Contents

1 Variables

A variable is a piece of computer memory that we can write data to and read from. We can use variables to store values such as numbers. All variables must have a data type.

1.1 Declaring variables

To create a variable, you need to **declare** one, which is to state its existence. This is done in a code statement containing the variable's **data type**, a space, and then the variable's **identifier** or **name**:

```
int x;
```

The above statement will declare a variable of data type `int` named `x`. The `int` data type is used to store integers. Therefore, we can store integers values in `x`.

When variables are declared, their initial value will be a **garbage value**, which is a completely random value. This is because C will assign the memory location that your variable's value is stored in to someplace random. This specific place might have been used by another program on your computer to store one of its variables, making the garbage value you recieve nearly unpredictable. Situations with unpredictable outcomes like this are known as **undefined behavior**.

Unless you want a pseudo-random value to use in your program, make sure you assign your variables your own value before using them.

1.2 Modifying variables

The most basic way we can modify a variable is to use the assignment operator, `=`.

```
int x;
x = 5;
```

The left operand of the assignment operator is a variable, and the right operand is an **expression**, which is a combination of operands and operators that produce a resulting value. Assignment will store the value in the right operand in the variable in the left operand.

In our case, our expression is just a value, `5`, which is a **literal** of type `int`. A literal value is a raw value written into our source code. Like variables, literals have a data type. Any number literal with no special characters prefixing or suffixing it is a literal of type `int`.

Here are some examples of expressions used with assignment. Note that lines starting with `//` are ignored by the compiler. These are known as **comments**. You can and should use them to document your code.

```
int x;

// The + operator is for addition
x = 2 + 3;

// x will now hold the value 5

// Similarly, - is for subtraction
x = x - 2;

// x will now hold the value 3
```

If you want to assign a value to a variable in the same statement that it is declared in, you can do this:

```
int x = 5;
```

This is called **initialization**.

1.3 How variables are stored

As stated previously, variables are stored in computer memory. Computer memory consists of a long sequence of 1's and 0's. Each space where a 1 or 0 can be stored is a **bit**. On nearly every modern machine, 8 of these bits constitute one **byte**. The space that variables take up in memory is always a whole number of bytes, regardless of the size of a byte on the machine that C is running on.

At their core, variables only store different combinations of 1's and 0's in the bytes that they take up. These different combinations are only given meaning by C and our code's interpretation of them.

1.4 Data types

C interprets the contents of a variable's bytes based on the variable's **data type**. C has multiple predefined data types. Some of these types are interpreted in the same way as others but are made distinct by their size.

The size of a data type in bytes is tied to the total number of different values that a variable of that type can hold. The greater the size of the type is, the greater the number of different storeable values is.

1.4.1 Basic type categories

1. Signed integers Signed integer types can hold positive and negative whole numbers. They are called "signed" because they contain a sign bit which determines whether the value they hold is negative or positive. The types that fit into this category, ordered from shortest to longest, are `char`, `short`, `int`, `long` and `long long`. The larger a type of this category is, the greater positive and negative values it can hold:

```
// char is always 1 byte long, so it can hold values from -128 to 127
char number_char = 100;

// int is larger than char, so it can hold values greater than 127
// Assuming that int is 2 bytes long, we can say that it holds values
// from -32768 to 37267
int number_int = 930;

// While the types "long" and "long long" use the same keyword,
// they are distinct types. Obviously, "long long" is the larger
// type of the two
```

```
long number_long = -100;
long long number_long_long = -10000;
```

Most of these types are NOT guaranteed to be the same size on every machine. This is due to the old way that C was designed. The idea was that not restricting the sizes of variables would make C more portable and flexible because it is then up to the compiler to decide how large the types should be according to a machine's specs.

Here is a table showing the relative sizes of these types:

Type	Size in bytes
char	always 1 byte
short	~2
int	2 or 4
long	4 or 8
long long	~8

If you keep adding or subtracting to a variable of one of these types, eventually, you will reach a point where you go past the minimum or maximum value that the type can hold. This will cause an **integer underflow** or **integer overflow**. In C, this causes undefined behavior.

Signed integers are stored in binary. If you don't know what binary is, see here.

2. Unsigned integers Unsigned integers are just like signed integers but they lack a sign bit. Instead of holding negative and positive values, unsigned integers can only hold 0 or positive numbers.

   The types that fit into this category have the same names as the signed integer types, only prefixed with "`unsigned =". Ordered from shortest to longest, they are =unsigned char`, `unsigned short`, `unsigned int`, `unsigned long` and `unsigned long long`. THe larger a type of this category is, the greater positive values it can hold.

```
// unsigned char is always 1 byte long
// It can hold values from 0 to 255
unsigned char num_unsigned_char = 120;

// Notice how unsigned char is the same size as char
// They can hold the same total number of different values,
```

```
// just with different meaning

unsigned long long num_long = 99999;
```

Just like with signed integers, the sizes of these types aren't the same across all machines. Here is a table showing their relative sizes:

Type	Size in bytes
unsigned char	always 1 byte
unsigned short	~2
unsigned int	2 or 4
unsigned long	4 or 8
unsigned long long	~8

When an unsigned integers underflows or overflows, there is no undefined behavior. Instead, these unsigned integers will "wrap around", meaning they will evaluate to their maximum value if you subtract 1 from 0, and they will evaluate to 0 if you add 1 to their maximum value.

Like signed integers, unsigned integers are also stored in binary, just without the sign bit.

3. Floating point numbers Floating point numbers can hold positive and negative values with decimal places. Due to their design nuances, they are not 100% accurate when used in calculations, however, they can be used to approximate values. There are only two float types: `float` and `double`. `double` is twice the size of `float`:

```
float pi = 3.14f;
double e = 2.71828;
```

You cannot have an `unsigned float` or `unsigned double`.

1.4.2 Characters and ASCII

`char` is an integer type, yet its name suggests that it holds alphabetic characters. Why is that? The reason is that characters and numbers are the same in C. C encodes characters in ASCII, which assigns each character a corresponding character code, which is just a number:

`./asciifull.gif`

Number literals, such as `'A'`, have an equal value to their corresponding character code. A's code is `65`, so the literals `'A'` and `65` have the same value.

1.4.3 The sizeof operator

You can use the `sizeof` operator to get the size of a data type or variable in bytes:

```
#include <stdio.h>

int main(void)
{
        int x = 22;

        // "sizeof(x)" is equal to "sizeof(int)" because x
        // is an int
        // You could also write "sizeof x" or "sizeof int"
        unsigned long size_of_x = sizeof(x);

        // When parentheses are used, sizeof looks like
        // a function, but it is not one. sizeof is
        // built into the compiler.

        // Print out size_of_x
        printf("x is %lu bytes long.\n", size_of_x);
        return 0;
}
```

If you don't know how to print variables, see here.

2 Printing variables

See `./output.org`