First round tasks as a `ZIP file.`

Data types and Variables¶

Computer programs are stored in memory in binary format. The program usually consist of roughly **1) code**, which contains the instructions for the computer a processor to execute the program, and **2) data**, which is stored and used during program execution.

In C (and many other programming languages) data is stored in variables. Each variable has a name and a data type that determines what values can be stored in the variable. C applies static type checking, meaning that compliance to declared data types is checked already at the compile time. Therefore, before a variable can be used, it must be declared with an indication of its data type.

The variable names in C are case-sensitive. The name can consists of alphabetic characters, numbers and underline (_), but must not start with number. These naming rules apply all different types of names in C: functions, data types, and so on. Below you can find an example of the declaration of the variable "number". The variable uses the *int* data type that represents an integer.

`int number;`

Another important feature of C language variable declaration is: local variables are always uninitialized unless it is explicitly initialized in code. Above declared variable can take any value (as it is not initialized).

Different data types are discussed more in the following chapters. Below you can find the most commonly used data types and their properties.

../../_images/data_types-2.svg

Integer data types¶

Integers are perhaps the most common data type in C (although this depends on the application area). There are different types of integers in C, differing in the amount of memory space they require, and consequently the number range they can represent. The integer data types are the following:

- **char** – size 8 bits (1 byte), signed values from -127 to 127, unsigned values from 0 to 255.
- **short int** – 16 bits (2 bytes), signed values from -32767 to 32767, unsigned values from 0 to 65535
- **int** – at least 16 bits, typically 32 bits (4 bytes), signed values from $-(2^{31} - 1)$ to $2^{31} - 1$, unsigned values from 0 to $2^{32} - 1$.
- **long int** – at least 32 bits, can be 64 bits (8 bytes)
- **long long int** – 64 bits, signed values from $-(2^{63} - 1)$ to $2^{63} - 1$, unsigned values from 0 to $2^{64} - 1$.

For each basic data type, a declaration can either contain signed and unsigned keywords (before the actual type), to specify whether the data type is intended for only positive values, or also for negative values. If this is not specified, the default is to apply signed type, except in the case of char, where the default behavior is implementation dependent. Unfortunately, the basic integer types do not always have the same range, and for example in the old implementations the size of int type can be shorter than in modern implementations.

For long int and short int a shorter form of long and short can be (and are usually) used.

Below are examples of few variable declarations. When a variable is declared, it can be set to have an initial value, or it can be left uninitialized. **If a variable is not initialized, its initial value is unknown**, and results in unpredictable program behavior. Therefore it is recommended to initialize the variable when declaring it, when possible.

```c
1  #include <stdio.h>
2
3  int main(void)
4  {
5      char varA = -50;
6      unsigned char varB = 200;
7      unsigned char varB2 = 500;  // Error, exceeds the value range
8      int varC;  // ok, but initial value is unknown
9      long varD = 100000;
10
11     /* Output the above values */
12     printf("%d  %u  %u  %d  %ld\n", varA, varB, varB2, varC, varD);
13
14     return 0;
15 }
```

In the above example, another way of commenting on the program is displayed. When the program is in line with two backslash (//), all the text after them is interpreted as a comment until the end of the line. The next line is interpreted as back to normal C code. Since C is a liberal in terms of design, this type of comment can be added after the C-language program line.

In the line 7 of the above program, a value greater than 8 binary bits is stored in unsigned char - type variable. Compiler produces a warning about this overflow but still it compiles and produces a executable. Program produces incorrect output as value is stored in incorrect variable. The C compiler will do its best to produce an executable program, which is often treacherous, as in this case, the program is clearly wrong and will not function correctly. **Compiler produced warnings must therefore always be taken seriously and fixed to get expected behaviour.**

Task 1.2: Fix Numbers¶

Please correct the above program so that it works as originally expected, printing the following values.

```
-50  200  500  0  100000
```

Points 10 / 10 My submissions 1

Deadline Tuesday, 8 June 2021, 19:59
To be submitted alone

This course has been archived (Saturday, 31 December 2022, 20:00).

Fix Numbers

Select your files for grading

main.c

Choose File No file chosen

Submit

Information: One problem with the basic data types presented above is that the exact size of the variable can differ between architectures. The C99 standard also includes new, fixed size integer definitions that improve the portability of programs between architectures. These are defined in the **stdint.h** header, and are as follows:

- **uint8_t, int8_t**: unsigned and signed 8-bit integer.
- **uint16_t, int16_t**: unsigned and signed 16-bit integer.
- **uint32_t, int32_t**: unsigned and signed 32-bit integer.
- **uint64_t, int64_t**: unsigned and signed 64-bit integer.

Constants are fixed values defined by the programmer when writing the code. By default integer constants assume int data type, i.e., they can represent a 32-bit value range in modern systems. Above we saw constants -50, 200, 500 and 100000. By default constants follow the decimal (base 10) number system, but there is a representation format for giving octal (base 8) constants, and for giving hexadecimal (base 16) constants. Octal constants start with digit 0. Hexadecimal constants are prefixed with `0x`. In the early parts of the course operating with decimal numbers is sufficient, but later we take a closer look at hexadecimal notation.

Below are examples of each of these notations.

```c
short a = 012; /* set variable a to octal 012, equal to decimal 10 */
short b = -34; /* just using decimal number here */
short c = 0xffff; /* hexadecimal constant, equal to decimal 65535 */
```

It is essential to recognize that different notations are simply different forms of presentation by same numbers. For example, 0xff represents the same numerical value as the 255 or 0377.

If constant for *long* data type is needed, 'L' needs to be added to the end of the number: `long la = 10000000000L;`.

Even though C is statically typed, the types are not strictly enforced, and the type of a value is implicitly converted when, for example, assigning integer constant into **char** type variable. For example, assigning value 1000 to **char** variable is possible, but as it is likely incorrect code, the compiler will warn about this. If programmer ignores this warning, the actual value of the variable will become equivalent to the 8 lowest bits of decimal 1000, which is a different number.

Floating point numbers¶

For presenting large numbers, or fractions of integers, floating point data types can be used. Internally, a floating point number is built from three components: the sign bit, significand, and exponent. The number is then composed of these three parts in the following way:

number = $(-1)^{sign}$ * 1.significand * $2^{exponent}$

Because of the way how floating point numbers are stored in binary memory, the floating point numbers cannot cover a continuous number space. Therefore floating point calculation operations do not always give an exactly correct results, but sometimes a value "close" to the correct result comes out. In addition, typically computation with floating point numbers is slower than with integers. Therefore integers are often used in C, and floating point numbers are only used when the integer value range is not sufficient. Additional details can be found in a related Wikipedia-article.

There are three floating point data types, differing in how many bits are allocated to the above three components:

- **float** – 32 bits (1b sign + 23b significand + 8b exponent)
- **double** – 64 bits (1b sign + 52b significand + 11b exponent)
- **long double** – 80 or 128 bits

The constants for floating point numbers can either use the conventional decimal format (e.g., 1.543), or exponent format (1e-2), or combination of both. The default data type for floating point constant is double, but if the constant is suffixed by 'F', it is assumed to be of type float. For example:

```c
float d = 0.534;
double e = 2e10;
float g = 0.111F;
```

String and character constants¶

String constants are included in double quotes, as we saw together with the **printf** call in the first example. Operating with strings in C requires understanding arrays and pointers, and therefore we defer that to Module 2 for now. The characters shown to user in C follow the ASCII encoding scheme, There are also various other encoding schemes, but the common property in all of them is that each visible character has a numeric character code. For example, in ASCII, letter 'A' is stored similarly as decimal number 65 in the system memory, but shown as 'A' when printed to the screen as character. C supports character constants to make it easier to operate with ASCII-encoded characters. Character constants are included in apostrophes ('):

`int char_A = 'A';`

It is important to make distinction between string constants ("text") and character constants ('t'), because they stand for different data type. The character constants are of **int** type, similar to normal integer constants, and strings are arrays of char variables (as will be discussed in further sections)

Note that constants '1' and 1 are different in C: '1' is same as decimal number 49 according to the ASCII system, whereas 1 is just decimal number 1, but is nothing relevant interpreted as ASCII. Both are integers nonetheless.

Arithmetic operators¶

Above we have seen cases of **assignment operator** (=) when initializing the variables together with declaration. Assignment can also be done separately of the declaration, and an earlier used variable can be re-assigned – after the following three lines, variable **var** has value 20:

```c
int var;  // Value is unknown as it is not initialized
var = 10;   // Now value is known
var = 20;
int varB = var = 20 + 10;  // Value of both varB and var is 30
```

Operators **+ (plus), - (minus), * (multiply), / (divide) and % (modulus)** are used as normal mathematical operators. As customary, + and - have lower precedence than *, / and % (i.e., the latter are evaluated first, regardless of their position in the expression). The modulus operator can only be applied for integers, but the others work for both floating point numbers and integers. Parenthesis can be used to control the precedence (order of computation) as taught in school.

Here are a few examples:

```c
float fa = 5.0 / 2; /* '5.0' indicates the use of floating-point constants */
int ia = 5 / 2; /* Result is different from above as result is stored in integer */
char cb = 3 * (1 + 2);
long lc = cb * fa;

int a = 0;
b = a++;  // Value of a is 1, value of b is 0
b = a--;  // Value of a is again 0, value of b is 1
b = ++a;  // Value of a is 1, value of b is also 1
b = --a;  // Value of a is 0, value of b is also 0
a += 5;   // Value of a is 5
a -= 3;   // Value of a is 2
```

The above example also shows that multiple operators and expressions can be used to form a single statement – here together with variable declaration and its initialization.

C provides an alternative unary way for incrementing or decrementing the value of a variable by one, by using increment and decrement operators. These operators take either postfix or prefix form. In postfix form, a++ increments value of a by one, and a– decrements the value of a by one. In prefix form, these operators are ++a and –a. The functionality in postfix and prefix formats is not completely equivalent: in postfix form the value of the expression is evaluated before the adjustment to the variable takes place, but in prefix form the value is evaluated after the adjustment. This can have significance when the unary increment/decrement operator is used as part of a longer expression.

Another alternative is to use assignment operators, such as a += 2 which is equivalent a = a + 2. The assignment operator formant works for all above arithmetic operators.

```c
#include <stdio.h>

int main(void)
{
    int varA; /* Value is unspecified now */
    varA = 10; /* value is set to 10 */
    varA++; /* value is 11 */
    varA *= 2; /* value is 22 */
    printf("Final value of varA: %d\n", varA);

    return 0;
}
```

Type conversions¶

Because C can do implicit type conversions between variables, there can be multiple data types as part of single expression. When larger data type is converted into a smaller one, the excess high-order bits are dropped, and therefore the value may change. When float is assigned into integer, the decimals will be truncated.

Conversions can be forced explicitly using a **type cast** by including the intended data type before an expression in parenthesis. Sometimes this can affect the outcome of the expression, as happens in the following example:

```c
float f = 1.5;
int a = f + f;
int b = (int) f + (int) f;
```

The above program causes the value of **a** to be 3, while value of **b** is 2. The first assignment to variable **a** calculates 1.5 + 1.5 = 3 (as floating point number), which is automatically converted to integer as part of assignment operation to **a**. In the second case (assignment to **b**), the value of **f** is first converted to integer on both sides of the plus operation, which causes its value to change from 1.5 to 1. After this the result becomes 2. Use of type casts is normally not necessary in simple programs, but sometimes are unavoidable.