C: 101

Some notes on the C programming language. To compile and run a C program using `gcc`:

```
$ gcc hello.c -o hello
$ ./hello
Hello, World!
```

Variables and Types

Variables and constants are the basic data objects that can be used in a program. These are referenced by name, these names are made up of letters and digits, but the first character of the name need to be a letter (the underscore `_` counts as a letter).

Declarations

All variables must be declared before use. A declaration specifies a type and a list of one or more variables of that type, for example:

```
int i;
char c;
```

or, using a list of variables

```
int max, min, step;
```

Variables can be initialized in the declaration, for example:

```
int max = 10, min = 0, step = 2;
```

Basic Data Types

There are four basic data types:

- `int` for integer numbers, for example `int i = 0;`.
- `char` for single characters, for example `char c = 'a';`.
- `float` for single-precision floating point, for example `float f = 3.14;`.
- `double` for double-precision floating point, for example `double d = 9e+12;`.

The following modifiers can be applied to these basic types: `short`, `long`, `signed`, and `unsigned`.

Constants

Constants are fixed values that do not change during program execution. This prevents unintended changing their values.

There are two ways of defining constants:

- `#define` use the preprocessor to define a constants.
- `const` using the reserved keyword to declare a constant variable.

Examples of the first option are:

```
#define MAX  1000
#define NEWLINE '\n'
```

Where the `#define` keyword is followed by the name of the constant, and the value, i.e.:

```
#define <name> <value>
```

During the preprocessor run, all the instances of the constant in the program are replaced by the value.

The second option is prefixing a variable declaration with the `const` keyword:

```
const int length = 10;
```

Operators

The binary (use two operands) arithmetic operators are:

- `+` for summing values.
- `-` for subtracting.
- `*` for multiplication.
- `/` for division.
- `%` for the remainder of integer division.

The relational operators used to compare values are:

- `>` to test for greater then.
- `>=` for greater or equal than.
- `<` for less than.
- `<=` for less or equal than.
- `==` for equality.
- `!=` for not equal.

Logical operators:

- `&&`
- `||`

When operators have different types, they are converted to a common type (this is not always possible). In general the only automatic conversions are those from a "narrower" operand into a "wider" operand without loosing information (for example, from integer to float).

Type casting can be used to force a conversion of a value, by including the intended type between parentheses. For example:

```
int sum = 17, count = 5;
double mean;
mean = (double) sum / count;
```

Arrays

An array is a structure to store a sequential, fixed size, collection of elements of the same type. Arrays indexes in C start a 0. The generic way of declaring arrays is:

```
<type> <name>[<size>];
```

For example, declare an array of integers of size 10 (the index of this array goes from 0 to 9):

```
int array[10];
```

Arrays can also be initialized in the declaration, for example to initialize the previous example with the numbers from 1 to 10::

```
int array[10] = {1,2,3,4,5,6,7,8,9,10};
```

To access the values of an array declare an expression with the array name, and the index of the intended value, for example to add the first two elements of the previous array:

```
add = array[0] + array[1];
```

To scroll through an list of elements of an array of size `N` a loop can be used, the following `for` loop iterates over all the elements of the array, starting at index 0, and finishing at index `N-1`

```
for (i = 0; i < N; i++) {
   (...)
}
```

Strings

Strings are a specific example of a one-dimension array. Where each element in the data collection is a single character. One important detail, the last element in a string is *always* the literal `\0`. An example of declaring and initializing a new string following the previous array examples is:

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

The following shortcut can be used to achieve the same declaration:

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

Note that in the second option there is no need to explicitly add the trailing `'\0'` literal. Strings can be transversed and handled like any other array.

Functions

Functions are used to organize a set of statements that execute a specific task. This isolates specific parts of the program, allowing to write code that is easier to read, understand and maintain.

The generic way for defining a function is:

```
<return_type> <name>(<argument_list>) {
  <body>
}
```

For example to declare a function named `add` that has two arguments (two) numbers, and returns the numbers addition:

```
int add(int x, int y) {
  return x+y;
}
```

The `return` keyword is used to return the intended value and control flow back to the function caller. To call a function, use the function name, and use parentheses to declare the arguments list. For example to call the `add` function example to sum the numbers `3` and `5`:

```
sum = add(3, 5);
```

All function argument are passed to functions by value for default, this means that you can change the value of the argument inside a function without changing the original variable from which was called. Arrays and strings are always passed by reference by default, this means that if you change an array inside a function the original array will also be change, because they point to the same data.

Functions must be declared before used, a function prototype (just the function signature) can be made explicitly before describing the function body. For example, the function prototype for the `add` function is:

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

Structs

Structures are used in C to organize together a collection of variables so it can be treated as an unit. For example to represent a point in 2D space the following *struct* can be used:

```
struct point {
  int x;
  int y;
};
```

Where `point` is the tag name for the declared *struct* that contains two variables of type `int`: `x` and `y`. Variables inside a *struct* are usually called members. This declaration defines a new type: `struct point` which can be used to declare new variables as any other type. For example to declare two new variables `p1` and `p2` of this type:

```
struct point p1, p2;
```

These variables can also be initialized in the declaration:

```
struct point p = { 10, 20 };
```

This declares a new point `p` where `x` is initialized with the value `10`, and `y` with the value `20`.

Individual members of the *struct* can be accessed with the `.` (single dot) operator. For example to print the values of the previous example:

```
printf("%d, %d", p.x, p.y);     // prints: 10, 20
```

Or to update the values:

```
p.x = 30;
p.y = 40;
```

Structures can be nested, for example a rectangle can be defined using two points:

```
struct rectangle {
    struct point p1;
    struct point p2;
};
```

The `.` operator is used to update or retrieved values in the the nested structure:

```
struct rectangle r;
r.p1.x = 10;
r.p1.y = 20;
printf("%d, %d", r.p1.x, r.p1.y);
```

Structures, as any other variable type, can be returned from functions, or handled as arguments. For example to define a function that creates and returns a new point given its' x an y coordinate:

```
struct point new(int x, int y) {
    struct point n;
    n.x = x;
    n.y = y;
    return n;
}
```

typedef

`typedef` is used in C to give new names to types. For example the declaration

```
typedef int Length;
```

creates a new alias (`Length`) for the type `int`. This new alias can be used exactly as any other type. For example to declare a new variable:

```
Length l = 10;
```

This approach can also be used to give name types to user defined *structs*. For example to give a new name (`POINT`) to the example *struct* defined in the previous section:

```
typedef struct point {
  int x,
  int y,
} POINT;
```

Now instead of having to use `struct point`, we can use `POINT` as a type. For example in the function prototype `new` that returns a new point:

```
POINT new(int x, int y);
```

Pointers

A pointer is a variable that stores the address of another variable. For example, the following declaration creates a new variable `i` of type `int` with the value `10`:

```
int i = 10;
```

To declare a new variable that is a pointer to an `int` an `*` is added, for example declare a new pointer `p`:

```
int *p;
```

To assign this new pointer to "point to" the `i` variable declared previously the `&` operator is used, which tells us the address where the variable is stored.

```
p = &i;
```

Now `i` continues to be a variable, and `p` is another variable (a pointer) that points to `i`. To get the value pointed by `p` use the `*`, this is usually call dereferencing:

```
printf("%d", i);   // prints 10
printf("%d", *p);  // prints 10
```

Pointers can be passed as function arguments, for example given the following prototype for the `swap`function:

```
void swap(int x, int y);
```

The goal of this function is to swap the value of two variables, the problem is that in this case the values for `x` and `y` are being passed by value, which means that even they are swapped inside the function, the variables from where the function were called will remain the same. But, we can change the function to receive two pointers as arguments:

```
void swap(int *px, int *py);
```

Now we are passing the arguments by reference, meaning that what we are use inside the function are not a copy of the values from the caller, but a reference to the values themselves (references), meaning that if the values are swapped inside the function, they will remain swapper when the function returns to the caller. Remember that now we need to pass the pointers when calling the function:

```
int a = 10, b = 20;
swap(&a, &b);
printf("%d, %d", a, b);  // prints: 20, 10
```

Array subscripting can also be achieved using pointers. For example, declaring a array `a` of `10` elements of type `int`:

```
int a[10];
```

Basically this is a block of ten consecutive objects of type `int`. If we declare `pa` as a pointer to an integer:

```
int *pa;
```

After the following assignment:

```
pa = &a[0];
```

the pointer is "pointing at" the first element of the array `a`. For example to print the first element of the arrawy pointed by `pa` we can use:

```
printf("%d", *pa);
```

We are using the `*` to *dereference* the pointer which gives us the value pointed at, i.e. the value stored in the first element of the array. So, if `pa` points to the first element of the array, `pa+1` points to the second, `pa+2` points to the third, more generically `pa+i` points to `i` elements after the element `pa` is pointing at.

This is true regardless of the type of variable. For example the following `for` loop counts the length of a string pointer `s`:

```
for (count = 0; *s != '\0'; s++)
  count++;
```

Because a string is an array of characters and `s` is of type `char *` (a pointer), the `*s` is used to get the character element in the position we are, and `s` (the pointer) is incremented by 1 in each iteration, i.e. point to the next element of the array (the next character of the string).

Files

To read or write from a file, the first step is to open the file. To do this first we declare new pointer of type `FILE` (types and functions described here are provided by the `stdio.h`):

```
FILE *fp;
```

And the open the file using the `fopen` function:

```
fp = fopen("input.dat", "r");
```

the first argument to the `fopen` function is a string with the name of the file to open, and the second argument is also a string defining the *mode*: read ("r"), write ("w"), or append ("a"). After the file is used, the `fclose` function is used to close the connection to the file.

```
fclose(fp);
```

The `fread` and `fwrite` functions, are used to read and write binary data from and to files. For example to write the `point` *struct* defined in a previous section to the file opened in `fp`:

```
fwrite(&p1, sizeof(struct point), 1, fp);
```

The first argument is a pointer to the variable, so in this example the `&` is required to get the memory address of the variable; the second argument is the size of the *struct* that is computed using the `sizeof` function; the third argument is the number of elements to write; and the last argument is the file pointer. To read back the data from file, and store it back in memory using the `fread` function:

```
fread(&p1, sizeof(struct point), 1, fp);
```

In particular single characters can be read from a file pointer using the `getc` function, and written using the `putc` functions:

```
c = getc(fp);
putc(c, fp);
```

And complete strings using the `fscanf` and `fprintf` functions, which behave as their `scanf` and `printf` counterparts but take an additional argument: the file pointer to scan from or print, for example:

```
fscanf(fp, "%s", str);
fprintf(fp, "%s", str);
```

Full lines of input can be read from a file using the `fgets` function, and written using the `fputs` function, for example:

```
fgets(line, MAX, fp);
fputs(line, fp);
```

Both functions take as one argument the character array pointer where to set/get the line and the file pointer. `fgets` takes an extra argument: the maximum number of characters that it will read.

References

- Brian W. Kernighan, and Dennis M. Ritchie. The C Programming Language (2nd ed.). Prentice Hall. 1988.

Disclaimer

This document is just a collection of notes, topics coverage is **not** complete, and many details were omitted on purpose if favor of simplicity. For complete details on the C programming language read the references.