コース: C Programming

C Programming

作家: Anrich Tait

Abstract

Quick explanation of document.

Contents

Chapter 1

Overview of C

1.1 Objectives:

1. Become familiar with general form of a C program and it's basic elements,
2. Why you should write comments
3. Use of data types and the differences between the various data types.
4. How to declare variables
5. How to change the values of variables
6. Evaluate arithmetic expressions
7. Read data values into a program and display them
8. Understand strings
9. Redirection to use files for input/output
10. Understand the differences between runtime errors, syntax errors and logic errors, and how to debug each.

1.2 The basics:

1. Both #define and #include are handled by the pre-processor, this is why we cannot change a variable or value that has been #define. The compiler is not capable of going back to change it.

2. Your variable names are also known as identifiers.

1.2.1 Steps in the compiling process:

1. preprocessing: Scans header files for relative prototypes. (So the compiler knows what printf is). Also looks for variables with #define

2. compiling: Turn code into assembly language before it is turned into 0's and 1's.

3. assembling: Where the assembly language is turned into machine code.

4. linking: Combines all the machine code into the final program that can be executed as your program.

1.2.2 Data Types:

1. int : short for integer. An int can be any whole number betweeen -32767 and 32767.

2. double : Basically a real number, which has an integral part and a fractional part that is seperated by a decimal point. For example: 3.14159; 0.0005; 150.05. Scientific notation can be used for doubles, for example: the real number

$$1.23 * 10^5$$

   is equivalent to 123000.0 where the exponent '5' means "move the decimal point 5 places to the right. In Scientific notation this number is written as 1.23E5. Read the letter e or E as "times 10 to the power": 1.23e5 means 1.23 times 10 to the power of 5. If the exponent has a minus sign the decimal point is moved to the left.

Only use double when necessary, using the int data type is faster in most cases. Also int computations are always precise whereas double numbers can have rounding errors.

1.2.3 Input/Output operations and functions

Data can be stored in memory in 2 diffrent ways:

1. By assinging it to a variable

2. By receiving input from a function like 'scanf'

When data is assigned to a variable through the use of a function like 'scanf' it is known as an 'input operation'. This data needs to be input each time the program is executed.
That data is then stored in memory and can be output by using an **output operation**.

All input and output in C is handled by **input/output functions**. These functions are activated via a **function call**.
When you call a function you are basically telling it do something with the information you give it. The program will not continue until that functions says it is finished.

The most common functions are contained in the stdio.h header file. Examples of these functions are 'printf' for output and 'scanf' for input.

A brief look at using 'printf()' The syntax for using printf is as follows:

```
1 printf("%[flags][width][.precision][len]specifier", var);
```

Below are some examples of each option in the above syntax:

[flags]:

1. - : left justify

[width]:

1. (number) : minimum number of characters to be printed

[precision]

1. .number : number of digits to be written. The number gets padded by 0s if the resulting number is short

[specifier]: denote type of output

1. c : character
2. d : integer

3. f : flaoting point

4. s : string

Some examples of printf()

```c
/* Example for printf() */
#include <stdio.h>

int main(){
    printf ("Integers: %i %u \n", -3456, 3456);
    printf ("Characters: %c %c \n", 'z', 80);
    printf ("Decimals: %d %ld \n", 1997, 32000L);
    printf ("Some different radices: %d %x %o %#x %#o \n",
                100, 100, 100, 100, 100);
    printf ("floats: %4.2f %+.0e %E \n", 3.14159,
                3.14159, 3.14159);
    printf ("Preceding with empty spaces: %10d \n", 1997);
    printf ("Preceding with zeros: %010d \n", 1997);
    printf ("Width: %*d \n", 15, 140);
    printf ("%s \n", "Educative");
    return 0;
}
```

```
Integers: -3456 3456
Characters: z P
Decimals: 1997 32000
Some different radices: 100 64 144 0x64 0144
floats: 3.14 +3e+00 3.141590E+00
Preceding with empty spaces:       1997
Preceding with zeros: 0000001997
Width:             140
Educative
```

Chapter 2

C

Things to focus on when writing code:

1. Correctness
2. Design
3. Style

Unsorted notes:

1. Source code is compiled into machine code via gcc(compiler).
2. Arguments are inputs to functions.
3. Functions take arguments and result in output
4. Types of outputs are: side effects (visual, audio output), return values (that can be used).
5. Look at the mario program.
   Notice that a do while is used at the top. This checks whether the user has co-operated by inputting a number bigger than 0. If the user inputs 0 it will again ask for the width.
6. Integers divided by integers truncate to only the decimal on the right of the "." it throws away all decimals
7. How ever you can type cast with (type) variable name. (see the calculator program for example)
8.

Chapter 3

Arrays

The compiling process:

1. preprocessing: Scans header files for relative prototypes. (So the compiler knows what printf is)
2. compiling: Turn code into assembly language before it is turned into 0's and 1's.
3. assembling: Where the assembly language is turned into machine code.
4. linking: Combines all the machine code into the final program that can be executed as your program.

Debugging
buggy.c

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5          for (int i = 0; i <= 3; i++)
6          {
7                  printf("i␣is␣%i\n", i);
8                  printf("#\n");
9          }
10 }
```

The code above will print a column of 3 #.
The commented line is an example of printf debugging. (A method of debugging where the programmer uses printf to check the values of variables while the code runs).
In this example our issue was that we expected a column of 3 # symbols but instead got 4. After using the printf debugging method it was easy to see that the "i" variable was incremented one to may times due to this little code section.
for (int i = 0; **i <= 3**; i++)

Chapter 4

Functions

4.1 Outline:

The following questions/topics will be addressed:

1. What are functions?
2. How are functions utilised?
3. Examples of using functions.

4.2 Definition

Functions break large computing tasks into smaller ones. This helps make code easier to change in the future and can also help with readability. Appropriate functions also hide details of operations from parts of the program that don't need them. C programs generally consist of many small functions rather than fewer large ones. The most basic example of a function is main(), every program has one and it is executed before any other functions in the program. Functions can be used for any task but generally it is best for each function to accomplish a specific task. A function declaration provides the actual body of the function.

The standard library provides numerous builtin functions that your program can call for example:

1. strcat() to concatenate strings
2. memcpy() to copy the memory location of one parameter to another.

4.3 Defining a function

Syntax:

```
return_type function_name(parameter list)
{
        body of function
}
```

Here is a break down of each part of this syntax:

1. return_type: The type of value the function will return.
2. function_name: Name of the function.
3. (parameter list): list of parameters passed to the function.
4. body of function: code to be executed

Example:

```
int add(int a, int b)
{
        int result = a + b;
        return (result);
}
```

Break down:

1. int: Return type of function (integer).
2. add: Name of function.
3. (int a, int b) parameters.
4. int result ā + b: declare result variable and code that will be executed.
5. return (result) returns the value of the result variable.

Note: if a function does not return anything it can be specified as void.

Functions can be called from other parts of a program by using the function name and passing in the required parameters.

```
int x = 5;
int y = 7;

int z = add(x, y);
```

The function 'add' is called with x and y as the parameters. The x parameter will replace 'a' and 'y' will replace 'b'. The result is stored in 'z'.

4.4 Function types

1. Functions that return a value: Perform a specific task and return a value of a specific data type using the return statement. The previous "add" function is an example of this.

```
char get_first_char(char *str)
{
        return[0];
}
```

2. Functions that return nothing: Perform a task but does not return any value.

```
void print_hello()
{
        printf("Hello, world!")
}
```

3. Functions that take no parameters: Does not take any paramters but perfomrs a specific task.

```
void clear_screen()
{
        system("clear");
}
```

4. Functions that take parameters but return nothing.

```
void greet(char *name)
{
        printf("Hello, %s", name);
}
```

4.5 Calling a function

When a function is called, control of the program is transferred to the function until it's return statement or closing brace is reached. Then the main() function is used again.

To call a function you need to pass the required parameters.

Passing parameters to a function basically means providing values or variables to the function.

Going back to the 'add' function as an example.
x and y are the parameters that are passed to the function. When the function is called with arguments the values of the arguments are assigned to the corresponding parameters. So if we call the 'add' function with arguments 3 and 4 the values 3 and 4 are assigned to x and y respectively.

4.6 Function arguments

For a function to use arguments it must declare variables that accept the values of the arguments. These variables are called the formal parameters of a function.

Formal parameters behave like other local variables and are created upon entry to the function and are destroyed upon exit.

There are two ways arguments can be passed to a function:

4.6.1 Call/Pass by value:

The function receives a copy of the argument passed, not the original argument. So any changes made to the argument inside the function do not change the original value of the argument.

4.6.2 Call/Pass by reference:

Calling a function by reference involves passing arguments to a function using the parameters address. This way the actual value is changed, not just a copy of it.

To pass a variable by reference in C, you must declare the function parameter as a pointer type using the * operator.

The following example function takes a pointer to an integer as a parameter.
Note: by default C passes arguments by value.

4.7 Function Prototypes

4.7.1 What is a function prototype

A function prototype is a declaration of a function that describes the function's interface to the rest of the program. It specifies the function's name, return type and parameter types.

These are usually specified at the beginning of a file before any other functions.

Syntax:

```
return_type function_name(parameter_list);
```

For example:

```
int max(int numOne, int numTwo);
```

This prototype declares a funtion named max that takes two integer parameters.

4.7.2 Why are function prototypes used?

1. Type checking: By declaring the function prototype before calling the function the compiler can check that the arguments passed to the function match the expected types.

2. Avoid implicit declaration: If a function is not given an implicit type the compiler assumes it is an integer, this can lead to misleading errors relating to different return types or number of arguments.

3. Optimization: Allows the compiler to generate more efficient code by providing information about the function's interface. This can lead to smaller, faster code.

4. Documentation: Can server as documentation for the function, making it easier for other programmers to understand how to use a function.

Chapter 5

Pointers

Syntax:

<type> *<name>

1. type = is the data type that the pointer will point to.
2. name = is the name of the pointer variable.

Example:

To declare a pointer variable that will point to an integer value the following syntax is used,

```
int *ptr;
```

This declares a pointer variable named "ptr" that can point to an integer value.

To assign a value to a pointer variable the address-of operator (&) is used to get the memory address of the variable.

Example:

Assign the address of an intger variable named "x" to the pointer variable "ptr",

```
int x = 10;
int *ptr = &x;
```

This assigns the address of ”x” to ”ptr”, so now ”ptr” points to x. To access the value that is stored at the memory location pointed to by a pointer variable the dereference operator is used (*).

5.1 Concept

Pointers are a fundamental concept in the C programming language, and understanding how they work is essential for writing efficient and effective C code. A pointer is a variable that stores the memory address of another variable. By using pointers, C programs can directly access and manipulate the values of other variables in memory, providing a powerful tool for building complex data structures and optimizing program performance.

At its core, a pointer is simply a memory address. When you declare a pointer variable in C, you are creating a variable that can store the address of another variable in memory. To declare a pointer variable, you use the * symbol before the variable name, like this: int *ptr;. This declares a pointer variable named ptr that can store the memory address of an integer variable.

Once you have declared a pointer variable, you can use the & operator to get the memory address of another variable, and assign that address to the pointer variable using the assignment operator =. For example, if you have an integer variable named x, you can get its memory address with &x, and assign it to the pointer variable ptr like this: ptr = &x;. Now, ptr points to the memory location of x, and you can use the pointer to access and manipulate the value of x directly.

One of the most common uses of pointers in C is to pass variables by reference to functions. In C, when you pass a variable to a function as an argument, the function gets a copy of the variable's value. However, by passing the memory address of the variable instead, you can allow the function to directly access and modify the variable's value. This can be useful for functions that need to modify variables outside of their own scope.

Another important use of pointers in C is for dynamic memory allocation. In C, you can use the malloc() function to allocate a block of memory of a specified size at runtime. The malloc() function returns a pointer to the beginning of the allocated memory block, which you can then use to access and manipulate the memory. Once

you are done with the memory, you should free it using the free() function to avoid memory leaks.

Pointers can also be used to create complex data structures in C, such as linked lists, trees, and graphs. By using pointers to connect nodes together in these structures, you can create efficient and flexible data structures that can be easily traversed and manipulated.

However, while pointers are a powerful tool in C programming, they can also be dangerous if used improperly. Pointer errors, such as dereferencing a null pointer or accessing memory that has already been freed, can cause program crashes or even security vulnerabilities. To avoid these errors, it is important to carefully manage and validate pointers in your code, and to use best practices for pointer manipulation, such as initializing pointers to null and checking for null before dereferencing them.

In conclusion, pointers are a fundamental concept in C programming that allow you to directly access and manipulate the values of variables in memory. Pointers are used extensively in C for passing variables by reference, dynamic memory allocation, and creating complex data structures. While pointers can be a powerful tool, they can also be dangerous if used improperly, and it is important to carefully manage and validate pointers in your code to avoid errors and vulnerabilities.

Summary:

The syntax for a pointer consists of the data type followed by an asterisk and the name of the pointer variable. The address-of operator (&) is used to assign a value to the pointer, and the dereference operator (*) is used to access the value stored at the memory location pointed to by the pointer.

5.2 Examples

5.2.1 How to use pointers:

In this example, swap() is a function that takes two integer pointers as arguments and swaps their values. By passing the memory addresses of a and b to swap() using the & operator, we allow the function to directly modify the values of a and b.

In this example, we use the malloc() function to dynamically allocate an array of 10 integers, and store its memory address in the pointer variable arr. We then use a loop to assign the values 0 to 9 to the array elements. Finally, we use the free() function to release the allocated memory.