Introduction to Programming Language (C)

(EEF 110E)

Dr. Emre DİNCEL

LECTURE NOTES

(WEEK 5)

❖ More About C Data Types

The two main data types (integer and floating point numbers) in C have variations. Depending on the required precision and range the programmer might want to use one of these variations. The **short** modifier states that the programmer requires the data type to occupy less space in memory (which results in less accuracy or smaller range of values that can be represented by the type). For example,

```
short int i, j, k;
```

The **long** modifier states that the programmer requires the data type to occupy more space in memory (which results in higher accuracy or larger range of values that can be represented by the type). For example,

```
long int bigData;
```

Long and short modifiers are not allowed to be used with float. To have more accurate floating point numbers we use the type double. For example,

```
double x;
```

defines a high accuracy floating point number with name x. Using **long double** it might be possible to obtain even higher accuracies. The number of bytes used for representing variables in different types depends on the machine, the operating system and the compiler used.

The number of bytes used for a specific type or for a given variable can be learned by using the **sizeof()** operator. For example,

```
#include <stdio.h>
int main()
{
   long int k;
   printf("Memory size occupied by k: %d\n", sizeof(k));
   printf("Memory size occupied by char type: %d\n", sizeof(char));
   return 0;
}
```

Also, check the following table:

Data Type	Size in Bytes	Value Range	Format Specifier
(signed) char	1	-128 to 127	%c
unsigned char	1	0 to 255	%c
short (signed) int	2	-32768 to 32767	%d
short unsigned int	2	0 to 65535	%u
(signed) int	4	-2147483648 to 2147483647	%d
unsigned int	4	0 to 4294967295	%u
long (signed) int	4	-2147483648 to 2147483647	%ld
long unsigned int	4	0 to 4294967295	%lu
float	4	3.4E-38 to 3.4E+38	%f
double	8	1.7E-308 to 1.7E+308	%lf
long double	10	3.4E-4932 to 1.1E+4932	%Lf

Please, however, note that the table given above may be different depending on the processor and operating system.

- Constants & Literals:

Constants refer to fixed values that the program may not alter during its execution. The values assigned to each constant variable are referred to as the literals. Generally, both terms, constants and literals, are used interchangeably. The **const** keyword is used to define a constant variable. For example,

```
const double pi = 3.1416; // Double literal
printf("%d", 105);        // Integer literal
char ch = 'A';            // Character literal
```

Hexadecimal and octal literals (Base-16 and base-8):

A literal preceded by **0x** prefix is interpreted as a hexadecimal literal. Similarly, a literal preceded by **0** prefix is interpreted as an octal literal. Check the below examples:

```
char n = 0x61;   // n = 97;
int oVar = 010;  // oVar = 8;
```

It is possible to define the type of constant literal using specifiers.

An integer literal can have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order. For instance,

```
a = 1102L;
b = 22u;
```

It is possible to represent floating point literals either in decimal form or exponential form.

```
a = 175585E-5; // double
b = 1.75585F;  // float
c = 3.33L;     // long double
```

- Bitwise Operators:

It is possible to do bitwise operations on integers using bitwise operators. Note that bitwise operators (except the unary ~ operator) have lower priorities than arithmetic operators, but have higher priorities than logic operators. Check the below table for the priority list of the bitwise operators:

Operator	Meaning	Type	Priority
~	Complement	Unary	Highest
<< >>	Shift left / right	Binary	...
&	And	Binary	...
|	Or	Binary	...
^	XOr	Binary	Lowest

Consider the following example:

```
#include <stdio.h>
int main()
{
   int a = 0x0F;
   int b = 0x22;

   printf("a = %02x\n", a);  // a = 0b 0000 1111 (binary) = 15 (decimal)
   printf("b = %02x\n", b);  // b = 0b 0010 0010 (binary) = 34 (decimal)
   printf("a & b = %02x\n", a & b); // 0000 0010 (binary) | 02 (printf-Hex)
   printf("a | b = %02x\n", a | b); // 0010 1111 (binary) | 2F
   printf("a ^ b = %02x\n", a ^ b); // 0010 1101 (binary) | 2D
   printf("a << 1 = %02x\n", a << 1); // 0001 1110 (binary) | 1E
   printf("a >> 1 = %02x\n", a >> 1); // 0000 0111 (binary) | 07
   printf("~a = %02x\n", ~a); // 1111 ... 0000 [since 32-bit] | FFFF FFF0
```

```
    printf("a && b = %02x\n", a && b); // 01   These results are
    printf("a || b = %02x\n", a || b); // 01   due to the logic operations
    printf("!a = %02x\n", !a);         // 00   which are not bit-wise!

    return 0;
}
```

- Random Number Generation (rand & srand Functions):

Random number generation is possible in C language. Those numbers are generated via a random number generator (RNG) when the corresponding function is called.

In C language, **rand()** function can be used to generate a random number between 0 and RAND_MAX which is defined in standard library (stdlib.h) and has a typical value of 32767.

```
int rand(void);
```

However, whenever the function is called, the random number generation algorithm produces the same output. Consider the C program given below:

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

int main()
{
   int a, i;

   for (i=0; i<5; i++)
   {
      a = rand();
      printf("a = %d\t", a);
   }

   return 0;
}
```

When this program is compiled and run, you can notice that we always get the same numbers,

a = 41 a = 18467 a = 6334 a = 26500 a = 19169

Because, such algorithms (which are used for the random number generation) start from an initial point and then generate the rest of the numbers. This initial point is also called as

"seed value". If we do not (somehow) change the seed value, we will end up with the same sequence of numbers.

For the reasons stated above, we use **srand()** function to change the seed before we start to generate random numbers.

```
void srand(unsigned int seedValue);
```

Here, we can use different seed values to obtain different sequence of random numbers. However, whenever we run the program, actually we will have the same numbers again if we write a constant value in "srand" function. In order to get different random values all the time, we can use time value (in seconds) which is defined in time.h library. Since the time value is always different, we will always obtain a different random number.

Examine the same example introduced earlier:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main()
{
   int a, i;

   srand((unsigned)time(0));

   for (i=0; i<5; i++)
   {
      a = rand();
      printf("a = %d\t", a);
   }

   return 0;
}
```

If you run the above program, it is possible to see that you will end up with different numbers each time. On the other hand, the following code segment can be used to produce random numbers between "0" and "N - 1":

```
randomNumber = rand() % N;
```

Similarly, we can also use the following syntax in order to generate a random number between "a - 1" and "a - 1 + b";

```
randomNumber = rand() % a + b;
```

❖ Introduction to Pointers

So far we have been using variables to access the main memory. This is a pretty static way of reaching memory as we have to define variables when writing the program and cannot change these in run (execution) time. Also with the types of variables it is not possible to handle large amounts of data.

Sometimes it is required to pass large amounts of data between functions, or allocate memory locations dynamically. In such situations, more flexible ways of handling the memory is necessary. Pointers are used for this purpose.

- Address Operator (&):

It is possible to find the actual address of a variable using the "**&**" operator. For instance,

```c
#include <stdio.h>
int main()
{
   int k = 5;
   float m = 2.784;

   printf("Value of k = %d\n", k);    // Value of k = 5
   printf("Address of k = %p\n", &k); // Address of k = 0065FEF2
   printf("Value of m = %f\n", m);    // Value of m = 2.784
   printf("Address of m = %p\n", &m); // Address of m = 0065FF0C

   return 0;
}
```

Here, note that we have used %p to format the memory address. So, if we would like to print the content of a pointer or address of a variable, %p should be used as a format specifier.

- Pointer Variables:

A pointer is actually a memory address. It is possible to say that the pointer points to an address in memory. We can define variables that are pointers. These variables allow us to reach the main memory directly.

A pointer variable can be defined by using an asterisk in front of the variable name. The syntax is given as follows:

```c
dataType *pointerName;
```

For example,

```
int *ptrK;
```

defines a pointer variable named “ptrK” which can point integer types of data in memory. It means that we can assign the address of an integer variable to this pointer!

Dereferencing: The value of the memory location where a pointer points can be reached again by using an asterisks in front of the variable name. This is called dereferencing. Examine the below code example:

```
#include <stdio.h>
int main()
{
   int num = 5;
   int *ptrNum;  // A pointer variable which is able to store an address
                 // of an integer variable
   ptrNum = &num;
   printf("Value of num = %d\n", num);         // Value of num = 5
   printf("Addr. of num = %p\n", &num);        // Addr. of num = 0065FEC0
   printf("Value of ptrNum = %p\n", ptrNum);   // Value of ptrNum = 0065FEC0
   printf("Addr. of ptrNum = %p\n", &ptrNum); // Addr. of ptrNum = 0065FED5

   // Value referenced by ptrNum = 5
   printf("Value referenced by ptrNum = %d\n", *ptrNum);
   return 0;
}
```

 Hint: **&** → Address ***** → Content

In the above code, the variable “ptrNum” keeps the address of the variable “num”. Therefore, when we print the variable “ptrNum” on the screen, we see the address of the variable “num”. Since “ptrNum” is also a (pointer) variable, it also has an address as it is seen. Finally, in order to reach to the value stored in this memory address (content of the memory address), we have used the syntax *ptrNum.

It is also possible to change the value of the memory location pointed by a pointer by the dereference operator (*).

Examine the below code example carefully:

```c
#include <stdio.h>
int main()
{
   int num = 5;
   int *ptrNum = &num;  // We can also initialize the pointer

   *ptrNum = 1;   // Change the content of the variable pointed by ptrNum

   printf("Value of num = %d\n", num);        // Value of num = 1
   printf("Addr. of num = %p\n", &num);       // Addr. of num = 0065FEC0
   printf("Value of ptrNum = %p\n", ptrNum);  // Value of ptrNum = 0065FEC0
   printf("Addr. of ptrNum = %p\n", &ptrNum); // Addr. of ptrNum = 0065FED5

   // Value referenced by ptrNum = 1
   printf("Value referenced by ptrNum = %d\n", *ptrNum);

   return 0;
}
```

- Null Pointers:

A pointer is said to be a null pointer when it points to address 0. A null pointer is assumed to be not pointing to a valid data address. Null pointers can be used to test whether a pointer is assigned to a value. For example,

```c
int *p = 0;  // Defined a null pointer

if (p == 0)
    printf("p is a null pointer.\n");
```