



C Programming

Lecture 7

Data Modifiers

*This material is developed by IMTSchool for educational use only
All copyrights are reserved*

Data Modifiers Summary

Data Modifiers are used to change some of the characteristics of the data types.

Sign	<i>signed</i>		<i>unsigned</i>	
Size	<i>short</i>		<i>long</i>	
Storage	<i>auto</i>	<i>register</i>	<i>extern</i>	<i>static</i>
Constant	<i>const</i>			
Volatility	<i>volatile</i>			

Rules

1- You can't use more than one modifier from the same category at the same time.

ex: You can't say *signed unsigned int*

2- No order, *unsigned short int* = *short unsigned int* = *int short unsigned* ...

3- Each modifier can be used only with certain data types.

ex: You can't say *signed float*

Signed Number representation

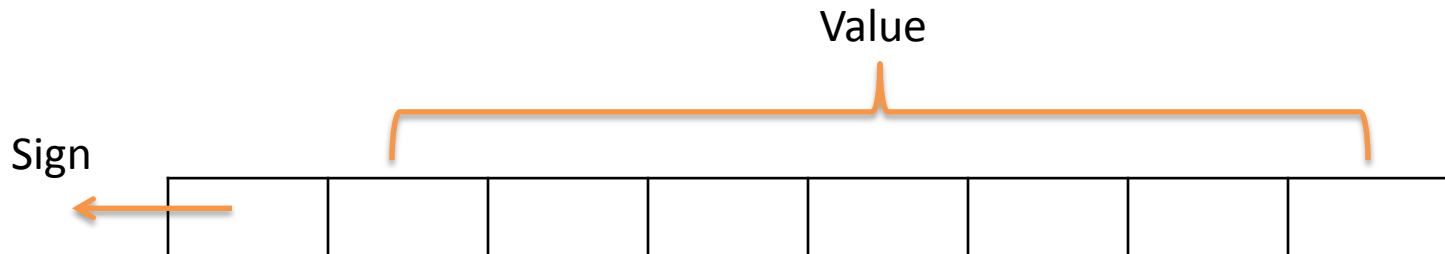
Q: How signed values are stored in memory ... ?

A: There are many techniques, each has its advantages and disadvantages.
Let's discuss some of these techniques ...

1) Signed magnitude

This is the simplest technique known, it depends on reserving a bit for the sign of the number, this bit will be **0** for **positive** values and **1** for **negative** values

Assuming 8 bit variable



Signed Number representation

Sign Magnitude Technique Problems

1- Two values for 0

The number 0 has two possibilities values, positive and negative !

00000000 → +ve Zero

10000000 → -ve Zero

2- Mathematical equations not correct

If you applied this equation directly the result will be wrong, The compiler has to do extra operations to get the correct result

00000001 → +1

+

10000001 → -1

=

10000010 → -2

Signed Number representation

2- 1's Complement

This technique defines the negative number as the 1's complement for its positive value. The 1's complement is obtained by inverting all the bits in the binary representation of the number (swapping 0s for 1s and vice versa).

Let's check the previous problems.

1- Two values for 0

Still exist !

00000000 → +ve Zero

11111111 → -ve Zero

2- Mathematical equations not correct

Solved !

00000001 → +1

+

11111110 → -1

=

11111111 → 0

Signed Number representation

3- 2's Complement



This technique defines the negative number as the 2's complement for its positive value. The 2's complement is obtained by calculating the 1's complement of a value then add 1 to the result.

Let's check the pervious problems.

1- Two values for 0

Solved !

0 Has just one value !

2- Mathematical equations not correct

Solved !

$$\begin{array}{lcl} 00000000 & \longrightarrow & 0 \\ 00000000 & \longrightarrow & \text{2's complement of 0} \end{array}$$

$$\begin{array}{lcl} 00000001 & \longrightarrow & +1 \\ + & & \\ 11111111 & \longrightarrow & -1 \\ = & & \\ 00000000 & \longrightarrow & 0 \end{array}$$

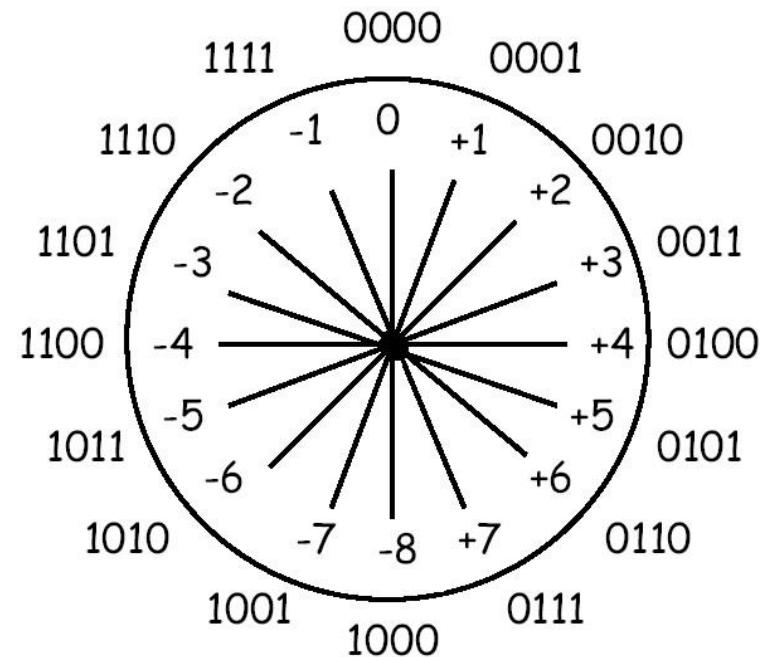
Signed Number representation

Now, the 2's complement technique is the *most common* used technique because it solved the two main problems raised by other techniques.

So, if we have a 4 bit variable which means that it can hold 16 different values. These 16 values will be divided 8 for positive values and 8 for negative values.

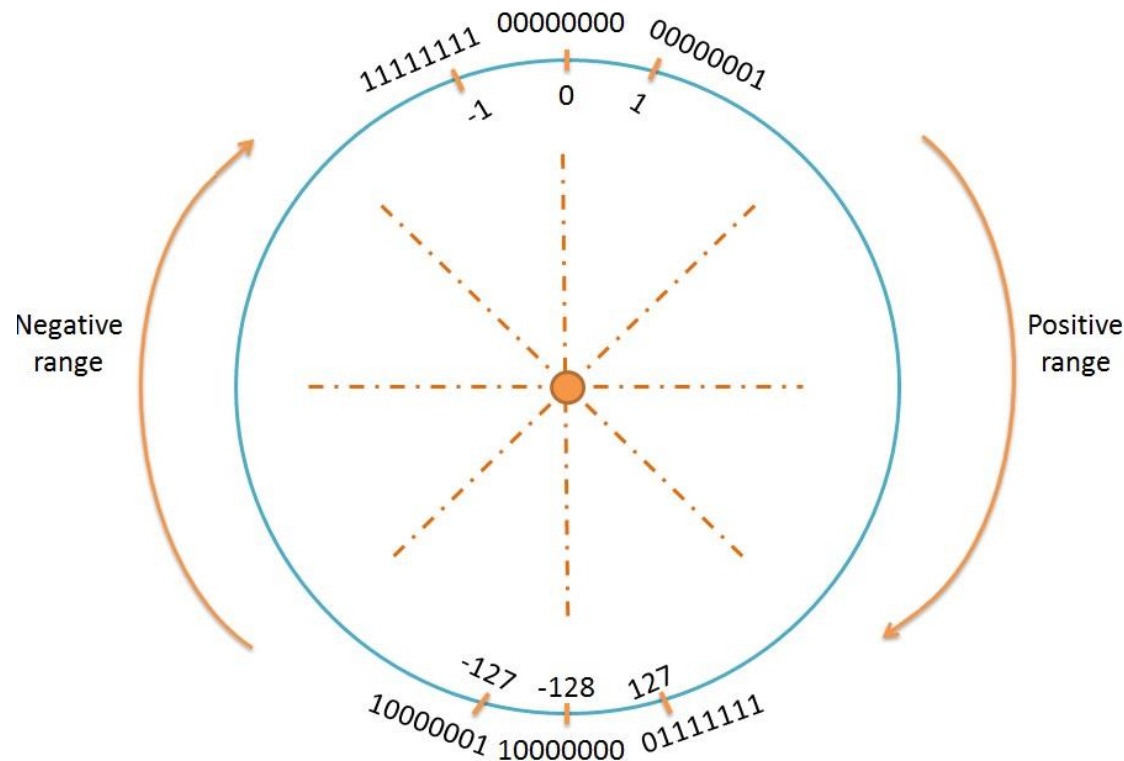
The 0 will be considered from the positive range.

So This variable will has a range from -8 to 7.



Signed Number representation

Another example for 8 bit variable, it would have a range from -128 to 127



Q: Why 8 bit variable has a range from -128 to 127 not from -127 to 128 ... ?

A: Because 0 is considered from the positive range

Sign modifier

If you want to use a variable for positive values only, then you can extend the positive range by using the sign modifier.

1- unsigned

- * Can be used only with *int* and *char* types.
- * Used to make the variable range in positive values only.

```
unsigned char x;
```



range from 0 to 255

```
signed char x;
```



range from -128 to 127

2- unsigned

- * Can be used only with *int* and *char* types.
- * by default any variable is signed, so it is redundant to write it.

Note, *float* and *double* are always signed, and you can not even write it explicitly.

```
signed float x;
```



Compilation error

Write a C code to check the type of sign representation technique in your compiler

The Code sequence is:

- 1- Define an unsigned integer variable and assign to it -1
- 2- print the variable using %u specifier (%u used to print the unsigned values)
- 3- The printed variable shall be one of the following:
 - > 0b1000000000000000000000000000000001 (Decimal 2147483649) -> **Sign Magnitude**
 - > 0b1111111111111111111111111111111110 (Decimal 4294967294) -> **1's Complement**
 - > 0b1111111111111111111111111111111111 (Decimal 4294967295) -> **2's Complement**

Time To Code



sizeof Operator

The **sizeof** is a C operator used to get the size of any **variable** or **data type**. It takes the variable or data type as an input and **returns its size in bytes**.

N.B. The sizeof is an operator not a function ! it looks like a function but it is not.

Example

```
printf("The size of int is %d", sizeof(int));
```

Get the
size of int
data type

```
float x;
```

```
printf("The size of x is %d", sizeof(x));
```

Get the size
of variable of
type float

Expected Output

Write a C code to print the size of all basic data types; char, int, float and double.

```
Size of char    is 1 bytes  
Size of int     is 4 bytes  
Size of float   is 4 bytes  
Size of double  is 8 bytes
```

Time To
Code



size modifier

This category affects the size of the data type, it has two modifiers:

1- short

This modifier comes only with int data type. *short int* is not less than 2 bytes.

2- long

This modifier comes only with int and double data types.

long int -> not less than 4 bytes

long double -> not less than 10 bytes

Examples

```
short int x ;  
  
long int y ;  
  
long double z ;
```

size modifier

Notes:

1- If you defined int variable without size modifier, the default may be *short* or *long* according to the compiler.

2- In most compilers, you can define short int like this:

```
short x ;
```

without mentioning the word *int*. Because the *short* keyword comes only with *int*.

3- Size of data types is not standard ! and may differ from environment to another.

Standard size of Basic Data types

The C standard defines a lower limit for the size of each data type. But the exact value may be different between compilers. For example, the C standard states that the *char* data type is not less than 1 byte. So you can find a compiler defines the *char* with size 1 byte and another compiler defines the *char* with size 2 bytes. Both of these compilers are following the standard.

Basic types limits in C standard

char	not less than 1 byte
short int	not less than 2 byte
long int	not less than 4 byte
float	not less than 4 byte
double	not less than 8 byte
long double	not less than 10 byte

How to make a standard types ... ?

Consider the following situation, we have two compilers, A and B

Compiler A

char is 1 byte
int is 2 bytes



Compiler B

char is 2 byte
int is 4 bytes

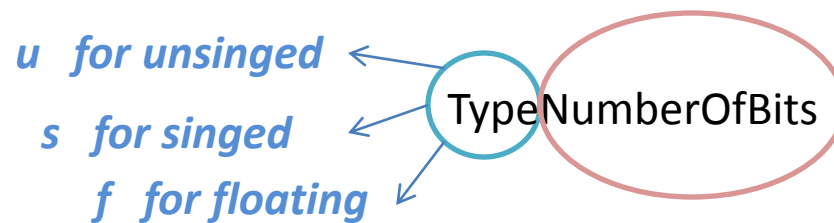
We started first with development using Compiler B, after writing a lot of code we decided to complete using Compiler A. But we faced a problem, while developing using Compiler B we used too many variables with type *char* knowing that the *char* was 2 bytes. When we decided to go to Compiler A we found that the *char* now is 1 byte, So the old code will not run correctly !

The primitive solution is to replace every *char* in the old code with *int* in the new code ! Which is too much load and may raise a lot of problems !

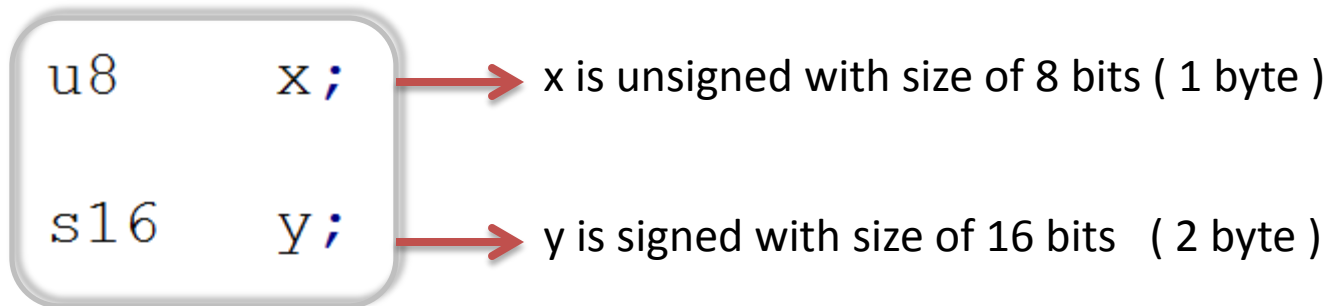
Standard Types

The best solution for this problem is to define new types that will not be changed between compilers !

The most common notation used world wide is:



Example



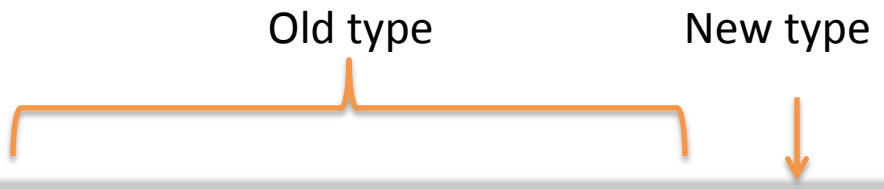
typedef keyword

This keyword used to define new types.

Syntax:

```
typedef  old_type  new_type;
```

Example



```
typedef unsigned short int u16;
```

The diagram illustrates the components of the typedef statement. An orange bracket labeled "Old type" spans the words "unsigned short int". An orange arrow labeled "New type" points to the identifier "u16".

Size Problem Solution

Back to our problem, In compiler B we used the type **u16** in our code which was equivalent to **unsigned char**. When we moved to compiler A, we just changed **u16** to be equivalent to **unsigned short int** ! and that is it, Only one line changed.

Now, we will define these types and use them in all programs we will use.

u8
u16
u32
s8
s16
s32
f32
f64

Use *typedef* keyword to define our new types. Then print their size using the sizeof operator to ensure that they are correctly defined .

Expected Output

```
The size of u8 is 1 bytes  
The size of u16 is 2 bytes  
The size of u32 is 4 bytes  
The size of s8 is 1 bytes  
The size of s16 is 2 bytes  
The size of s32 is 4 bytes  
The size of f32 is 4 bytes  
The size of f64 is 8 bytes
```

Time To Code



Milestone

Starting from now, it is **forbidden** to use the C basic types !
We will use the Our new *Standard types*



constant variable

const variable is a variable that keeps its *initial value* fixed during the execution of the whole program. Trying to change the value of a constant variable generates a compiler error

```
const u8 x = 10;
```

Note

1- Defining the variable as constant is like a promise from the developer not to change the value of this variable. So the compiler would generate a compilation error when the you try to make a change on the variable *directly*. But, we can use some *indirect* ways to change the variable !

2- Defining uninitialized constant variable is useless ! why ... ?

pointer to constant

Remember pointer syntax

Pointee_type *Pointer_Name ;

Rule

Any modifier before the * describes the pointee, Any modifier after the * describes the pointer itself

pointer to constant value



const u8* ptr

constant pointer to value



u8 * const ptr

constant pointer to constant value



const u8 * const ptr


constant pointer as input argument

```
void func1 (void)
{
    u8 x = 10;

    /* Some Code */

    func2 (&x)
}
```

Func1 sends the address of x to func2
but func2 receives this address in a
pointer to constant value



```
void func2 ( const u8* ptr )
{
    /* Here we can only read the value */
    /* of x but not changing it          */
    /* because ptr is pointing to        */
    /* Constant value                     */

}
```


constant hacker

Having a pointer to constant value, then change the constant using the pointer ... !
What will be the output ... ?

```
void func (void)
{
    const u8 x = 10;

    u8* ptr = &x;

    *ptr = 20;

}
```

Would x be
changed ... ?



Storage Modifier



auto



register



Location

RAM

CPU register or RAM

Scope

Local on a scope or
Global on the project

Local on a scope or
Global Not Applicable

Life Time

Local: Inside the scope
Global: The Whole project

Local: Inside the scope
Global: Not Applicable

static



extern



Location

RAM

No memory, Just a mirror

Scope

Local on a scope or
Global on the project

Local on a scope or
Global on the project

Life Time

The Whole project
whatever Local or Global






Local: Inside the scope
Global: The Whole project

auto Modifier

By default, any variable without a storage modifier is auto. i.e. the auto is the default storage modifier for all types of variables. So, it is **useless** to use to write the word auto while defining a new variable. Some compilers may arise **warning** when trying to write the word auto with variable definition stating that it is a useless modifier. Some other compilers give a **compilation error**. For that, we **never** use this keyword.

when **auto** comes with

	Global Variable	Local Variable
 Location	RAM	RAM
 Scope	The whole project	Between the { } where it is defined
 Life Time	Always alive	Removed by end of { } where it is defined




register Modifier

The register is a memory location *inside the processor*, which is considered the *fastest* memory in any computer system (Will be discussed later in more details). In C Language, the keyword **register** is used with a variable definition to store that variable in processor register instead of RAM. This is usually done with critical variables that need to be accessed in very fast way (like for example a variable to Enable the airbag system in a car system).

Notes:

- 1- **register** keyword is not ensured by the compiler, *i.e.* when you define a variable with register modifier, the compiler may store it in processor register or **neglect** the keyword register and consider this variable like auto variable. Each compiler defines the rules that control the acceptance criteria to store this variable in processor register or not. *i.e.* These rules are not standard.
- 2- In C, register keyword is **allowed only** with *local variables*.
- 3- register keyword **doesn't affect** the *scope* and the *lifetime* for a local variable.

Static Modifier

	Global Variable	Local Variable
 Location	RAM	RAM
 Scope	The C file where it is defined	Between the { } where it is defined
 Life Time	Always alive	Always alive

Question:

Does static keyword affects Scope or Life Time ... ?

Answer:

In Global variable, it affects the scope

In Local variable, it affects the life time

extern Modifier

extern modifier used to declare *global* variable in a file, defined in another file.

File1.c

```
u8 x = 10;
```

File2.c

```
extern u8 x;
```

Notes:

- 1- extern variable doesn't allocate a *new memory space*, it just allows file2.c to see the variable *x* which is define in file1.c
- 2- extern variable *can not be initialized* while it is being declared. It comes with the initial value declared in the original file.
- 3- extern can be used only with *global variable that are not static*.

Data Modifiers Summary

Sign	<i>signed</i>		<i>unsigned</i>	
Size	<i>short</i>		<i>long</i>	
Storage	<i>auto</i>	<i>register</i>	<i>extern</i>	<i>static</i>
Constant	<i>const</i>			
Volatility	<i>volatile</i>			

The End ...





www.imtschool.com



www.facebook.com/imaketechologyschool/

*This material is developed by IMTSchool for educational use only
All copyrights are reserved*