Chapter: 3

Advanced data types, variables & Expression

Data type modifiers, variable declaration & initialization, type conversion.

3.1 *DATA - TYPE MODIFIERS*

C has five basic data types: ***void***, ***char***, ***int***, ***float***, and ***double***. These basic types, except type void, can be modified using C's type modifiers to more precisely fit our specific need. The type modifiers are

long ***short*** ***signed*** ***unsigned***

The type modifier precedes the type name. For example, this declares a long integer: `long int i;`

long and short modifiers : The ***long*** and ***short*** modifiers may be applied to ***int***. As a general rule, ***short ints*** are often smaller than ***ints*** and ***long ints*** are often larger than ***ints***. For example, in most 16-bit environments, an ***int*** is 16 bits long and a ***long int*** is 32 bits in length. In the ANSI C standard, the smallest acceptable size for an ***int*** is 16 bits and the smallest acceptable size for a ***short int*** is also 16 bits. In most 16 bit environments, there is no difference between an ***int*** and a ***short int***. Further, in many 32-bit integers and long integers has the same size. Exact effect of long and short on integers is determined by the environment and by the compiler.
The ***long*** modifier may-also be applied to ***double***. Doing so roughly doubles the precision of a floating point variable.

The signed modifier : The ***signed*** modifier is used to specify a signed integer value. (A signed number means that it can be positive or negative.) Default integer declaration automatically creates a signed variable so there is no need of signed modifier. The main use of the ***signed*** modifier is with ***char***. Some case char is signed or unsigned. To ensure a signed character variable in all environments, we must declare it as ***signed*** char.

The unsigned modifier : The unsigned modifier can be applied to char and int. It may also be used in combination with long or short. It is used to create an unsigned integer.

The difference between signed and unsigned integers is in the way the high-order bit of the integer is interpreted. Signed integers are important for a great many algorithms, but they only have half the absolute magnitude of their unsigned relatives . For example, here is 32,767 shown in binary:

```
01111111      11111111
```

If this is a signed value and the high-order bit is set to 1, the number would then be interpreted as -1 (assuming two's complement format). However, if this is an unsigned value, then when the high-order bit is set to 1, the number becomes 65,535.

Type	***Typical size in bits***	***Minimal range***
`char`	8 or 1 byte	-127 to 127 **(related to 2^8)/2**
`unsigned char`	8 or 1 byte	0 to 255 **(related to 2^8)**
`signed char`	8 or 1 byte	-127 to 127 **(related to 2^8)/2**
`int`	16 or 32 (2 byte or 4 byte)	-32,767 to 32,767 **(related to 2^16)/2**
`unsigned int`	16 or 32 (2 byte or 4 byte)	0 to 65,535 **(related to 2^16)**
`signed int`	16 or 32 (2 byte or 4 byte)	Same as int
`short int`	16 (2 byte)	Same as int
`unsigned short int`	16 (2 byte)	0 to 65,535 **(related to 2^16)**
`signed short int`	16 (2 byte)	Same as short int i.e. int
`long int`	32 (4 byte)	-2,147,483,647 to 2,147,483,647 **(2^32)/2**
`signed long int`	32 (4 byte)	Same as long int
`Unsigned long int`	32 (4 byte)	0 to 4,294,967,295 **(2^32)**
`float`	32 (4 byte)	Six digit of precision
`double`	64 (8 byte)	Ten digit of precision
`long double`	80 (10 byte)	Ten digit of precision

bit and byte: 1 and 0 are bit's . And 1 byte = 8 bit. In 1 byte or 8 bit we can have **00000000** or **11111111** or any combination of 1's and 0's out of total 256 (=2^8) combinations. In the case of signed data type 1 bit is used for signing. For example 2 takes space in ***int*** type data

1 bit	***15 bits***						
0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0 (=2)

NOTES :

1. C allows a shorthand notation for declaring ***unsigned***, ***short***, or ***long*** integers. We can simply use the word ***unsigned***, ***short***, or ***long*** without the ***int***. The ***int*** is implied. For example, both declare unsigned ***int*** variables.
```
unsigned count;
unsigned int num;
```

2. It is important to remember that variables of type char may be used to hold values other than just the ASCII character set. C makes little distinction between a character and an integer, except for the magnitudes of the values each may hold. A signed char variable can also be used as a 'small' integer when the situation does not require larger numbers.

3. When outputting integers modified by ***short***, ***long***, or ***unsigned*** using ***printf( ),*** you cannot simply use the ***%d*** specifier. The reason is that ***printf( )*** needs to know precisely what type of data it is receiving. To use ***printf( )*** to output a ***short***, use ***%hd.*** To output a ***long***, use ***%ld***. When outputting an ***unsigned*** value, use ***%u***. To output an ***unsigned long int***, use ***%lu***. Also, to ***output*** a ***long double*** use ***%Lf*** (upper case L ?).
4. The ***scanf( )*** function operates in a fashion similar to ***printf( )***. When reading a ***short int*** using ***scanf( )***, use ***%hd***. When reading a ***long int***, use ***%ld***. To read an ***unsigned long int***, use ***%lu***. To read a ***double***, use ***%lf***. To read a ***long double***, use ***%Lf***.
5. In C, you may use a ***char*** variable any place you would use an ***int*** variable. For example, we can use a ***char*** variable to control the loop that is summing the numbers between 1 and 100.

3.2 Advanced variable declaration (Local and Global)

There are two basic places where a variable will be declared: inside a function and outside all functions. These variables are called **local variables** and **global variables**, respectively.

Local variables : Local variables (declared Inside a function) may be referenced only by statements that are inside that function. They are not known outside their own function. One of the most important things to understand about local variables is that they exist only while the function in which they are declared is executing. That is, **a local variable is created upon entry into its function and destroyed upon exit**. So it is perfectly acceptable for **local variables in different functions to have the same name**.

```
        #include<stdio.h>
        void f1(void), f2(void);
        int main(void){     f1();  return 0; }
        void f1(void){ int count;
                  for(count=1; count<11; count++) {printf("\n count : %d \n",count); f2();}
                  }
        void f2(void){      char count;
                            printf("Alphabets :");
                            for(count='a'; count<='z'; count++) printf(" %c",count);
                  }
```

1. The C language contains the keyword ***auto***, which can be used to declare local variables. However, since all local variables are, by default, assumed to be ***auto***, it is virtually never used.
2. Within a function, local variables can be declared at the start of any block. They do not need to be declared only at the start of the block that defines the function. For example, the following program is perfectly valid:
```
    #include<stdio.h>
    int main(void){ int i;
        for(i=0; i<10; i++){   if(i==5){   int j; /*Declare j within the if block*/
              j=i*10;
              printf("%d", j);} }  return 0;}
```
   A variable declared within a block is known only to other code within that block. Thus, j may not be used outside of its block. **Most C programmers declare all variables used by a function at the start of the function's block** because it is simply more convenient to do so.
3. Remember one important point: **You must declare all local variables at the start of the block in which they are defined, prior to any program statements.** For example, the following is incorrect:
```
    #include <stdio.h>
    int main(void) {    printf("This program won't compile.");
                        int i;       /* this should come first */
                        i = 10; printf("%d", i); return 0;}
```
4. **Remember. local variables do not maintain their values between functions calls**. When a function is called, its local variables are created, and upon its return, they are destroyed. This means that local variables cannot retain their values between calls.
5. The formal parameters to a function are also local variables. Even though these variables perform the special task of receiving the value of the arguments passed to the function, they can be used like any other local variable within that function .

Global variables : Global variables are known throughout the entire program and may be used by any piece of code in the program. Also, they will hold their value during the entire execution of the program. Global variables are created by declaring them outside any function. For example, consider this program:

```
            #include <stdio.h>
            void f1 (void) ;
            int max; /* this is a global variable */
            int main (void) {   max = 10;    f1() ;       return 0; }
            void f1 (void) {    int i; for(i=0; i<max; i++) printf("%d",  i); }
```

Here, both ***main()*** and **`f1( )`** use the global variable ***max***. The ***main( )*** function sets the value of ***max*** to ***10***, and **`f1( )`** uses this value to control its ***for*** loop.

1. In C, a local variable and a global variable may have the same name. When ***local*** and ***global*** variables share the same name, the compiler will always use the local variable.For example :
```
            #include <stdio.h>
            void f1 (void) ;
            int max;                         /* this is a global variable */
            int main (void) {   max=10; printf("%", max); . . . statements  }
            void f1 (void) {    int max;      /* this is a local variable */
                                max=300; printf("%", max); . . . . statements }
```

In ***main( ),*** the reference to ***max*** is to the **global variable**. Inside ***f1( ),*** a **local variable** called ***max*** is also defined. When the assignment statement inside ***f1( )*** is encountered, the compiler first looks to see if there is a local variable called ***max***. Since there is, the local variable is used, not the global one with the same name. That is, **when local and global variables share the same name, the compiler will always use the local variable**.

2. Global variables are very helpful when the same data is used by many functions in your program. However, you should always use local variables where you can because the excessive use of global variables has some negative consequences.
   - First, global variables use memory the entire time your program is executing, not just when they are needed. In situations where memory is in short supply, this could be a problem.
   - Second, using a global where a **local variable will do a perfect job** makes a function **less general**, because it relies on something that must be defined outside itself.

Here, the function ***power( )*** is created to compute the value of ***m*** raised to the ***e***'th power. Since ***m*** and ***e*** are ***global***, the function cannot be used to compute the power of other values. It can only operate on those contained within **m** and **e**. In this case any program that uses ***power( )*** must always declare ***m*** and ***e*** as ***global variables*** and then load them with the desired values each time ***power( )*** is used.

```
/*Non-general form of a function*/
       #include<stdio.h>
       int power(void);
       int m, e;
       int main(void){m=2; e=3;
                     printf("%d to the power %d is %d^%d = %d", m,e,m,e, power());
                     return 0;}

/*Non-general version of power(): this function can not be used or called for any other values because m,e are global and fixed in main() here power() lost its general purpose */
       int power(void) {    int temp_1, temp_2;
                            temp_1=1;
                     /*Using e which is global and has a fixed value defined in main()*/
                            temp_2=e;
                     /*Using m which is global and has a fixed value defined in main()*/
                            for( ; temp_2>0; temp_2--) temp_1=temp_1*m;
                            return temp_1;}

/*general form of a function*/
#include<stdio.h>
int power(int m, int e);
int main(void){   int m, e;
do{printf("\nEg : %d to the power %d is %d^%d = %d", 2,3,2,3, power(2,3));
  printf("\nm to the power e is m^e = ? Enter m and e (m=0 to  exit):"); scanf("%d %d", &m, &e);
  printf("\n%d to the power %d is %d^%d = %d", m,e,m,e, power(m,e));} while (m);
  return 0;}

/*general version of power(): this function has parameterized local variables */
       int power(int m, int e){   int temp;
                                  temp=1;
                                  for( ; e>0; e--) temp=temp*m;
                                  return temp;}
```

By parameterizing ***power( )***, can be used to return the result of any value raised to some power. Here the function is complete within itself-no extra baggage need be carried about when it is used.

3. **Using a large number of global variables can lead to program errors** because of unknown and unwanted side effects. A major problem in developing large programs is the accidental modification of a variable's value because it was used elsewhere in the program. This can happen in C if you use too many global variables in your programs.

3.3 Constants Advanced

1) C also allows us to use scientific notation for floating-point numbers. Constants using scientific notation must follow this general form:

   number **E** sign **exponent**

   The ***sign*** is optional. Although the general form is shown with spaces between the component parts for clarity, there may be no spaces between the parts in an actual number. For example, the following defines the value ***1234.56*** using scientific notation: ***123.456E1***
2) These two assignment statements are equivalent. `char ch ;      ch = 'A ' ;   ch = 65 ;`
3) The compiler must decide what type of constant are used in the program. For example, is 1000 an ***int***, an ***unsigned***, or a ***short***? The reason is that C automatically converts the type of the right side of an assignment statement to that of the variable on the left. So, for many situations it doesn't matter what the compiler thinks 1000 is.
4) However, this can be important when you use a constant as an **argument to a function**, such as in a call to ***printf()***. By default, the C compiler fits a numeric constant into the **smallest compatible data type** that will hold it. Assuming 16-bit integers, *10* is an ***int*** by default and *100003* is a ***long***. Even though the value 10 could be fit into a ***char***, the compiler will not do this because it means **crossing type boundaries**. The only exceptions to the smallest-type rule are **floating-point** constants, which are assumed to be **doubles**.

5) C allows you to specify the exact type by using a suffix. For **floating-point types**, if you follow the number with an '**F**', the number is treated as a **float**. If you follow it with an '**L**', the number becomes a **long double**. For integer types, the '**U**' suffix stands for **unsigned** and the '**L**' stands for **long**.

6) A hexadecimal constant must begin with '`0x`' (a zero followed by an x) then the constant in hexadecimal form. An octal constant begins with a zero. For example, **`0xAB`** is a hexadecimal constant, and **`024`** is an octal constant. You may use either upper- or lowercase letters when entering hexadecimal constants.

7) If we use ***short int (%hd)*** data type to printing the number **`42340`**, it displays **`-23196,`** because it thinks that it is receiving a ***signed short integer***. The problem is that **`42,340`** is outside the **range of a short int**. To make it work properly, we must use the ***%hu*** specifier .

8) **`printf("%f", 2309);`** is telling ***printf( )*** to expect a floating point value, but the compiler assumes that 2309 is simply an ***int***. Hence, it does not output the correct value. To fix it, we must specify 2309 as 2309.0. Adding the decimal point forces the compiler to treat the value as a ***double*** .

The string : A string is a set of characters enclosed by double quotes. We used string in ***printf( )*** and ***scanf( )*** functions.
Keep in mind one important fact : although C allows you to define string constants, it does not formally have a string data type. Strings are supported in C as character arrays. To display a string using ***printf( )*** you can either make it part of the control string or pass it as a separate argument and display it using the ***%s*** format code. For example, this program prints **Once upon a time** on the screen:

```
printf("%s %s %s", "Once", "upon", "a time");
```

Here, each string is passed to ***printf( )*** as an argument and displayed using the ***%s*** specifier.

3.4 Variable initialization

A variable may be given an initial value when it is declared. This is called ***variable initialization***. The compiler may be able to produce faster code using **initialized variable**. The general form of variable initialization is shown here:

```
type var-name = constant;
```

For example, `int count = 100;` declares **count** as an ***int*** and gives it an initial value of 100.

NOTES :

1. **Global variables** are initialized only once, at the start of program execution. Global variables may be initialized using only constants. Global variables that are not explicitly initialized are automatically set to zero.
2. **Local variables** are initialized each time a function is entered. Local variables can be initialized using constants, variables, or function calls as long as each is valid at the time of the initialization. Local variables that are not initialized should be assumed to contain unknown values. Although some C compilers automatically initialize un-initialized local variables to 0.
3. When you declare a list of variables, you may initialize one or more of them. For example, this fragment initializes ***`min`*** to ***`0`*** and ***`max`*** to ***`100`***. It does not initialize **`count`**.
   ```
   int min=0, count, max=100;
   ```
4. A local variable can be initialized by any expression valid at the time the variable is declared. For example, initializ-e a local variable ***y*** using a global variable ***x***, `int y = x;` and initialize a local variable ***z*** using another local variable ***y*** and a function ***myfunc()***
   ```
   int z = myfunc(y);
   ```
5. You cannot initialize a global variable using another variable. `int a=1, b=2, c=a;` is wrong in this case. However a local variable can be initialized using any expression valid at the time of the initialization. . `int a=1, b=2, c=a;` is correct in this case

3.5 Type conversion in "Expression" and "Assignment"

Type conversion in "Expression" : C lets us mix different types of data together in one expression. For example, this is perfectly valid :

```
char ch;                                      ch = '0';
int i;                                        i = 10;
float f;                                      f = 10.2;
double outcome;                               outcome = ch * i / f;
```

C allows the mixing of types within an expression because it has a strict set of conversion rules that dictate how type differences are resolved.

Conversion using integral promotion : One portion of C's conversion rules is called ***integral promotion***. Integral promotion is only in effect during the evaluation of an expression. In C, whenever a ***char*** or a ***short int*** is used in an expression, its value is automatically elevated to ***int*** during the evaluation of that expression. This is why you can use ***char*** variables as ***'little integers'*** anywhere , you can use an int variable.

type promotion according to type-conversion algorithm : After the automatic integral promotions have been applied, the C compiler will convert all operands "***up***" to the type of the largest operand. This is called ***type promotion*** and is done on an ***operation-by-operation*** basis, as described in the following **type-conversion algorithm**.

```
IF an operand is a long double → THEN the second is converted to long double
ELSE IF an operand is a double → THEN the second is converted to double
ELSE IF an operand is a float → THEN the second is converted to float
ELSE IF an operand is an unsigned long → THEN the second is converted to unsigned long
ELSE IF an operand is long → THEN the second is converted to long
ELSE IF an operand is unsigned → THEN the second is converted to unsigned
```

<u>There is one additional special case:</u> If one operand is ***long*** and the other is ***unsigned int***, and if the value of the ***unsigned int*** cannot be represented by a ***long***, both operands are converted to ***unsigned long***. Once these conversion rules have been applied, each pair of operands will be of the same type and the result of each operation will be the same as the type of both operands. For example :

For **int** `i;` **float** `f;` ***i*** is elevated to a ***float*** during the evaluation of the `i*f` .

Even though the final outcome of an expression will be of the largest type, the type conversion rules are applied on an operation-by-operation basis. For example., in `100.0/(10/3)` the division of `10` by `3` produces an integer result, since both are integers. Then this value is elevated to `3.0` to divide `100.0`.

Type conversion in "Assignment" : In an assignment statement in which the type of the right side differs from that of the left, the type of the right side is converted into that of the left.

- When the type of the left side is larger than the type of the right side, this process causes no problems.
- However, when the type of the left side is smaller than the type of the right, data loss may occur.

For example, this program displays -24:

```
char ch; int i; i = 1000; ch = i;
```

The reason for this is that only the low-order eight bits of ***i*** are copied into ***ch*** . Since this sort of assignment type conversion is not an error in C, no error message will be received.

When there is an **integer-to-character** or a **longer-integer to shorter-integer** type conversion across an assignment, the basic rule is that the appropriate number of **high-order bits will be removed**.

- In many environments, ***8 bits*** will be lost when going from an ***int*** to a ***char***, and ***16 bits*** will be lost when going from a ***long*** to an ***int***.
- When convel1ing from a ***long double*** to a ***double*** or from a ***double*** to a ***float***, precision is lost.
- When converting from a **floating-point** value to an **integer value**, the fractional part is lost, and if the number is too large to fit in the target type, a garbage value will result.
- The conversion of an ***int*** to a ***float*** or a ***float*** to ***double***, and so on, will not add any precision or accuracy. These kinds of conversions will only change the form in which the value is represented.
- Some C compilers will always treat a ***char*** variable as an ***unsigned*** value. Others will treat it as a ***signed*** value. Thus, what will happen when a character variable holds a value greater than 127 is implementation-dependent.
- It is best to declare the variable explicitly as either ***signed*** or ***unsigned***.

3.6 The TYPE CASTS

To transform the type of a variable temporarily. For example, to use a **floating-point** value for one computation, but wish to apply the **modulus operator** to it elsewhere. Since the **modulus operator** can only be used on **integer** values, we have a problem. One solution is to create an **integer** variable for use in the **modulus operation** and **assign** the value of the **floating-point variable** to it when the time comes. This is a somewhat ***inelegant solution***, however. The other way around this problem is to use ***a type cast, which causes a temporary type change***. A type cast takes this general form:

```
(type) value .
```

where type is the name of a valid C data type. For example,

```
float f;     f = 100.2;
      /*print f as an integer */
printf("%d", (int) f);
```

- The single argument of ***sqrt( )*** must be of type ***double***. It also returns a ***double*** value.
- You cannot cast a variable that is on the ***left side of an assignment statement***. For example: this is an ***invalid*** statement in C:

```
int num;    (float) num = 123.23;     / * this is incorrect * /
```