Course
ELEC-A7100
Course materials
Your points
Microsoft Teams
Code Vault

This course has already ended.

« 2 Functions for string handling Course materials 4 Precedence »
ELEC-A7100 / 4. Strings / 3 Helpful Qualifiers and Operators

Fourth round tasks as a `ZIP file.`

We will now look into **strings** in the C language. Actually strings are just an application of C arrays, with **char** - type array members (i.e., the individual characters). In the beginning you will learn to handle simple strings. The we step aside a for a bit, and take a look at few useful specifiers in C language, needed in understanding the string functions, that are discussed in the end of this section.

Helpful Qualifiers and Operators¶

Size of variable or data type¶

The **sizeof** operator can be used to query the size of a data type or variable (the size of same data type may differ between different system architectures). When we get to the data structures (later on this course) the actual size of the data structure may vary between different environments.

Here is a short example illustrating the use of **sizeof**:

```
#include <stdio.h>

int main(void) {
    short a;
    short *pa;
    printf("size a: %lu\n", sizeof(a));
    printf("size *pa: %lu\n", sizeof(pa));
    printf("size short: %lu\n", sizeof(short));
}
```

Above example shows the combination of getting size of the data type and size of the variable. **printf** uses `%lu` format specifier, because `sizeof` returns the unsigned long - data type value. Try to execute the program and find out what *sizeof* returns for each case. **Once you understand the program, see what happens when you change the data type, for example, from short data type to char or int.**

Declaring new data types¶

Sometimes it is inconvenient to use long data type specifications repeatedly. With **typedef** declaration, the program can specify new data types. The C library also specifies some commonly used type aliases for specific uses, such as **size_t**, an unsigned integer used for representing size of data objects in C. The return value of the **sizeof** operator is of **size_t** type. An alternative "mySize" type could be specified and used in the following way:

```
typedef unsigned long mySize;
long b;
mySize a = sizeof(b);
```

Typedef may become useful especially later with data structures and abstract data types.

Constant parameters and variables¶

Parameters and variables can be declared as constant by the **const** qualifier. Such variables can be read, but they cannot be written to.

Using the **const** qualifier is useful for documenting function interfaces: it tells that a particular function parameter is not going to be modified by the function. This is not mandatory, but helps the programmer defend against possible programming errors. An example of (an erroneous) function with a constant parameter, that the function tries to modify:

```
#include <stdio.h>

void a_func(const int *param)
{
    int a = *param;
    a = a + 1; /* ok, because the original parameter is unchanged */
    *param = *param + 1; /* will NOT work, because the parameter is changed */
}

int main(void)
{
    int a = 10;
    a_func(&a);
    printf("a: %d\n", a);
}
```

The above function declaration indicates that the parameter "param" is not to be referred by a pointer. Therefore the compiler returns an error on the line *7* that tries to modify the value behind "param" pointer. **If you remove the const qualifier, the program will compile.**

The const qualifier can also be used with variables. This makes the variable constant, that cannot be changed during the lifespan of the program. Note that in this case the variable needs to be initialized immediately.

For example:

```
const size_t maxSize = 10; /* global variables, visible throughput the program */

int main(void)
{
    int i;
    for (i = 0; i < maxSize; i++) {
        /* do something */
    }
}
```

C has a option to define global variables, such as the above. They will be visible in all functions of a program. Excessive use of global variables will be avoided in a good program structure as it makes difficult to read the larger programs, so they should be avoided whenever possible. Sometimes, global variables, however, are unavoidable.

Local static variables¶

Variables declared inside a function are called local variables. When a function is called, its local variables are normally valid only as long as the function is being executed. When the function is exited, these variables memory are released and data stored in these local variables are lost. An exception can be made to lifetime of these local variables using the keyword 'static' before the local variable declaration. A local variable can be defined using `static` - qualifier. In this case, the variable memory is maintained throughout the execution time of the program and the value is retained even after the function call. When the function is called again, the value stored during the previous call of the function is used.

In practice, these static variables are used rarely in program. Here is an example for local static variables.

```
#include <stdio.h>

int tuplaa(void) {
    static int arvo = 1;
    arvo *= 2;
    return arvo;
}

int main(void) {
    for (int i = 0; i < 10; i++) {
        printf("Now: %d\n", tuplaa());
    }
}
```

Thus, the function initializes a static variable only during the first call of the function and during each function call, return value is doubled. **What is the return value after the final call of the function ?** ,**What happens when you delete `static` qualifier from the variable declaration ?**

Static functions and global variables¶

The visibility of functions and global variables can be restricted by the use of **static** qualifier. This is completely different from the static local variables, but both just happened to have same keyword. static functions(static global variable) are functions(variable) that are only visible to other functions in the same ".c" file. Larger software, consisting of hundreds of source files (i.e.,program modules) utilize this feature.

Linux is an example of a large program, which consists of a large number of C-language source files. For example, there is a function in the file tcp_input.c

```
static void tcp_sndbuf_expand(struct sock *sk)
{
    /* Code */
}
```

which can be used only tcp_input.c and not in any other files. Thus, developers can control which functions can be used by other program modules, and which ones are "hidden". The aim is again to prevent the programmer's mistakes and avoid the bad software design. Most of the software life cycle is in fact surprisingly long.

« 2 Functions for string handling Course materials 4 Precedence »