed at easing the programming difficulty as well as making use of the new hardware features. We'll be following as much of $C11$ standard as possible as it adds some significant changes to C language.

2.2 Data types

Data types are used to represent data and data comes from real world. In real world we have the following data types

- Integer
- Character
- Real Numbers

All other data types can be formed from these basic types (for example, a string is just a sequence of characters). So, in C language we have only these basic types but they are supported in various data sizes as follows

- short int
- int
- long int
- long long int
- char
- float
- double
- long double

Now, each of these type is supported in unsigned version also. In signed version, one bit is used to identify if the number is positive or negative. So, for a 32 bit signed integer, it can represent only up to $2^{31} - 1$, while a 32 bit unsigned integer can represent up to $2^{32} - 1$. A boolean data type is also present in C standard which can be used to hold a bit. This data type can be used using the keyword $\_Bool$. If we use an int (or $char$), to get a boolean value, we need to logical AND (&) it with 1. With $\_Bool$, this conversion is not necessary as the byte representing the boolean will always have the most significant 7 bits as 0.

So, let's see what these data types mean to computer. As we have seen in the previous chapter, all data must be converted into a bit stream before being given to the processor. So, even though we use alphabets and digits while writing programs, they are converted to bits while stored in memory and then given to processor. How many bits a data type takes is defined by the $sizeof$ the data type. In C language, the operator $sizeof$ gives the no. of bytes (i.e., 8 * no. of bits) a data type takes. Since, memory accesses are restricted to multiple of bytes (RAM doesn't allow to access data at a granularity lower than 8 bits at a time due to practical reasons) $sizeof$ always return at least 1 for any data type. (A $\_Bool$ also take a byte of storage as that's the smallest accessible unit in a memory, though it actually requires just a bit of storage)

Now, lets see the $sizeof$ the various data types in C. Since, the data types are directly given to the processor, the $sizeof$ data types depend on the processor architecture. So, C standard just tells the minimum required size specification and have let the compiler designers choose their size as per the processor architecture- C compilers are used for 8 bit embedded processors to 64 bit desktop processors. So, this size variation does make sense.

Data type	Min. size required
char	1
short int	2
int	2
long int	4
long long int	8
float	4
double	8
long double	10

$sizeof\ char$ is 1 byte as it was sufficient for $ASCII$ encoding. But for extended $ASCII$ character support, $wchar\_t$ which supports up to 16 bits is defined in $< stddef.h >$. $C11$ standard also defines $char16\_t$ or $char32\_t$ in $< uchar.h >$ header file thus supporting Unicode characters which requires up to 21 bits. So, in today's world, a $char$ and an int take the same size and integer variable ($char32\_t$ of 4 bytes) is used to store Unicode characters and the data type $char$ is used mainly to refer to a $byte$ of data than an actual character.

2.3 Constants and variables

We have seen the data types, but to use them in a program we need to have a variable. A $variable$ is a named entity to represent a specific data type. The type of a variable is fixed during the program run, but its value can be changed, and hence the name variable. (In an object oriented language like CPP, a class can be taken as a data type and its instance become a variable). To assign a value to a variable we use $constants$. The following are the example usage of variables and constants.

```
int a; // 'a' is an int variable
a = 5; // 5 is an int constant
```

2.3.1 Constant Types

In C language we have the following constants

- Integer constant
  - Decimal constant
  - Octal constant
  - Hexadecimal constant
- Floating constant
- Character constant
- Enumeration constant

2.3.1.1 Integer Constant C standard supports decimal, octal and hexa-decimal constants being used to assign integer values. Their example usage is as shown below.

```
#include <stdio.h>


int main()
{
    enum month{jan = 1, feb, mar, apr, may, jun, july, aug, sep, oct,
        nov, dec};//jan is having int value 1, feb value 2 and so on
    int a,b,c;
    a = 10; //10 is a decimal constant
    b = 0xa; // a is a hexadecimal constant
    c = 012; //12 is an octal constant
    enum month d = oct; //oct is an enumeration constant

    printf("a = %d, b = %d, c = %d, d = %d ",a,b,c,d);
    return 0;
}
```

We can use an int instead of $enum$ as both takes same amount of memory. But the use of $enum$ ensures that a variable can hold only a particular set of integer values rather than the whole range of integers. Thus it leads to less program errors and makes the code more readable by providing a set of defined constants

Here, a is having the decimal value of 10. So, in memory a will be like

```
000...1010
```

Similarly, b will be in memory like

```
000...1010
```

and c and d will also be like

```
000...1010
```

i.e.; all a, b, c and d are having same integer values given using different constants. The memory to be allotted to an integer constant is determined by its value, minimum being the $sizeof(int)$. For example, 40 is allotted the $sizeof(int)$ while 0xffffffff is allotted $sizeof(long)$ as it won't fit the $sizeof(int)$ (assuming $sizeof(int)$ is 4 and $sizeof(long)$ is 8).

2.3.1.2 Floating Constant The representation of a floating point number is implementation specific. $C11$ do specifies IEC 60559 format for floating point representation but its not mandatory that all implementations must support them. But most current implementations do support them and hence it's good to have a look at them. This link `http://steve.hollasch.net/cgindex/coding/ieeefloat.html` is a good look as $IEEE$ 754 is identical to IEC 60559.

Constant values can be assigned to float or double variables in various ways as shown below. If a constant cannot be exactly representable in the float or double variable the implementation is recommended to show a warning as per C standard. But this is just a recommendation and not a strict requirement.

```
#include <stdio.h>
int main()
{
    float a = 10.2;
    float b = 2.3f;
    double c = 3.4l;
    double d = 1.2e-3;
    printf("a = %.2f, b = %5.2f, c = %05.2lf, d = %le\n",a,b,c,d); //
        Just diff format specifiers
    //%.2f means 2 digits after decimal point will be printed
    //%5.2f means the output will have a total of minimum 5 places
        including 2 decimal digits and a point. If lesser digits are
        there, then the remaining space is filled with white space.
    //%05.2f is same as %5.2f except that the remaining space, if any,
        are filled with 0s than white space

    return 0;
}
```

2.3.1.3 Character Constant Characters can be assigned value either by using a character in single quotes or by giving the integer value from the character code. And this int value can be given using hex or octal representation as well, as shown below. Escape sequences are applicable to character constants like $'\backslash n'$, $'\backslash t'$ etc.

```
#include <stdio.h>
int main()
{
    char a, b, c, d, e;
    a = 'a';
    b = '\0';
    c = 0;
    d = '\x41'; //41 is a hexa decimal value whose corresponding ANSII
        char is assigned to d

    e = '\101'; //101 is an octal constant whose corresponding ASCII
        char is assigned to e

    printf("a = %d, b = %d, c = %d, d = %c e = %c",a,b,c,d,e);

    return 0;
}
```

Here, a is having the $ASCII$ value of $'a'$ which is 97. So, in memory a will be like

```
01100001
```

Similarly, b and c will be in memory like

```
00000000 //ASCII value of \0 is 0
```

and d and e will be like

```
01000001
```

2.3.1.4 Enumeration Constant Enumeration constants are assigned integer values starting from a given initial value which by default is 0. An example is shown below:

```
enum player{ Dhoni = 1, Kohli, Yuvraj, Aswin = 5, Jadeja, Mishra = 10};
```

Here, Dhoni is having an integer value 1, Kohli 2, Yuvraj 3, Aswin 5, Jadeja 6 and Mishra 10. That is, these names can be used wherever these values are needed.

2.4 String Literal

A character string literal is a sequence of zero or more characters enclosed in double-quotes, as in "xyz". A $UTF8$ string literal is the same, except prexed by $u8$. A wide string literal is the same, except prexed by the letter L, u, or U. All escape sequences applicable to a character constant is applicable for a string literal except that for $'$ an escape sequence is not mandatory. Any sequence of string literals will be combined into a single string literal during the translation phase of the compiler. Thus, "abc" "de" is equivalent to "$abcde$". Another important property of string literal is that it cannot be modified and is usually stored in the RO Data segment.

```
char p[] = "hello world";
char *q = "hello world";
```

Here, individual characters of p can be modified as the characters of the string literal "hello world" are copied to the memory allocated to p which is 12 bytes. But, individual characters of the content of q can only be read and not modifiable as p is pointing to a string literal, which is stored in the RO data segment of the program. i.e.

```
p[2] = 'p'; //valid
char c = q[2]; //valid
q[3] = 'q'; //Invalid
```

The last statement causes segmentation fault as explained in 1.1.1

2.5 Implicit Type Conversion

We can round off this chapter with an important point about implicit type conversion. Whenever we do an operation with different data types, the lower ranked data type is promoted to the higher ranked one, as, operations are meant to be performed on same types of data. For example when we add an int and a $float$, the int is promoted to $float$ and addition of two $floats$ takes place using two floating point registers. Similarly, when we add a $char$ and an int, the 8 bits of char is made into 32 bits (assuming 4 byte size for int), by padding it with $0's$.

One important point about implicit type conversion is that, it depends only on the source operands and is independent of the resultant data type. So, if we multiple two $integers$ and store in a $long$, the result will be calculated as int (usually 4 bytes) and then stored in $long$ (usually 8 bytes). Another common example of this behavior is for division operation. When we divide two $integers$, the result will be int only, even if we assign it to a $float$. So, in these cases the programmer has to explicitly cast one operand to the desired output type.

2.5.1 What exactly happens during type conversion

Before reading the description below, think how it can happen- you won't think wrong.

- $unsigned$ to $signed$ or vice verse: There is no change in the representation in memory. When casting to signed, the most significant bit is taken as a sign bit which would otherwise be used for representing the number. So, this type casting is necessary during conditional checks as a negative number when type casted to unsigned will give a huge int value.

- *char* to *int*: If *int* is 4 bytes- the top 3 bytes are filled with 0's and bottom most byte is the same byte used to represent the *char*.
- *int* to *char*: Only the lowermost byte of the *int* is taken and made to a *char*. So, if *int* value is 511, its char value will be 255.

```
00000000 00000000 01111111 11111111 //511
//11111111 is 255
```

- *int* to *float* or *double*: The fixed integer is converted to a representable floating point value. So, this might cause a change to entire bits used to represent the integer.
- *float* or *double* to *int*: The integral part of the floating point value is saved as integer. The decimal part is ignored. For example, 5.9 will be changed to 5.
- *float* to *double*: The extra mantissa bits supported in *double* are filled with 0's so are the extra exponent bits. The bits of float (32 of them) are used without modification.
- *double* to *float*: The extra mantissa bits supported in *double* are truncated in floating representation so do the extra exponent fields. If the truncated exponent field were non zeroes, it might cause a change to other mantissa bits as well as the number would then need an approximation to fit in a float size.

PROBLEMS

2.1 Consider an implementation where int is 4 bytes and long int is 8 bytes. Which of the following initializations are valid?

```c
#include <stdio.h>
int main()
{
   long int a = 0x7fffffff * 0x7ffffff;
   long int b = 0x7ffffffff * 0x7ffffff;
   long int c = 0x7fffffff * 0x7fffffff;
   long int d = 0x7fffffff * 0x7fffffffl;
   printf("a = %ld, b = %ld, c = %ld, d = %ld\n", a, b, c, d);

   return 0;
}
```

2.2 Consider an implementation where int is 4 bytes and long int is 8 bytes. What will be the output of the following code?

```c
#include <stdio.h>
int main()
{
    int i = 0;
    size_t  a = sizeof i, b = sizeof (long);
    printf("a = %zd, b = %zd\n", a, b); //If %zd is given the compiler
        will automatically give it the correct type whether short, long
        or normal. This is useful for special data types like size_t
        whose size is implementation specific

    return 0;
}
```

2.3 What will be printed by the following code?

```
#include <stdio.h>
#include <string.h>

int main()
{

   char buff[255] = "abc\
pee\n";

   printf("%s", buff);

   return 0;
}
```

2.4 What will be printed by the following code?

```
#include <stdio.h>
#include<string.h>
int main()
{

   char buff[] = "abc" "hello";

   printf("%zd\n", strlen(buff));

   return 0;
}
```

2.5 How can you print the following sentence exactly as it is by changing the assignment to buff?

```
"Hello\\" "World\\"
```

```
#include <stdio.h>
#include <string.h>

int main()
{

   char buff[255] = "\0";

   printf("%s", buff);

   return 0;
}
```

2.6 What will be the output of the following code?

```
#include <stdio.h>

int main()
{

   char *p = "Hello World";
   char q[] = "Hello World";

   printf("%zd %zd", $sizeof$ p,  $sizeof$ *p);
   printf("\n");
   printf("%zd %zd", $sizeof$ q,  $sizeof$ *q);

   return 0;
}
```

2.7 What will be the output of the following code?

```
#include <stdio.h>

int main()
{
    {
        char a = 5;
        int b = 5;
        if(a == b)
            printf("char and int compared equal\n");
    }

    {
        int a = 5;
        long int b = 5;
        if(a == b)
            printf("int and long compared equal\n");
    }
    {
        float a = 5.0;
        double  b = 5.0;
        if(a == b)
            printf("float and double compared equal\n");
    }
    {
        float a = 5.2;
        double  b = 5.2;
        if(a == b)
            printf("float and double again compared equal\n");
    }
    {
        float a = 5.2;
        if(a == 5.2)
            printf("float compared equal with constant\n");
    }
    {
        double a = 5.2;
        if(a == 5.2)
            printf("double compared equal with constant\n");
    }

   return 0;

}
```

2.8 What will be the output of the following code?

```
#include <stdio.h>

int main()
{
    int a = 5;
    if(a > -1)
        printf("5 is > -1\n");

    return 0;
}
```

2.9 Both int and $float$ on gcc takes 4 bytes and $float$ can represent a wider range of numbers than an int. Then what's the advantage of int compared to $float$ or why can't we just replace int with $float$?

2.10 Is there anything wrong with the following code? If so, how can we correct it?

```c
#include <stdio.h>

int main()
{
        int a, b;
        scanf("%d%d", &a, &b);
        float result = a/b;
        printf("%d divided by %d is %f\n", a, b, result);

        return 0;
}
```