Default Data Types and Variables in C

In order to understand variables and Default Data Types in C, we first need to understand how data is stored in computer.

Memory in Your Computer

The **instructions** that make up your program, and the **data** that it acts upon, have to be **stored somewhere that's instantly accessible** while your computer(Embedded System) is executing that program.

When your program is running, the program instructions and data are stored in the **main memory or the random access memory (RAM)** of the machine. RAM is volatile storage. When you **power off** your device, the **contents of RAM are lost**. You think if RAM as a **Temporary Workspace**.

Your PC has permanent storage in the form of one or more disk drives. In embedded systems, you have Flash memory(for both code and data). Anything you want to keep when a program finishes executing needs to be printed or written to disk, because when the program ends, the results stored in RAM will be lost.

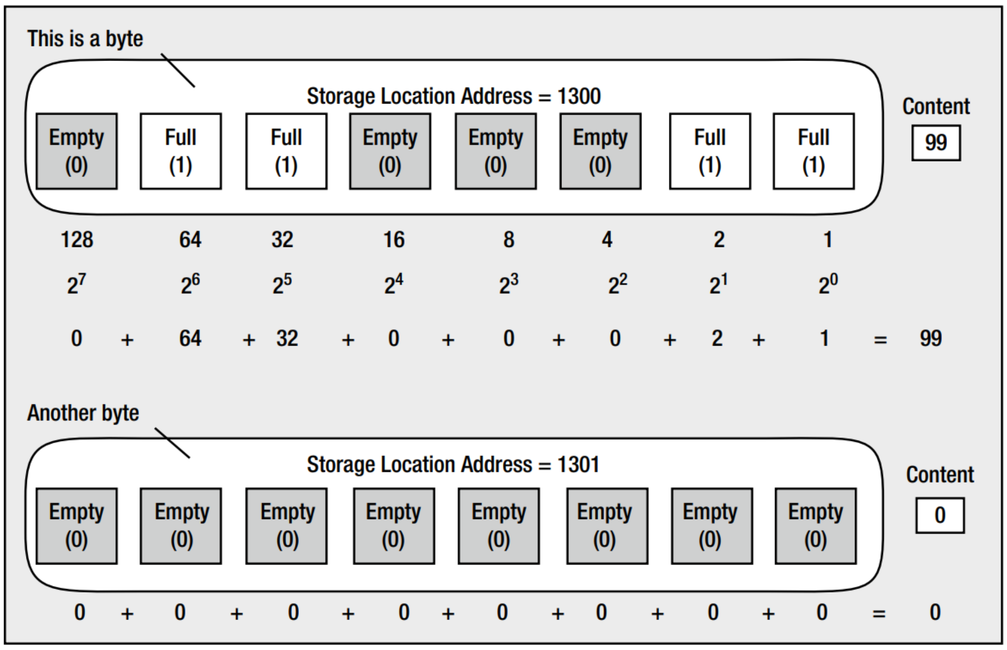
You can think of RAM as an ordered sequence of boxes. Each of these boxes is in one of two states: either the box is full when it represents 1 or the box is empty when it represents 0. Therefore, each box represents one binary digit, either 0 or 1. The computer sometimes thinks of these in terms of true and false: 1 is true and 0 is false. Each of these boxes is called a bit, which is a contraction of binary digit.

You can think of RAM as an ordered sequence of binary digits. A binary digit, commonly known as a bit, is the smallest unit of digital information. Each binary digit is either 0 or 1. The computer sometimes thinks of these in terms of true and false: 1 is true and 0 is false. A bit can only be either 1 or 0.

For convenience, the bits in memory are grouped into sets of eight, and each set of eight bits is called a byte. To allow you to refer to the contents of a particular byte, each byte has been labeled with a number, starting from 0 for the first byte, 1 for the second byte, and so on, up to whatever number of bytes you have in your computer's memory. This label for a byte is called its address. Thus, the address of each byte is unique. Just as a street address identifies a particular house, the address of a byte uniquely references that byte in your computer's memory.

In short, memory consists of a large number of bits that are in groups of eight (called bytes) and each byte has a unique address. Byte addresses start from 0.

Image below shows the storage location of two bytes and how value of those bytes are calculated which is simply converting binary value to decimal.



First byte is located at address 1300 and its value is 99. The second byte is located at address 1301 and its value is 0.

Fundamental Data Types in C

In programming, **a data type is a classification that specifies which type of value the compiler can understand**. Data type **defines the set of operations that can be performed on the data and the meaning of those operations**.

Integer Types:

There are five signed integer types. Most of these types can be designated by several synonyms, which are listed below:

Type	Synonyms
signed char	
int	signed, signed int
short	short int, signed short, signed short int
long	long int, signed long, signed long int
long long	long long int, signed long long, signed long long int

For each of the five signed integer types in table above, there is also a corresponding unsigned type that occupies the same amount of memory, with the same alignment.

Type	Synonyms
_Bool	bool
unsigned char	
unsigned int	unsigned

Type	Synonyms
unsigned short	unsigned short int
unsigned long	unsigned long int
unsigned long long	unsigned long long int

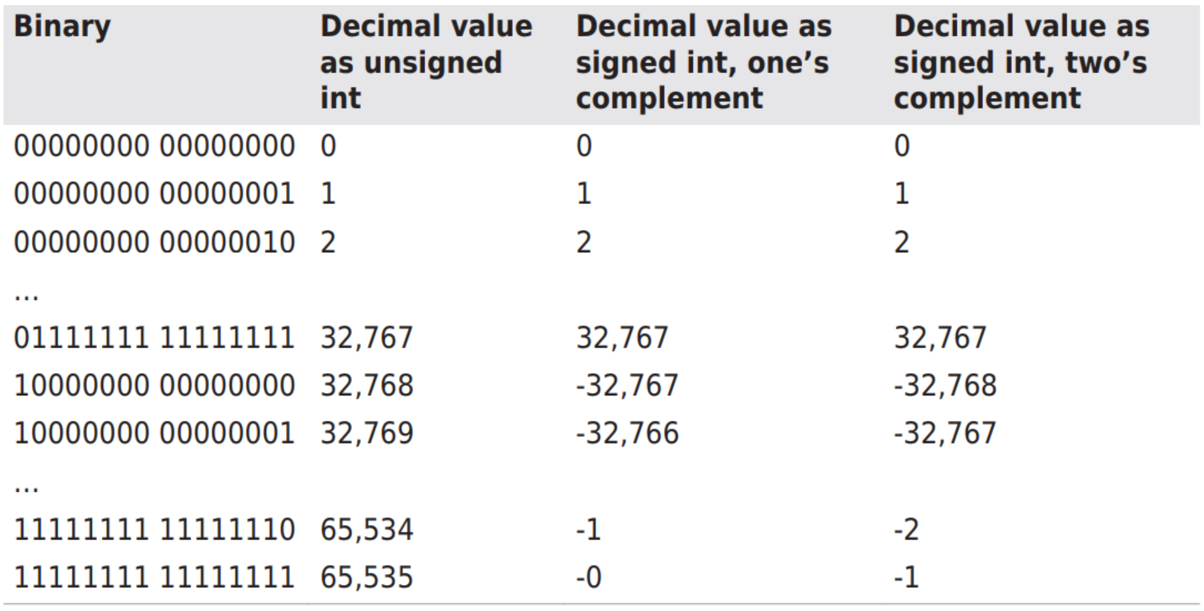
The type char is also one of the standard integer types. You can do arithmetic with character variables. It’s up to you to decide whether your program interprets the number in a char variable as a character code or as something else.

C99 introduced the unsigned integer type _Bool to represent Boolean truth values. The Boolean value true is coded as 1, and false is coded as 0.

The **type int** is the integer type best adapted to the target system’s architecture, with **the size and bit format of a CPU register**.

The internal representation of integer types is binary. Signed types may be represented in binary as sign and magnitude, as a one’s complement , or as a two’s complement . The most common representation is the two’s complement.

Below is the binary representations of signed and unsigned 16-bit integers.



Binary	Decimal value as unsigned int	Decimal value as signed int, one’s complement	Decimal value as signed int, two’s complement
00000000 00000000	0	0	0
00000000 00000001	1	1	1
00000000 00000010	2	2	2
...			
01111111 11111111	32,767	32,767	32,767
10000000 00000000	32,768	-32,767	-32,768
10000000 00000001	32,769	-32,766	-32,767
...			
11111111 11111110	65,534	-1	-2
11111111 11111111	65,535	-0	-1

Common storage sizes and value ranges of standard integer types:

Type	Storage size	Minimum value	Maximum value
char	(same as either signed char or unsigned char)		
unsigned char	one byte	0	255
signed char	one byte	-128	127
int	two bytes or four bytes	-32,768 or -2,147,483,648	32,767 or 2,147,483,647
unsigned int	two bytes or four bytes	0	65,535 or 4,294,967,295
short	two bytes	-32,768	32,767
unsigned short	two bytes	0	65,535
long	four bytes	-2,147,483,648	2,147,483,647
unsigned long	four bytes	0	4,294,967,295
long long(C99)	eight bytes	-9,223,372,036, 854,775,808	9,223,372,036, 854,775,807
unsigned long long (C99)	eight bytes	0	18,446,744,073, 709,551,615

Floating-Point Types:

C also includes special numeric types that can represent non-integers with a decimal point in any position. The standard floating-point types for calculations with real numbers are as follows:

1. float : For variables with single precision
2. double : For variables with double precision
3. long double : For variables with extended precision

- Single Precision:

  Size: Single-precision floating-point numbers are 32 bits in size, divided into three components:

  ```
    Sign bit: 1 bit (0 for positive, 1 for negative)
    Exponent: 8 bits
    Significand: 23 bits
  ```

  Precision: Single precision provides about 7 decimal digits of precision. This means that the fractional part of a number can be accurate up to seven digits.

  Example - The number 3.14 might be represented as follows:

  ```
    Sign bit: 0 (positive)
    Exponent: 10000000 (biased exponent, which represents 128 in decimal)
    Significand: 10010010000111101011100 (in binary)
    Combining these, the binary representation might be:
  01000000010010010000111101011100
  ```

For detailed calculation visit here

- Double Precision:

  Size: Double-precision floating-point numbers are 64 bits in size, divided into three components:

  ```
    Sign bit: 1 bit (0 for positive, 1 for negative)
    Exponent: 11 bits
    Significand: 52 bits
  ```

  Precision: Single precision provides about 15 decimal digits of precision. This is considerably higher precision compared to single precision, making it suitable for applications that require more accurate representation of real numbers.

  Example - The number 3.14 might be represented as follows:

  ```
    Sign bit: 0 (positive)
    Exponent: 10000000000 (biased exponent, which represents 1023 in decimal)
    Significand: 10010010000111101011100001010001111010111000010100000101 (in
  binary)
    Combining these, the binary representation might be:
  010000000000100100100001111010111000010100011110101110000101000000101
  ```

In C, arithmetic operations with floating-point numbers are performed internally with double or greater precision. we will see floating point math in future.

Table below shows the value ranges and the precision of the real floating-point types:

Type	Storage size	Value range	Smallest positive value	Precision
float	4 bytes	±3.4E+38	1.2E-38	6 digits
double	8 bytes	±1.7E+308	2.3E-308	15 digits
long double	10 bytes	±1.1E+4932	3.4E-4932	19 digits

Variable

A variable in a program is a specific piece of memory that consists of one or more contiguous bytes, typically 1, 2, 4, 8 or 16 bytes. Every variable in a program has a name, which will correspond to the memory address for the variable.

You use the variable name to store a data value in memory or retrieve the data that the memory contains.

Naming Variables:

The name you give to a variable, conveniently referred to as a variable name, can be defined with some flexibility. A variable name is a sequence of one or more :

- uppercase letters or lowercase letters

- digits and
- underscore characters (_)

Some valid variable names are:

```
Radius diameter Month_May Knotted_Wool D678
```

Some points to take care are:

- A variable name must not begin with a digit
- A variable name must not include characters other than letters, underscores, and digits

Another very important point to remember about variable names is that they are **case sensitive**.

Follow these conventions and guidelines when naming variables in C for better readability.

Descriptive Names: Choose names that clearly convey the purpose or meaning of the variable. Avoid Single-Letter Names. A person reading the code should be able to understand the role of the variable just by looking at its name.

```
// Good
int numberOfStudents;

// Avoid
int n;
```

CamelCase or Snake_case: Use either CamelCase or snake_case for variable names. Pick one style and be consistent throughout your codebase.

```
// CamelCase
int totalMarks;

// snake_case
int total_marks;
```

Avoid Keywords and Reserved Words: Don't use C keywords or reserved words as variable names. For example, avoid names like int, char, or while as they are reserved by the language.

```
// Avoid
int int = 5;
```

Meaningful Constants: If you're using constants, make sure they have meaningful names in uppercase.

```
// Constants
#define MAX_STUDENTS 100
```

Remember that good variable names contribute to code readability and maintainability. Choose names that make your code self-explanatory and easy to understand by other developers (or yourself) in the future.

Using Variables:

Lets understand the how to use variables in C with an example to add two numbers and print the result.

```
// Learn to use variables in C
#include <stdio.h>
int main()
{
    // Addition of two integers
    int first_number; // Declare a variable called first_number
    int second_number = 20; // this is called initialization
    int result;

    first_number = 10; // Store 10 in salary
    result = first_number + second_number; // perform addition and store the value
in result

    printf("Addition: %d + %d = %d.\n", first_number, second_number, result);
    return 0;
}
```

The statement below identifies the memory that you're using store the first number:

```
int first_number; // Declare a variable called first_number
```

this statement is called a variable declaration because it declares the name of the variable. The name, in this case, is first_number. Variable declaration ends with a semicolon.

declaration syntax:

```
<variable_type> variable_name;
```

The declaration for the variable, first_number, is also a definition because it causes some memory to be allocated to hold an integer value, which you can access using the name first_number.

Note: a declaration introduces a variable name, and a definition causes memory to be allocated for it.

The statement below is a simple arithmetic assignment statement:

```
first_number = 10; // Store 10 in first_number
```

I takes the value to the right of the equal sign and stores it in the variable on the left of the equal sign. Here you’re assigning that the variable first_number will have the value 10.

Both declaration and assigning value can be done at the same time as below.

```
int second_number = 20; // this is called initialization
```

This is called **initialization**. Which basically **stores the given value at the time of memory allocation for the variable**. It is always best to initialize a variable.

declaration syntax:

```
<variable_type> = arithmetic_expression;
```

The arithmetic expression on the right of the = operator specifies a calculation using values stored in variables or explicit numbers that are combined using arithmetic operators such as addition (+), subtraction (–), multiplication (*), and division (/). There are also other operators you can use in an arithmetic expression, we will see them in future.

The statement below performs the addition of two numbers and assigns to value to result.

```
result = first_number + second_number; // perform addition and store the value in
result
```

We will see more on Expressions and Assignment in future.

The result is printed in the below statement:

```
printf("Addition: %d + %d = %d.\n", first_number, second_number, result);
```

Here printf() accepts there are now four arguments inside the parentheses, separated by a commas. an argument is a value that’s passed to a function.

- The first argument is a **control string**, so called because **it controls how the output specified by the following argument or arguments is to be presented**. this is the character string between the double quotes. it is also referred to as a format string because it specifies the format of the data that are output.

- The second argument is the name of the variable, first_number. The control string in the first argument determines how the value of first_number will be displayed.
- The third argument is the name of the variable, first_number. The control string in the first argument determines how the value of second_number will be displayed.
- The fourth argument is the name of the variable, first_number. The control string in the first argument determines how the value of result will be displayed.

If you look carefully, you'll see **%d** embedded in the control string. this **is called a conversion specifier for the value of the variable**. Conversion specifiers determine how variable values are displayed on the screen. in other words, they specify the form to which an original binary value is to be converted before it is displayed.

Integer Types:

```
%d: Signed decimal integer
%u: Unsigned decimal integer
%o: Unsigned octal
%x or %X: Unsigned hexadecimal (lowercase or uppercase)
```

Floating-Point Types:

```
%f: Floating-point (decimal notation)
%e or %E: Floating-point (scientific/exponential notation)
%g or %G: Floating-point (chooses %f or %e based on precision and value)
```

Characters and Strings:

```
%c: Character
%s: String
```

Pointers:

```
%p: Pointer (prints the memory address)
```

Width and Precision:

```
%*.*f: Specifies width and precision for floating-point numbers dynamically.
```

Others:

```
%n: Returns the number of characters written so far (stores the value in an
integer pointer).
%%: Prints a literal percent character.
```

We will be using all these as and when we need them.