Functions and Storage Options

Overview

- Functions general
- Scope of variables
- Recursive functions

2

Functions and Storage Classes

As mentioned in the introduction, C is a function oriented language. A C program is really a number of function calls. When creating C programs we therefore have to think in a function oriented way. A function is a discrete block of code which evaluates to a result. You can look at it as a black box. You put something in it, it performs some calculations and you get a result. When using functions we do not care how it performs the calculations. All we are interested in is what the result is and what we have to deliver to the function so that it can perform its calculations.

Function Signatures

- A function consists of a return type, name and arguments
- The declaration is a description of the function characteristics (signature)

```
<type> name([<arg_type arg, arg_type arg, ...>]);
```

- The definition is the implementation of the function (body with code)

```
<type> name([<arg_type arg, arg_type arg, ...>])
{
}
```

3

Functions an introduction

A function is a discrete block of code which returns a result. We have already seen a few uses of functions. For example *printf()* is a function used for output to the screen. Also the well know *main()* is a function namely the start function of your application. If we use the *printf()* function we can also look at it as a black box. We give it what we want to print. What actions it needs to perform to print the actual string on the screen is not what we are interested in.

Creating functions can be divided into two parts:

- Declaring functions
- Defining functions

Declaring Functions

The declaration of a function specifies what the prototype of the function is. It states to the user and the compiler how the function needs to be used. This prototype consists of three parts:

- Return type: the resulting value of the function
- Name: the name of the function for example *printf*
- The arguments: the input parameters

This declaration is needed before we are going to use the function. The general form of the declaration is:

```
<return_type> function_name([arg_type,...]);
```

After we declared the function we can use the function by using its name and passing its arguments.

The return type of the function is the resulting value of the function. Most functions in C return a value which either is the result of some calculation or a condition code. This return value can be any of the types used in the previous modules. When a function does not return a value we have to specify this using the type void (nothing)

```
void function_name();
```

The argument list is the list of values the function needs to perform its function. This list is not mandatory. Some functions do not need arguments but just perform some action. If we do pass arguments we have to specify in the function declaration what the type of each argument is. The arguments are separated by a comma. If no arguments are passed we have to specify void.

```
void print(void);
/* returns: nothing
   name: print
   arguments: nothing
*/

int sqr(int);
/* returns: int
   name: sqr
   arguments: one int
*/

int multiply(int,int);
/*
   returns: int
   name: multiply
   arguments: two ints
*/
```

The function declaration needs to be repeated in every source file where we use the function. If 20 different source files use the function *multiply()* we have to declare it in all of those source files. An easier way is to create a so called header file which contains the declarations of commonly used functions (see next module). Whenever we want to use a function in a source file we only have to include the header file which contains the declaration of the function. This mechanism has already been used in the previous chapters. Whenever we wanted to use the *printf()* function we had to include the header file *stdio.h*. Including a header file is done with the statement

```
#include <filename.h>
```

The compiler includes the file specified between the brackets when compiling the code.

Defining Functions

After declaring the function we have to create the function body (implementation). The body of the function is placed in a source file. This can be any source file. It does not have to be the source file where the function is used (where the function is declared).
The definition of a function is almost the same as its declaration. The function declaration has a semi colon (;) at the end. The function definition has no semi colon but the body (implementation) of the function between brackets.

The body of the function has the following general form.

```
return_type name([argument list])
{
    /* Declaration of variables */

    /* Executable code */

    /* Ending of function (see next few sections) */
    return [expression];
}
```

A simple example:

```
void print(void)
{
    printf("Hello this is a simple function\n");
}
```

Function Calls

- Before calling a function it needs to be declared
- Call function by using its name and pass the arguments between ()
  - ```
    void main()
    {
      print(); /* Call function print() */
    }
    ```

4

Function Calls

After declaring and creating a function we can use it in any other function by executing it. We can execute a function with a function call. We can call a function by using its name and passing its arguments between ().

```
void print(void);

void main()
{
    int a;

    print();      /* Call to function print */
    a = a + 10;
}

void print(void)
{
    printf("Hello this is a simple function\n");
}
```

The function *main()* is the start of your application from this function we can call other functions. The example first creates a variable 'a'. The next line calls the function *print()*. By calling a function the system jumps to the begin address of the function and starts executing the statements of that function. When the function ends the system jumps back to the point from where the function was called.
```

Functions and Return Value

- If function has no resulting value specify `void`
  - `void print(void);`
- If it does return a value, specify the type of the return value
  - `double pi(void);`
- Return the value using the `return` statement
  - `return [expression];`
- Can use return value of function in calculations

5

Functions and Return

A function can return a value. If a function does not return anything we have to specify `void`.

```
void print(void); /* Returns nothing */
```

When a function does return a value we have to specify what kind of value it returns.

```
double pi(void);
```

The example shows the signature of a function *pi()* which has no arguments but returns a `double` value. We can return a value in a function by using the *return* statement followed by what we want to return.

```
double pi(void)
{
    return 3.1415927;
}
```

If we specify that a function returns a result and we forget to place a return statement in that function the compiler will generate a warning or error depending on the compiler switches.

If a function returns a value, we can use it in a calculation or assign it to a variable.

```
void main()
{
    double value;

    value = pi();       /* value is what the function pi() returns */
    value = 2 * pi();   /* value is 2 times the result of function pi() */
    printf("Pi: %f\n", pi());  /* Prints the value of pi on the screen */
}
```

It is not required to assign the value a function returns to a variable. It is possible to discard the return value of a function.

```
void main()
{
    double value;
    pi();
}
```

In the above example we call the function *pi()* which returns a number. This number is not used in any calculation or assigned to any value. Because we do not use the value of the *pi()* function it is in this case redundant to call the function. This is not the case for all functions. Some functions return an error value to specify if an operation succeeded or failed. Sometimes, if we do not use error checking, we discard this value. But it is not redundant to call that function. For example the *printf()* function also returns a value but it is not redundant to call it to print something on the screen.

When a function returns a void and we assign the result value of that function to a variable we get an error. This because void means nothing is returned.

```
void print(void);

void main()
{
    int val;
    val = print();      /* Causes an error */
}
```

Function Arguments

- When a function has arguments we have to specify their types
  - `int multiply(int a, int b);`
- Call function passing values or variables
  - `multiply(10, 20);`
- Argument values are copied onto the stack
  - Call by value
- Changing copies has no effect on original variables

6

Function Arguments

Often functions need input values to perform some action. For example the *printf()* function needs the string to be passed that we want to print on the screen. When declaring a function we have to specify the types of the input values (arguments) of that function.

```
int multiply(int a, int b);
```

Inside the function *multiply()* the variables 'a' and 'b' exist without declaring them inside the function. The argument list can be seen as part of the variable declaration section inside a function.

```
int multiply(int a, int b)
{
    return (a * b);
}
```

The function multiply returns the multiplication of the two input values. We have to pass these values to that function upon calling that function.

```
void main()
{
    int var1;
    int var2;
    int res;

    res = multiply(var1, var2); /* Pass the values of var1 and var2 */
    res = multiply(40, 30);     /* Pass 40 and 30 */
}
```

The names of the variables we pass to the function multiply do not have to be the same of as the argument variables inside the function. The variables 'var1' and 'var2' are not the same as 'a' and 'b'. They can be the same but it is not mandatory.

The names of the variables 'a' and 'b' are bounded to the function *multiply()*. Passing the values var1 and var2 is called "call by value". The values of 'var1' and 'var2' are copied onto the stack and are given the names 'a' and 'b'. When the function *multiply()* ends, these variables are destroyed. Changing the variables 'a' and 'b' inside the function *multiply()* have no effect on the variables 'var1' and 'var2' because 'a' and 'b' are copies of 'val1' and 'val2'.

Recursive Functions

- Functions can call itself
- Upon recursive call all local variables are created again
- Often used for mathematical expressions and algorithms

7

Recursive Functions

After creating and declaring functions the functions can be executed by calling the function. Most times function calls are placed inside other function bodies. For example the *main()* function (the beginning of the program) is the start of a chain reaction of function calls. From the *main()* you can execute a function. For example *multiply()* which performs some action and may return a result. The function you initially call can call other functions and that function can call other functions as well so creating the chain reaction.

It is however possible that a function calls itself with different arguments. Such a function is called a recursive function. Recursive functions are used a lot for data structures and mathematical functions.
There is one important rule when creating recursive functions:

- They need a stop condition

For example let us take the algorithm for getting the Greatest Common Divisor.
The algorithm for finding the greatest common divisor (GCD) is:

GCD = x if y == 0
else GCD = GCD(y, x % y)

If we implement this in a function:

```
int GCD(int x, int y)
{
    if(y == 0)
          return x;
    return GCD(y, x % y);
}
```

Scope of Variables

- Variables can be used in their scope
- Variable identifier limited to its scope
- Types of scope
  - Local scope
  - Function scope
  - Program scope
  - File scope

8

Scope and Storage Classes

Each name which is used in a C program has a so-called "scope". This can be seen as the range of visibility of that name. For example, we could make a global variable which is known in every part of a program, including the functions which make up that program. Similarly, we can define a name which is only known in a local piece of code.

The types of scope are:

- Local scope: This refers to names which are declared and only usable within blocks.
- Function scope: This refers to names declared and only usable within functions.
- Program scope: These are names which are declared outside any function or block and can be used everywhere, even from other source files. Names with program scope are often said to be global.
- File scope: As with program scope but declared to be only visible in the current source file.

Automatic Variables

- Variables inside functions are called automatic variables
- Automatic variables are created upon function start
- Automatic variables are destroyed upon function end

9

Automatic Variables

- Inside functions
- As a function argument

Automatic variables are the variables declared inside functions. When the function begins the variables are created on the stack and when the function ends they are destroyed. These variables include the arguments of the function.

```
void print(void)
{
    int auto_1;
}

void print_val(int a)
{
    int local_var;
}

void main()
{
    int local_main;
    print();
    print_val(local_main);
}
```

Inside the function *main()* the variable *local_main* is declared. This variable is created when the function *main()* starts and is destroyed when the function ends. It is thus an automatic variable. The call to the function *print()* executes that function. When the function *print()* is executed the variable *auto_1* is created on the stack. When the function is finished this variable is destroyed. The function *print_val()* creates the argument *a* and its variable *local_var*. Both these variables are created when the function starts and destroyed when it ends.
These local variables are therefore only available inside these functions. The function *print_val()* cannot use *auto_1*.

Global Variables

- A global variable is declared at file scope
  - Can be accessed by all functions
- Can be used in any source file by any function

10

Global Variables

Until now we declared variables inside functions. These variables are called automatic variables. They can only be used inside these functions or we have to pass them as an argument to another function. There are however variables which can be used by all functions of a program in the same source file or even in another source file. These variables are called global variables and must be declared outside a function.

```
int linenumber;
void print_line(void);

void main()
{
    int local;

    linenumber = 0;
    print_line();
}

void print_line(void)
{
    printf("Line : %d\n", linenumber);
    linenumber++;
}
```

The variable *linenumber* can be used by any function. The function *main()* gives it the value o and the *print_line()* function uses it to print *linenumber* on the screen. The disadvantage of these global variables is that any function can change them. We have no control any more over that variable. It is therefore advisable to avoid the usage of global variables.

When we declare an automatic variable with the same name as the global variable we overrule that global variable inside the function where the automatic variable is declared.

```
void main()
{
    int linenumber;
}
```

External Variables

- Possible to use global variable of another source file
- Use keyword ‘extern’ to declare it
  - `extern int linenumber;`
- The keyword extern says the variable is created elsewhere instead of this source file
- We need to declare it else we cannot use it

11

External Variables

When a source file has a global variable we can use it in any other source file as long as those source files are linked together. But the compiler checks our code to see if we use variables that are not declared. If we create a second source file with the following function the compiler would give an error.

```
void print_more_lines()
{
    printf("Line : %d\n", linenumber);
    linenumber++;
    printf("Line : %d\n", linenumber);
    linenumber++;
}
```

The compiler always compiles one source file at a time. In this source file the variable *linenumber* was not declared so it will give an error. But we want to use the global variable of the other source file. If we declare the global variable *linenumber* again the program would have two global variables with the same name. To declare the variable but don’t allocate memory for it (just a place-holder) we have to use the keyword *extern* which specifies that the variable is defined in another source file.

```
extern int linenumber;

void print_more_lines()
{
    ...
}
```

If we specify an external variable and we forget to declare it in another linked source file the linker will give an ‘unresolved’ error.

Static Global Variables

- Normally a global variable can be used by any source file
- Use the static specifier to bound its use to the current source file
  - `static int linenumber;`

12

Static Global Variables

The scope of a global variable is the entire program. It is often preferable to have a global variable in one source file but this global variable should not be accessible by any other source file. This is possible by specifying the variable as a static global variable. A static global variable is bounded to the scope of the source file.

Functions and Static Variables

- A static function variable is a kind of persistent variable
- Remembers the value between function calls
- Holds last value assigned
- Also called internal static variable

13

Static Variables and Functions

In the previous example of global variables we use a global variable *linenumber* which was incremented each time a line was printed. We also stated that we advise to avoid the usage of global variables. So if we wanted to have the *linenumber* variable but not using a global variable we have to create it inside the function *main()* and pass it every time to the function *print()*:

```
int print(int linenumber);

void main()
{
    int linenumber = 0;
    linenumber = print_line(linenumber);
}

int print_line(int linenumber)
{
    printf("Line : %d\n", linenumber);
    linenumber++;
    return linenumber;
}
```

This has a few disadvantages. First we let the function *main()* create something that is used only by the function *print_line()*. The second disadvantage is the rather cryptic call to the function *print_line()*. We have to pass it the variable *linenumber* and let the function *print_line()* return the changed value. Because, as you know, with functions the value is passed by value (a copy is made) so changing the argument *linenumber* does not affect the original variable *linenumber* in the *main()* function.
We can solve this by using a so called internal static variable. An internal static variable is assigned an initial value but from there on it holds its last assigned value. An internal static variable is not destroyed when the function ends.

Using this technique the *print_line()* function is transformed into:

```
void print_line(void);

void main()
{
    print_line();
    print_line();
    print_line();
}

void print_line()
{
    static int linenumber = 0;

    printf("Line : %d\n", linenumber);
    linenumber++;
}
```

The variable *linenumber* is initialised once and from that point on will remain its last assigned value.

The output of the example is:

```
Line: 0
Line: 1
Line: 2
```

Register Variables

- Request for variable to be placed in register
  - `register int a;`
- Access to register variable is faster
- Not always granted

14

Register Variables

Register variables are used for variables to speed up calculations. If a variable is a register variable it is placed in a processor register when using it in an operation or calculation. Access to a variable is stored in a processor register is much faster than access to a variable in main memory. This is however a request to the system but is not always granted.
Modern compilers will optimise your code automatically and will place variables in a register when possible.