The Use of #define, printf() and scanf()

We have mentioned earlier that C compiler has a preprocessor built into it. We have discussed `#include<stdio.h>` in that context. Any line that begins with `#` is called a preprocessor directive. If the line `#define RATE 0.05` occurs in a C source file that is being compiled, the preprocessor first changes all occurrences of `RATE` to `0.05` *except in quoted strings and comments*. A `#define` line can occur anywhere in the program and it affects only the lines that come after it. Normally, `#define` statement(s) appear at the beginning of a C source file. Also, by convention, the identifiers (symbolic constant) that are to be changed by the preprocessor are written in capitals (such is `RATE`).

To see the utility of `#define` directives, consider the program "simpint.c" discussed in the previous lecture. We have simple interest rate 5% which we have assigned to the identifier `rate` in the middle of the source file. If the simple interest changes to 6%, then we have to change "`rate=0.05;`" to "`rate=0.06;`". On the other hand, suppose that we copy the source file "simpint.c" to a new file "msimpint.c" and edit the newly created file as shown below:

```
/* Calculate total amount accrued in 5 years when rate
 of interest is 5% and principal amount is Rs. 100000.00*/
#include<stdio.h>
#define RATE 0.05
int main()
{
        int years;
        double princp,total;

        years=5;                //no of years
        princp=100000.00;       //principal amount

        total=princp*(1+RATE*years);

        printf("Total amount accrued in %d years is Rs. %lf\n",years,total);
        return 0;
}
```

Listing 6: C program "msimpint.c"

Compiling with `$ gcc msimpint.c ↩` and running the default executable with `$ ./a.out ↩` produces the following output as before:

```
Total amount accrued in 5 years is Rs. 125000.000000
```

Analysis of "msimpint.c" program

- `#define RATE 0.05`

  This is a `#define` directive in which `RATE` is a symbolic constant. Note that there is no semicolon ";" at the end of this line.

- `total=princp*(1+RATE*years);`

  The symbolic constant `RATE` is used here. In the preprocessing stage, this statement is replaced with `total=princp*(1+0.05*years);`

If we want to calculate with simple interest of 6%, we only replace `#define RATE 0.05` with `#define RATE 0.06`, i.e., change "0.05" to "0.06". Also, note that we don't need real identifier `rate` and the corresponding assignment statement in the program.

Use of parenthesis in `#define`

Sometimes we need to use parenthesis in `#define` directive to obtain correct result. To illustrate this, suppose that we want to calculate x^2 for a real number x. We know that $x^2 = x * x$ and this can be included exactly in a code. However, suppose that we implement this using `#define` directive. We define two directives for the same purpose and see that only one of them works in all situations. We create a new file "csq.c" as shown below.

```
//Illustrating correct/incorrect way to calculate x^2
#include<stdio.h>
#define SQ(x) (x)*(x)
#define WSQ(x) x*x
int main()
{
   double x=2.0;
   printf("Using SQ %lf  Using WSQ %lf\n",SQ(x),WSQ(x));//both give correct result
   printf("Using SQ %lf  Using WSQ %lf\n",SQ(x+1),WSQ(x+1));//1st gives correct result
   return 0;
}
```

Listing 7: C program "csq.c"

Compiling with `$ gcc csq.c ↩` and running the default executable with `$ ./a.out ↩` produces the following output:

```
Using SQ 4.000000   Using WSQ 4.000000
Using SQ 9.000000   Using WSQ 5.000000
```

Clearly, the `#define` directive with symbolic constant `WSQ(x)` does not work correctly in the calculation of $(x+1)^2$.

Analysis of "csq.c" program

- `printf("Using SQ %lf  Using WSQ %lf\n",SQ(x),WSQ(x));//both give ......`

  In the preprocessing stage, the statement is replaced with
  `printf("Using SQ %lf  Using WSQ %lf\n",(x)*(x),x*x);`
  Clearly, with $x = 2.0$, both give 4.0 as expected.

- `printf("Using SQ %lf  Using WSQ %lf\n",SQ(x+1),WSQ(x+1));//1st gives ......`

  Here, in the preprocessing stage, the statement is replaced with
  `printf("Using SQ %lf  Using WSQ %lf\n",(x+1)*(x+1),x+1*x+1);`
  Clearly, with $x = 2.0$, `SQ(x+1)` gives $(2.0 + 1) * (2.0 + 1) = 9.0$, which is correct. Since "*" has higher precedence, the expression `x+1*x+1` is equivalent to $2x + 1$ which gives $2 \times 2.0 + 1 = 5.0$. Thus, `WSQ(x)` may not work correctly in all cases.

Note: The value of π is used in many C programs such as finding the area of circle, converting radian to degree, etc. Hence, the value of π can be defined in terms of `#define` directive. Because of its widespread uses, it has already been defined in a C system using symbolic constant `M_PI`. However, it is defined in `math.h` header file as shown below (in my Linux system).

```
# define M_PI  3.14159265358979323846 /* pi */
```

Hence, to use it in your program, you need to include `math.h` header file using `#include<math.h>` and write `M_PI` in place of π. Alternately, you may copy and paste the above line above "main()" in your source file.

Use of printf() and scanf(): This function "printf()" is used for output and the function "scanf()" is used for input . We have already used "printf()" in our previous C programs such as in "hello.c". The 'f' in "printf()" and "scanf()" stands for formatted. The arguments of both "printf()" and "scanf()" consist of a *control string* and *other arguments*. The first argument is the *control string* which may contain conversion specifications or formats. A conversion specification starts with a % character and ends with a conversion character. For example, in the format %d used in "simpint.c", 'd' is the conversion character and %d is used to print a decimal integer. To print `Hello`, we use

```
printf("Hello");
```

where we have not use new line character `'\n'`. Also, we have "Hello" as the control string which is the only argument. On the other hand, the same output can be observed using

```
printf("%s","Hello");
```

where the control string `"%s"` (first argument) contains the format `%s` which causes the string `"Hello"` (second argument) to be printed on the screen. In addition, the same output can be obtained by

```
printf("%c%c%c%c%c",'H','e','l','l','o');
```

where the "printf()" has six arguments. The first arguemt is the control string `"%c%c%c%c%c"` which has formats `%c` which is used to print a character expression. The other arguments like `'H'`, `'e'`, etc. are character constant. For example, the sixth argument `'o'` corresponds to last `%c` in the control string.

A number in scientific notation is a number between 1 and 10 multiplied by a power of 10. To convert a number to scientific notation, move the decimal point such that there is only one nonzero digit to its left, then multiply by 10 to the power of the number of places you moved the decimal. For example, 234.54 becomes 2.3454×10^2 and 0.00235 becomes 2.35×10^{-3} in scientific notation. Below is a table showing conversion characters that are widely used in "printf()" function in C programs.

printf() conversion characters	
Conversion character	How the corresponding argument is printed
`c`	as a character
`d`	as a decimal integer
`f`	as a real number with decimal point (for both `float` and `double`)
`lf`	as a `double` real number with decimal point
`e`	as a real number in scientific notation (for both `float` and `double`)
`g`	as a real number with `e`-format or `g` format whichever is shorter.
`s`	as a string

```
#include<stdio.h>
int main()
{  char ch='A';
   float b=123.8;
   double c=123.8;
   double d=1.23456789e4; // = 1.23456789 x 10^4
   double e=1.23456789e8;

   printf("Illustating printf()\n"); // only control string
   printf("%c for Apple\n",ch); // printing a character using %c
   printf("b=%f  and  c=%f\n",b,c); // printing float and double using %f
   printf("c=%lf\n",c); // printing double using %lf
   printf("b=%e  and  d=%e\n",b,d); // printing using %e
   printf("d=%g  and  e=%g\n",d,e); // printing using %g
   return 0;
}
```

Listing 8: C program "cprin.c"

Compiling with `$ gcc cprin.c ↩` and running the default executable with `$ ./a.out ↩` produce the following output:

```
Illustating printf()
```

```
A for Apple
b=123.800003  and  c=123.800000
c=123.800000
b=1.238000e+02  and  d=1.234568e+04
d=12345.7  and  e=1.23457e+08
```

Analysis of the output of "cprin.c" program

- ☐ `b=123.800003  and  c=123.800000`

  By default, `%f` prints six-places after decimal with rounding. Also, note that `b` is not the actual value assigned. The reason for this is that any real decimal number is converted in binary format and most of the real numbers cannot be represented exactly using finite number of binary digits fixed for a given data type. Since `float` is less accurate (since it uses less number of binary digits) than `double`, we observe the discrepancy in the output. Note that `%f` and `%lf` are effectively the same in "printf()" because "printf()" expects double and float arguments are promoted to double.

- ☐ `b=1.238000e+02  and  d=1.234568e+04`

  By default, six places after decimal with rounding is printed. In the output for `d`, '7' is rounded to '8'.

- ☐ `c=12345.7  and  d=1.23457e+08`

  By default, six *significant* digits (with trailing zeros removed) are printed. Here, `c` is printed using `%f` and `d` is printed using `%e` since these are shorter to express these numbers.

Remark about formats in printf(): Between the `%` that starts the format and conversion character that ends it, there may appear in order (these will be discussed with examples later)

- ☐ zero or more flag characters that modify the meaning of conversion specification
- ☐ an optional positive integer that specifies the minimum field width of the converted argument
- ☐ an optional precision, which is is indicated by a period (i.e., `.`) followed by a nonnegative integer
- ☐ an optional `h` or `l` which is a "short" or "long" modifier respectively.
- ☐ an optional `L` which is a "long" modifier.

Since % is used for formatting, to print % in a string, we need to use %%. Thus, the statement "`printf("She was warded 70%% marks\n");`" produces output `She was warded 70% marks`

Escape character: The backslash character `\` is called the escape character. It is used to escape the usual meaning of the character that follows it. Here are few special characters (refer

to a book for others) and the way they are written in C.

Name of character	Written in C
alert	`\a`
backslash	`\\`
double quote	`\"`
horizontal tab	`\t`
newline	`\n`
vertical tab	`\v`

For example, the statement "`printf("Change\nA gift of life\n");`" produces the output

```
Change
A gift of life
```

On the other hand, the statement "`printf("Change\vA gift of life\n");`" produces

```
Change
      A gift of life
```

The function "scanf()" is similar to "printf()" but it is used for input. Its first argument is a control string having formats that correspond to various ways the characters in the input stream are to be interpreted. The other arguments are *addresses*. For example, in the statement

```
scanf("%d",&a);
```

the format `%d` is matched with the expression `&a` and "scanf()" interprets characters in the input stream as decimal integer and stores the result at the address of `a`. The expression `&a` is read as ""address of `a`" since `&` is the address operator.

When the input is entered via keyboard, a sequence of characters is typed and this sequence of characters is called the input stream that is received by the program. If the programmer types `123`, then the program receives it as a sequence of characters. A "scanf()" function converts this string of decimal digits into an integer value and store the value at an appropriate place (address) in memory.

scanf() conversion	
Conversion character	How characters in the input stream are converted
`c`	character
`d`	decimal integer
`f`	floating-point number (`float`)
`lf` or `LF`	floating-point number (`double`)
`s`	string

When the execution of a program hits a "scanf()" statement, it waits there for the input(s) from the user. It is always preferable to print about the input(s) before the "scanf()" statement so that the user knows about the input(s). The input for floating-point number can be given either in terms of decimal representation or scientific notation. Below is a program where we read three characters, an integer, and two floating point numbers (a float and a double).

```
#include<stdio.h>
int main()
{
 char ch1,ch2,ch3;
 int a;
 float b;
 double c;

 printf("Enter three characters:");
 scanf("%c%c%c",&ch1,&ch2,&ch3);
 printf("Enter an integer and a floating no:");
 scanf("%d%f",&a,&b);
 printf("Enter a double no:");
 scanf("%lf",&c);

 printf("Three characters entered are %c, %c, %c\n",ch1,ch2,ch3);
 printf("The integer entered is  %d\n",a);
 printf("The float entered is  %f\n",b);
 printf("The double entered is  %f\n",c);
 return 0;
}
```

Listing 9: C program "cscn.c"

Compiling with $ gcc csn.c ↩ and running the default executable with $ ./a.out ↩ produce the following input/output:

```
Enter three characters: ApS↩
Enter an integer and a floating no: 23 3.4e-3↩
Enter a double no: .0034↩
Three characters entered are A, p, S
The integer entered is 23
The float entered is  0.003400
The double entered is  0.003400
```

The shaded parts with ↩ are the inputs by the programmer.

Analysis of the input/output of "cscn.c" program

- ☐ Note that there is no space between formats in the control string (e.g, `"%c%c%c"` or `"%d%f"`).
- ☐ There is no newline character (i.e. `'\n'`) in the "scanf()" function.

- ☐ There is space between `int` and `float` inputs, i.e., between `23` and `3.4e-3`.
- ☐ There is no space between character inputs, i.e., `ApS`. The reason is that a blank space is also character.

Caution about reading characters in C: Consider the very simple code given below in which first a character is read followed by an integer. Then the input values are printed.

```
#include<stdio.h>
int main()
{
   char ch;
   int a;
   printf("Enter a character:");
   scanf("%c",&ch);
   printf("Enter an integer:");
   scanf("%d",&a);
   printf("Entered character is %c\n",ch);
   printf("Entered integer is %d\n",a);
   return 0;
}
```

Compiling and running the executable produce the following input/output which is self-explanatory.

```
Enter a character: A↩
Enter an integer:23 23↩
Entered character is A
Entered integer is 23
```

Now consider the same program but the integer is read first followed by the character.

```
#include<stdio.h>
int main()
{
 char ch;
 int a;

 printf("Enter an integer:");
 scanf("%d",&a);
 printf("Enter a character:");
 scanf("%c",&ch);
 printf("Entered character is %c\n",ch);
```

```
 printf("Entered integer is %d\n",a);
 return 0;
}
```

Compiling and running the executable produce the following input/output:

```
Enter an integer: 23↩
```

Enter a character:Entered character is

```
Entered integer is 23
```

It is seen that it accepts the integer input. Then it prints "Enter a character:" but does not wait for the input. Instead, it prints "Entered character is" followed by blank line and then the output "Entered integer is 23". The reason for this strange behaviour is the following. When we type input `23` followed by pressing the return/enter key, we also enter a newline character (i.e, `'\n'`). This newline character is taken as input for the character. Hence, the code does not wait for the character input. Instead it prints the output of the statement "`printf("Entered character is %c\n",ch);`", where is `ch` is now `'\n'`. Hence, the new blank line appears followed by the output of the next statement. To remedy this problem, there are few strategies available. One of the strategy is to keep a blank space before `%c`, i.e., replace "`scanf("%c",&ch);`" with "`scanf(" %c",&ch);`".

Return to Lecture Topics