BIL104E: Introduction to Scientific and Engineering Computing

Lecture 7

- Manipulating Strings
- Scope and Storage Classes in C

Strings

- Declaring a string
- The length of a string
- Copying strings
- Reading strings with scanf(
- The gets() and puts() functions

What Is a String?

As introduced in Lecture 6, "Storing Similar Data Items," a string is a character array terminated by a null character (\0).

For instance, a character array, array_ch, declared in the following statement, is considered a character string:

```
char array_ch[7] = {`H', `e', `l', `l', `o', `!', `\0'};
```

A series of characters enclosed in double quotes ("") is called a **string constant**.

Declaring and Initializing Strings

a character array can be declared and initialized like this:

```
char arr_str[6] = {`H', `e', `l', `l', `o', `!'};
```

Here the array arr_str is treated as a character array. However,if you add a null character (\0) into the array, you can have the following statement:

```
char arr_str[7] = {`H', `e', `l', `l', `o', `!', `\0'};
```

You can also initialize a character array with a string constant. For example, the following statement initializes a character array, str, with a string constant, "Hello!":

```
char str[7] = "Hello!";
```

Declaring and Initializing Strings

You can declare an unsized character array if you want the compiler to calculate the total number of elements in the array. For instance, the following statement;

```
char str[] = "I like C.";
```

If you like, you can also declare a char pointer and then initialize the pointer with a string constant. The following statement is an example:

```
char *ptr_str = "I teach myself C.";
```

Strings

WARNING

Don't specify the size of a character array as too small. Otherwise, it cannot hold a string constant plus an extra null character. For instance, thefollowing declaration is considered illegal:

```
char str[4] = "text";
```

The following statement is a correct one: because it is big enough to hold the string constant plus an extra null character '\0':

```
char str[5] = "text";
```

String Constants Versus Character Constants What Is a String?

When a character variable ch and a character array str are initialized with the same character, x, such as the following,

```
char ch = `x';
```

```
char str[] = "x";
```

1 byte is required to hold char but 2 bytes are required to hold string.

Strings

Another important thing is that a string is interpreted as a char pointer. Therefore, you can assign a character string to a pointer variable directly, like this:

```
char *ptr_str; ptr_str = "A character string.";
```

However, you can not assign a character constant to the pointervariable, as shown in the following:

```
ptr_str = `x'; /* It's wrong. */
```

When a character variable ch and a character array str are initialized with the same character, x, such as the following,

Strings

It's legal to assign a character constant to a dereferenced char pointer like this:

```
char *ptr_str;

*ptr_str = `x';
```

Strings

```
1: /*  Initializing strings */
2: #include <stdio.h>
3:
4: main()
5: {
6:     char str1[] = {`A', ` `,
7:                    `s', `t', `r', `i', `n', `g', ` `,
8:                    `c', `o', `n', `s', `t', `a', `n', `t', `\0'};
9:    char str2[] = "Another string constant";
10:   char *ptr_str;
11:   int i;
12:
13:   /* print out str2 */
14:   for (i=0; str1[i]; i++)
15:        printf("%c", str1[i]);
16:   printf("\n");
17:   /* print out str2 */
18:   for (i=0; str2[i]; i++)
19:       printf("%c", str2[i]);
20:   printf("\n");
21:   /* assign a string to a pointer */
22:   ptr_str = "Assign a string to a pointer.";
23:   for (i=0; *ptr_str; i++)
24:        printf("%c", *ptr_str++);
25:   return 0;
26: }
```

```
A string constant
Another string constant
Assign a string to a pointer.
```

How Long Is a String?

Sometimes, you need to know how many bytes are taken by a string. In C, you can use a function called `strlen()` to measure the length of a string.

Strings

```
1: /*  Measuring string length */
2: #include <stdio.h>
3: #include <string.h>
4:
5: main()
6: {
7:    char str1[] = {`A', ` `,
8:                   `s', `t', `r', `i', `n', `g', ` `,
9:                   `c', `o', `n', `s', `t', `a', `n', `t', `\0'};
10:   char str2[] = "Another string constant";
11:   char *ptr_str = "Assign a string to a pointer.";
12:
13:   printf("The length of str1 is: %d bytes\n", strlen(str1));
14:   printf("The length of str2 is: %d bytes\n", strlen(str2));
15:   printf("The length of the string assigned to ptr_str is: %d bytes\n",
16:   strlen(ptr_str));
17:   return 0;
18: }
```

```
The length of str1 is: 17 bytes
The length of str2 is: 23 bytes
The length of the string assigned to ptr_str is: 29 bytes
```

Copying Strings with strcpy()

When a character variable ch and a character array str are initialized with the same character, x, such as the following,

The syntax for the strcpy() function is

```
#include <string.h>

char *strcpy(char *dest, const char *src);
```

Strings

```
1: /* Copying strings */
2: #include <stdio.h>
3: #include <string.h>
4:
5: main()
6: {
7: char str1[] = "Copy a string.";
8: char str2[15];
9: char str3[15]; 1
0: int i;
11:
12: /* with strcpy() */
13: strcpy(str2, str1);
14: /* without strcpy() */
15: for (i=0; str1[i]; i++)
16: str3[i] = str1[i];
17: str3[i] = `\0';
18: /* display str2 and str3 */
19: printf("The content of str2: %s\n", str2);
20: printf("The content of str3: %s\n", str3);
21: return 0;
22: }
```

```
The content of str2: Copy a string.
The content of str3: Copy a string.
```

Reading and Writing Strings - The gets() and puts() Functions

The gets() function can be used to read characters from the standard input stream.

The syntax for the gets() function is

```
#include <stdio.h>
char *gets(char *s);
```

The puts() function can be used to write characters to the standard output stream (that is, stdout).

The syntax for the puts() function is

```
#include <stdio.h>
int puts(const char *s);
```

Strings

```
1: /*  Using gets() and puts() */
2: #include <stdio.h>
3:
4: main()
5: {
6:    char str[80];
7:    int i, delt = `a' - `A';
8:
9:    printf("Enter a string less than 80 characters:\n");
10:   gets( str );
11:   i = 0;
12:   while (str[i]){
13:     if ((str[i] >= `a') && (str[i] <= `z'))
14:        str[i] -= delt; /* convert to upper case */
15:     ++i;
16:   }
17:   printf("The entered string is (in uppercase):\n");
18:   puts( str );
19:   return 0;
20: }
```

```
Enter a string less than 80 characters:
This is a test.
The entered string is (in uppercase):
THIS IS A TEST.
```

Using %s with the printf() Function

See previous example.

The scanf() Function

The syntax for the scanf() function is

```
#include <stdio.h> int

scanf(const char *format, …);
```

Strings

```
1: /* 13L05.c: Using scanf() */
2: #include <stdio.h>
3:
4: main()
5: {
6:    char str[80];
7:    int x, y;
8:    float z;
9:
10:   printf("Enter two integers separated by a space:\n");
11:   scanf("%d %d", &x, &y);
12:   printf("Enter a floating-point number:\n");
13:   scanf("%f", &z);
14:   printf("Enter a string:\n");
15:   scanf("%s", str);
16:   printf("Here are what you've entered:\n");
17:   printf("%d %d\n%f\n%s\n", x, y, z, str);
18:   return 0;
19: }
```

```
Enter two integers separated by a space:
10 12345
Enter a floating-point number: 1.234567
Enter a string:
Test Here are what you've entered:
10 12345
1.234567
Test
```

Block Scope

In this section, a block refers to any sets of statements enclosed in braces ({ and }). A variable declared within a block has block scope. Thus, the variable is active and accessible from its declaration point to the end of the block. Sometimes, block scope is also called local scope.

For example, the variable i declared within the block of the following main function has block scope:

```
int main()
{
   int i; /* block scope */
   . . .
   return 0;
}
```

Usually, a variable with block scope is called a local variable.

Nested Block Scope

You can also declare variables within a nested block. If a variable declared in the outer block shares the same name with one of the variables in the inner block, the variable within the outer block is hidden by the one within the inner block for the scope of the inner block.

```
int main()
{
   if(expression){
     if(expression){
       . . .
     }
    }
  return 0;
}
```

Scope and Storage Classes in C

```
1: /* 14L01.c: Scopes in nested block */
2: #include <stdio.h>
3:
4: main()
5: {
6:    int i = 32; /* block scope 1*/
7:
8:    printf("Within the outer block: i=%d\n", i);
9:
10:   { /* the beginning of the inner block */
11: int i, j; /* block scope 2, int i hides the outer int i*/
12:
13: printf("Within the inner block:\n");
14: for (i=0, j=10; i<=10; i++, j--)
15: printf("i=%2d, j=%2d\n", i, j);
16: } /* the end of the inner block */
17: printf("Within the outer block: i=%d\n", i);
18: return 0; 19: }
```

```
Within the outer block: i=32
Within the inner block:
i= 0, j=10
i= 1, j= 9
i= 2, j= 8
i= 3, j= 7
i= 4, j= 6
i= 5, j= 5
i= 6, j= 4
i= 7, j= 3
i= 8, j= 2
i= 9, j= 1
i=10, j= 0
Within the outer block: i=32
```

Function Scope

Function scope indicates that a variable is active and visible from the beginning to the end of a function.

In C, only the goto label has function scope. For example, the goto label, start, shown in the following code portion has function scope:

```
int main()
{
   int i; /* block scope */
   . . .
   start: /* A goto label has function scope */
   . . .
   goto start; /* the goto statement */
   . . .
   return 0;
}
```

Program Scope

A variable is said to have program scope when it is declared outside a function. For instance, look at the following code:

```
int x = 0;      /* program scope */
float y = 0.0; /* program scope */
int main()
{
   int i; /* block scope */
   . . .
   return 0;
}
```

Here the `int` variable `x` and the `float` variable `y` have program scope.

The Storage Class Specifiers -*The auto Specifier*

The auto specifier indicates that the memory location of a variable is temporary. In other words, a variable's reserved space in the memory can be erased or relocated when the variable is out of its scope.

Only variables with block scope can be declared with the auto specifier. The auto keyword is rarely used, however, because the duration of a variable with block scope is temporary by default.

The Storage Class Specifiers -*The static Specifier*

The static specifier can be applied to variables with either block scope or program scope. When a variable within a function is declared with the static specifier, the variable has a permanent duration. In other words, the memory storage allocated for the variable is not destroyed when the scope of the variable is exited, the value of the variable is maintained outside the scope, and if execution ever returns to the scope of the variable, the last value stored in the variable is still there.

For instance, in the following code portion:

```
int main()
{
  int i; /* block scope and temporary duration */
  static int j; /* block scope and permanent duration */
  .
  .
  return 0;
}
```

File Scope and the Hierarchy of Scopes

In C, a global variable declared with the static specifier is said to have file scope. A variable with file scope is visible from its declaration point to the end of the file. Here the file refers to the program file that contains the source code. Most large programs consist of several program files.

The following portion of source code shows variables with file scope:

```
int x = 0; /* program scope */
static int y = 0; /* file scope */
static float z = 0.0; /* file scope */
int main() {
  int i; /* block scope */
  .
  .
  .
  return 0;
}
```

The extern Specifier

As stated in the section titled "Program Scope," a variable with program scope is visible through all source files that make up an executable program. A variable with program scope is also called a global variable.

For instance, suppose you have two global int variables, y and z, that are defined in one file, and then, in another file, you may have the following declarations:

```
int x = 0; /* a global variable */
extern int y; /* an allusion to a global variable y */
int main() {
extern int z; /* an allusion to a global variable z */
int i; /* a local variable */
. . .
return 0;
}
```

The Storage Class Modifiers - *The const Modifier*

If you declare a variable with the const modifier, the content of the variable cannot be changed after it is initialized.

```
const double circle_ratio = 3.141593;
const char str[] = "A string constant";
```

In addition, you can declare a pointer variable with the const modifier so that an object pointed to by the pointer cannot be changed. For example, consider the following pointer declaration with the const modifier:

```
char const *ptr_str = "A string constant";
```

After the initialization, you cannot change the content of the string pointed to by the pointer `ptr_str`. For instance, the following statement is not allowed:

```
*ptr_str = `a'; /* It's not allowed here. */
```

The Storage Class Modifiers - *The volatile Modifier*

Sometimes, you want to declare a variable whose value can be changed without any explicit assignment statement in your program. For instance, you might declare a global variable that contains characters entered by the user. The address of the variable is passed to a device register that accepts characters from the keyboard. However, when the C compiler optimizes your program automatically, it intends to not update the value held by the variable unless the variable is on the left side of an assignment operator (=). In other words, the value of the variable is likely not changed even though the user is typing in characters from the keyboard:

```
void read_keyboard()
{
  volatile char keyboard_ch; /* a volatile variable */
  .
  .
  .
}
```