Chapter 2 Notes

September 15, 2017

So here are Notes from the second chapter. I would say it's more about how language elements work, that are inside functions.

1 Variable Names

Variable names are case sensitive, cannot be equal to reserved words and it has several other limitaions. So if the usual guidelines for naming variables are followed, the should be no complaint from the compiler.

2 Data Types ans Sizes

C only likes numbers. Everything is a number in C. And if it's not a number, it should be converted to a number or a series of numbers. Number can be small or big, signed or unsigned, with or without floating point.

The purpose of C is to make code as fast as possible, for this reason the `int` is usually made to match the usual size for integer operations on the architecture. For this reason it can be of different size on different architectures. In order to make the way programs work more predictable I usually use integer types from `<stdint.h>`.

3 Constants

Constants usually should be outside the code. Several types of constants exist.

The `#define` constants are defined before the compilation.

The `enum` constants are defined during compilation. Their advantage is that the compiler assignes them values, and if a value has to be inserted between two `enum` constants, the user doesn't have to check that there are no constants with the same value.

4 Declarations

Variables are required to be declared in C. It's not like in dash where any variable can be used without being declared or in Lisp, where a name can be used if it's

not accessed.

Variables can be declared and initialized. External and static varialbes are initialized to zero if they are not explicitly initialized in the code. The stack variables are not initialized because functions would become slower (of course unused assignments would be optimized out, but anyways it's how it was decided and how it is and it makes somehow sense).

5 Arithmetic Operations

There exist built in arithmetic operators in C. The are what would be necessary for OS development: `+`, `-`, `*`, `/`, `%`. Two more operators are unary: `+` and `-`.

6 Relational and Logical Operators

Relational and Logical operators are not very complicated. They are similar to what we have in many other languages, like Java for example.

7 Type Conversions

Type conversion is when an interpretation of the contents of a variable is assigned to another variable. Because C likes numbers and dislike strings, `double` and `float` numbers can be converted to in, while when we try to convert chars and strings, we only get their internal representation. C considers chars as being numbers and strings as pointers.

I think it should be possible to convert a function to an array of chars. So let's test it.

```
#include <stdio.h>

void dothings()
{
  printf("doing things\n");
}

int main()
{
  unsigned char *pch;
  pch = (void (*))dothings;
  for (int i = 0; i < 10; i++)
    printf("%02x ", pch[i]);
  printf("\n");
}
```

If we compile it with

```
gcc -O0 -g f2a.c -lm -o f2a
```

Then if we look at the output of `objdump -d f2a`, we see, that we indeed successfully converted a function to an array of `char`.

8 Increment and Decrement Operators

To increment and decrement, the usual operators are used. It should be remembered, that when the parser reads code, it tries to read as much as possible. Therefore `++a` is `++a` and not `+(+a)` because when the parser sees `+` it knows that is can not only be a `+` operator, but also the first character of `++` and `+=`.

9 Bitwise Operators

C has bitwise operators. And because C has signed and unsigned integer varialbes, it is taken into account for the right shift. The result is that for both signed and unsigned integers, a right shift gives the same thing an entire division by 2 would to: it duplicates the last bit.

10 Assignment Operators and Expressions

Assignment operators are also useful. Their purpose is to modify variables, which is a special feature of C. In many languages, like Mercury and Oz, it is very difficult or even impossible to modify a variable. But C is different, it uses a completely special paradigm, which makes it more similar to machine code and assembly.

11 Conditional Expressions

This section describes the ternary operator. It is of very low precedence, so it usually doesn't need parentheses (only if there are assignment operators inside).

12 Precedence and Order of Evaluation

Because not everything is isolated in C (like in Lisp), there are special rules of precedence of operators.

The lowest are the assignment and the ternary operator.

The highest are the unary operators.

For the rest the order is (from highest to lowest):

- arithmetic
- shift

- integer comparison
- integer equality
- logical
- boolean

It's a lot to remember that's why I usually use short expressions and when not sure, parentheses.

The weirdest and the hardest to understand is the lowest precedence comma operator. In order to try to somehow understand it, I wrote a little test program:

```
int main()
{
  int a;
  int b;
  int c = (a = 1, b = 10);
  printf("%d\n", c);
}
```

It prints 10. Don't really know why, nor how it works. Perhaps it's something like `progn` in Common Lisp? Not sure that we will have more info about it in this book.