2. Basic constructs

Ján Dugáček

October 13, 2017

Table of Contents

An example of C code

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) {
        if (argc < 2) {
                printf("Needs an argument\n");
                return 1;
        }
        int num = 1;
        int arg = atoi(argv[1]);
        while (arg > 1) {
                num = num * arg;
                arg = arg - 1;
        }
        printf("Result %i\n", num);
        return 0;
}
```

You don’t need to know much to do anything

- The elements of the previous example are enough to solve almost any task
- Of course, most of the language is exists to make programming more practical
- C is a minimalistic language itself, containing only 32 keywords
- C++ in contrast embraces large quantities of functions for most common purposes
- We’ll start with C and continue with the essential parts of C++

Variables

- *Everything* in digital format is a number or a group of numbers (addresses, texts, pictures, programs, ...)
- There are several formats for numbers, depending on the required size and need to support negative numbers and decimals
- Numbers are always binary code, groups of ones and zeroes, a bit is a single value that can be zero or one, a byte is a group of eight bits ($8^2 = 256$ possible values)
- On computers, numbers usually can be saved on 1 byte (256 values), 2 bytes ($2^{16} = 65536$ values), 4 bytes ($2^{32} = 4294967296$ values) or 8 bytes ($2^{64} = 18446744073709551616 = 1.8 \cdot 10^{19}$ values)
- A number stored someplace with a name is called *variable*
- A single number is called *primitive data type*

Available types

- `int` - standard sized integer (usually `int32_t`, range -2147483648 to 2147483647)
- `short int` - short sized integer (usually `int16_t`, range -32768 to 32767)
- `char` - very short sized integer, often used to store letters (usually `int8_t`, range -128 to 127)
- `long int` - short sized integer (usually `int64_t`, range -9223372036854775808 to 9223372036854775807)
- `unsigned int` - integer for non-negative values (usually `uint32_t`, range 0 to 4294967295)
- There are unsigned versions of all other sized integer types

Available types

- `float` - stores numbers with decimal point (usually 32-bit, 6 decimals, greatest numbers are around 10^{38})
- `double` - stores numbers with decimal point (usually 64-bit, 15 decimals, greatest numbers are around 10^{308})
- Processors usually support also `long double` that is 80 bits large
- Video cards support also a 16 bit float
- Stored using IEEE754 (see: `https://www.h-schmidt.net/FloatConverter/IEEE754.html`)

Advanced exercise

- Do this only if you already know how to use variables!
- Calculate π using the Monte Carlo method (scatter many points randomly in a square, calculate the fraction of them that is closer to its centre than a half of the square's side)
- Hint: you may use `rand()` to generate random numbers
- Why is the result so imprecise?
- Challenge: Do it without computing any square root (neither manually nor in the program)
- Second powers of the same numbers are computed over and over. Would it be useful to store the computed second powers of numbers for later use?

Usage

```
#include <stdio.h>
int main(int argc, char** argv) {
        int x;
        x = 2;
        int y = 4;
        y = y * (x + 4);
        printf("Computed %i\n", y);
        return 0;
}
```

- We first create variable `x`
- Then we set value 2 to `x`
- After, we create variable `y` and immediately set its value to 4
- In the next step, `x` and 4 are summed, the result is multiplied by `y` and set as a new value to `y`
- The resulting value of `y` is written into the terminal with a nice introduction

Usage #2

```
#include <stdio.h>
int main(int argc, char** argv) {
        int x = -1024 - 2;
        short int y = x * x;
        int z = x / 4;
        printf("Computed %i and %i\n", y, z);
        return 0;
}
```

- We first create variable x and save -1026 into it
- Then we create variable y and save the square of x into it, which does not fit there
- After, we create variable z and set its value to x divided by 4, because both x and 4 are integers, the result is an integer, rounding the value down
- The resulting values of y and z are written into the terminal

Usage #3

```
#include <stdio.h>
int main(int argc, char** argv) {
        float x = 15 / 2;
        float y = 15.0 / 2;
        float z = (float)15 / 2;
        float w = x / 2;
        printf("Computed x=%f y=%f z=%f w=%f\n", x, y, z, w);
        return 0;
}
```

- We first divide 15 by 2, rounding down because both numbers are integers and result is integer, recalculate it to float and save it into `x`
- Then we divide 15.0 by 2, because 15.0 is a decimal, it is a float, arithmetic between a float and an int yields a float, the resulting float is saved into `y`
- After, we convert the integer 15 to float, divide it by 2, the resulting float is saved into `z`
- Next, we divide the float `x` by 2 and save it into variable `w`
- The resulting values of variables are written into the terminal

Exercises

1. Set 17 to `x`, divide it by 4 (rounded down), set $x^2 - 12$ to `y`, add 18 to the result and write the result as `Result is 22`
2. Calculate $(3+2-12)\cdot((9-2)\cdot\ 5)+(3+2-12)\cdot(8+((9-2)\cdot\ 5))$ without writing `3 + 2 - 12` or `(9 - 2) · 5` more than once or calculating anything yourself
3. Calculate $\frac{3+2-12}{(9-2)\cdot\ 5}+(3+2-12)\cdot(8+\frac{(9-2)\cdot\ 5}{3+2-12})$ without writing `3 + 2 - 12` or `(9 - 2) · 5` more than once or calculating anything yourself

Shortcuts

- Lines like `x = x + 4` are used a lot, so they can be shortened to `x += 4`
- Analogically, you can use `x -= y * 2` (subtract 2 multiplied by y from x and save it into x), `x /= 1.5` or `x *= 1.01`
- `x += 1` can be further shortened to `x++` or `++x`
- Analogically, there is also `x--` or `--x` for `x -= 1`

Advanced exercise

- Do this only if you already know how to use `if`, `while` and `for`!
- Calculate x in $x + 1 = \frac{1}{x}$
- You may assume that x is positive
- Challenge: Do not calculate anything more than 1000 times, but limit your precision only by the maximum decimals that can be stored in primitive types and use no prior knowledge

Condition

```
if (x < 0)
        x *= -1;
```

- Here, we check if `x` is lesser than 0
- Only if `x` is lesser than 0, multiply by -1
- This will replace `x` by its absolute

Condition #2

```
int changed = 0;
if (x >= 0) {
        x *= -1;
        changed = 1;
}
```

- Here, we check if `x` is greater than or equal to 0
- If the condition is met, multiply `x` by -1 and set variable `changed` to 1
- Variables defined in a *block* (the part in curly brackets) are not available outside of it

Condition #3

```
int changed = 0;
int equals = (x == y);
if (x > y || x < 2 * y) {
        changed = 1;
        if (equals) {
                changed = 2;
        }
}
```

- Here, we check if `x` is greater than `y` *or* `x` is less than two times `y`
- We also check if `x` equals `y` and save the result of the comparison into variable `equals`
- If the first condition is met, 1 is assigned to `changed` and we check if `x` was *previously found to be* equal to `y`
- The result of comparison can be 1 (true) or 0 (false)

Condition #4

```
int z = 0;
if (x > y && x != 1) {
        z = 1;
        if (x = y - 1) {
                z = 2;
        }
}
```

- Here, we check if `x` is greater than `y` *and* `x` is not equal to 1
- If the condition is met, 1 is assigned to `z` and `y - 1` is assigned to `x` and *if x is non-zero (true)*, 2 is assigned to `z`
- **Do not confuse = (variable assignment) with == (comparison)! It is a huge source of errors!**

Condition #5

```
int z = 0;
if (x > y && (x = y || y == 1)) {
        z = 1;
}
```

- Here, we check if `x` is greater than `y` *and if that is true*, we assign `y` into `x` and if the result is non-zero (true) or `y` is equal to 1, the condition is met
- If the condition is met, 1 is assigned to `z`
- If `x` is not greater than `y`, the condition is never true and the rest is ignored, thus `y` is never assigned to `x`
- Do not confuse `&&` and `||` with `&` and `|`, they mean something else but usually lead to different outcomes, so a program using `&` instead of `&&` may seem okay but then behave weirdly

Condition #6

```
int z = 0;
if (!(x > y) && (x == 1 || (x = y))) {
        z = 1;
}
```

- Here, we check if it's not true that `x` is greater than `y` *and if that condition is met*, we check if `x` is equal to one, *if that is false*, we assign `y` into `x`, check if it's non-zero and go inside the block if the one of these two conditions is met
- If `x` is equal to 1, the condition is true regardless of the value of `y` and the next condition is ignored, thus `y` is never assigned to `x`

Inline condition

```
int z = (!(x > y) && (x == 1 || (x = y))) ? 1 : 0;
```

- This does the same as the previous, if the condition is met, z is initialised with 1, otherwise it's initialised with 0
- It is useful only when assigning values into a variable depending on a condition

```
int z = (x > 1) ? ((y > 1) ? 2 : 1) : 0;
```

- It can be nested too
- If x is greater than 1, then if y is greater than 1, 2 is set into z, otherwise 1, if x is not greater than 1, 0 is set into z

While loop

```
unsigned long int y = 1;
while (x > 1) {
        y *= x;
        x--;
}
```

- `while` loop is much like `if`, but at the end of the block, it checks for the condition again and if it's still true, the block is executed again, after it is checked again and so it goes forever while the condition is true
- This particular loop computes the factorial of `x` (the result is in variable `y`)

While loop #2

```
unsigned long int y = 1;
while (x > 1) {
        y *= x;
        if (y > 1000000000000000) break;
        x--;
}
```

- This is like the previous one, but we check if `y` is greater than a billiard, if it is, the loop is *broken*, immediately interrupted and the execution jumps out of it
- A lesser variant of `break` is `continue`, which skips the rest of the block but goes to check for the condition again
- If there is nothing to stop the cycle from looping, the program will freeze

```
while (1) {}
```

While loop #3

```
unsigned long int y = 1;
do {
        y *= x;
        x--;
} while (x > 1);
```

- This is a variation of the `while` loop that starts executing the block and checks the condition if it's going to continue after
- The only difference is that the block is executed at least once regardless of the condition
- The code above *does not* correctly compute the factorial of `x`, if `x` is 0, the result is 0, which is incorrect because $0! = 1$

For loop

```
int x = 0;
int i = 0;
while (i < 10) {
        x += i;
        i++;
}
```

- Defining a variable, changing it and checking it in a loop is so common that a new construct was created to shorten it:

```
int x = 0;
for (int i = 0; i < 10; i++) {
        x += i;
}
```

- This does the same as the code above

For loop #2

```
int x = 0;
for (int i = 0; i < 10; i++) {
        if (i % 2 == 1) continue;
        x += i;
}
```

- The first part defines a variable (but may also assign a value to an existing variable or may be left blank), the second part is a condition that must be met if it's to be repeated and the last part is a change that happens after every loop (the condition is checked right afterwards, so in the case above, it never enters the block with `i` equal to 10)
- The % sign is modulo, the result of `i % 2` is the remainder of division of `i` by 2
- `continue` makes the loop restart, but `i` is increased before the check (unlike in `while`)
- The code above sums all even numbers between 0 and 10 (including 0, excluding 10)
- The variable change in each iteration of the loop is called *iterator* and usually named `i` (if there are more of them, they are named `i`, `j`, `k`, ...)

Exercise

1. Calculate the product of odd numbers between 23 and 37
2. Calculate $\int_{-2}^{4} sin(x^2)dx$
3. Calculate π by computing the ratio between the integral of a constant and the integral of the circle function $y = \pm\sqrt{1 - x^2}$

- Hint:

```
#include <math.h>
int main(int argc, char** argv) {
        float x = 1;
        float y = sin(x);
        float z = sqrt(x);
```

- You must add `-lmath` to the compiler arguments or it won't run)

Advanced exercise

- Find primes lesser than 10000 using Eratostenes' sieve
- Is it faster than trying to divide each number by all primes lesser than its square root?

Arrays

```
int array[10];
for (int i = 0; i < 10; i++) {
        array[i] = 20;
}
int otherArray[] = {3, 1, 8, 27, 2412412};
```

- First, we create an array of 10 elements
- Then we iterate through these 10 elements and set the value of each to 20
- `continue` makes the loop restart, but `i` is increased before the check (unlike in `while`)
- In the end, we create and initialise an array inline, its size is determined by the number of elements in the initialiser
- **Elements are indexed from zero!**

Problems with arrays

```
int array[10];
int afterArray = 42;
array[10] = 13;
```

- The size of an array cannot be increased (it's a limitation of the way computers work)
- The size of an array cannot depend on a variable in C++ if compiled with Microsoft Visual Studio
- There are tricks around these issues
- Because elements are indexed from zero, the last line writes at the 11th element of a 10 element array
- Array boundaries are not checked, so writing there overwrites the variable behind the array

Arrays of higher dimension

```
int array[10][10];
array[8][3] = 5;
```

- Arrays can be of any dimension
- In memory, they are written as unidimensional arrays, the location is calculated from the coordinates
- They cannot be used as function arguments

Strings

- A string is an array of variables of type `char`

```
#include <stdlib.h>
int main(int argc, char** argv) {
        char text[] = "123";
        int num = atoi(text);
```

- Because the size of an array cannot be learned, every string must end with a 0 (the 0 sign does *not* have numeric value 0)
- Assigning a string into an integer assigns its address in memory
- String can be converted into an integer with `atoi` (`atof` for float), if `stdlib.h` is included
- If the array is defined elsewhere, the type is `char*`

Exercise

1. Create an array of first 10 odd numbers and print them afterwards
2. Create a pair of vectors of size 100, set some values to them and compute their dot product
3. Create a pair of 2D arrays of size 100x100, set some values into them and compute their matrix product

Calling functions

```
char text[] = "123";
int num = atoi(text);
printf("Converted %s to %i", text, num);
```

- A function call consists of the function's name and a list of comma-separated brackets in regular brackets
- We have already seen the usage of several functions
- Functions usually have fixed number of arguments with static types, but exceptions are common (like `printf`)
- `printf` deduces its arguments from the formatting string - `%s` is substituted by a string, `%i` by an integer, `%f` by a float, the order must be the same
- A function can call itself recursively

Defining functions

```
float square(float num) {
        num = num * num;
        return num;
}
int main(int argc, char** argv) {
        float a = 3.5;
        float b = square(a);
```

- A function needs to be defined with a return type followed by its name and the comma-separated argument list in brackets
- Its arguments are copied into the variables created when the function is called, so the variables used in the call are not changed from inside the function
- Function ends when `return` is called, returning the value after the keyword as the function's result
- If a function is not meant to return anything, the return type is `void` and nothing follows the `return` keyword

const

```
float square(const float num) {
        const result = num * num;
        return result;
}
```

- If a variable is declared as `const`, it cannot be changed (unless the program is writing at an incorrect address)
- It's useful to make sure that some variables are not changed when they are assumed to be the same all the time they are defined
- It's recommended to write `const` before every variable that does not need to be changed (it's called *const correctness*)
- It allows the compiler to make additional assumptions about the code and optimise it better
- Note: prefixes to variable types like this are called *modifiers*

The main function

```
int main(int argc, char** argv) {
        printf("Hello world!");
        return 0;
}
```

- It is the function called when the program starts
- Its arguments are the space-separated command line arguments given by the user
- Its return value is the program's return value, it should be 0 if the program finished correctly or something else if it failed; the number may indicate the reason of failure (in command lines, counterintuitively, 0 is true and anything else is false)

Command line arguments

```
#include <stdlib.h>
int main(int argc, char** argv) {
        int num = atoi(argv[1]);
```

- argv is an array of strings that were written as command line arguments
- If the program is named prog, calling it as ./prog 12 will make the program start with ./prog in argv[0] and 12 in argv[1]
- In this case, we convert argv[1] from string to a number and set it into variable num
- argc contains the size of the argv array

Exercise

1. Create a function that computes the third power of its argument
2. Create a function that powers the first argument by the second argument, using float multiplication or division if the exponent is -1, 0, 1, 2, 3 or 4

- Hint:

```
#include <math.h>
int main(int argc, char** argv) {
        float x = 2;
        float eleventhPowerX = pow(x, 11);
```

- You must add `-lm` to the compiler arguments or it won't run)

Homework

- Find the values of a, b and c where $f(x) = a \cdot \ln(x \cdot b) - b \cdot \sqrt{200 - x \cdot c} + c$ has the smallest sum of second powers of values of $f(1)$, $f(2)$, ... $f(100)$
- Advanced homework: Create a calculator program that reads a formula with plus, minus, multiplication and division from command line arguments, assuming that all operations are brackets to avoid dealing with operator priority (you may want to assume that all numbers, signs and brackets are separated by breaks into different command line arguments or written all as one argument)