C introduction

Variables

Contents

Usage

Declaration:

```
type identifier;
```

Assignment:

```
identifier = value;
```

Definition (all at once):

```
type identifier = value;
```

Example:

```
int number;                    /* declaration */
number = 42;                   /* assignment */
int another_number = 23;       /* definition */
```

Saving lines

Multiple declarations

```
int number, another_number;
```

Multiple Definitions

```
int number = 42, anothernumber = 23;
```

But be careful:

```
int a = 23, b = 23;
```

$\neq$

```
int a, b = 23;
```

$\rightarrow$ Avoid multiple variable definitions at one line!

Integer numbers

- Keywords: *int*, *short*, *long*
- Stored as a binary number with fixed length
- Can be *signed*(default) or *unsigned*
- Actual size of *int*, *short*, *long* depends on architecture

Example (64 Bit):

```
int a;              /* Range: -2.147.483.648 to 2.147.483.647 */
unsigned short b;                          /* Range: 0 to 65.535 */
```

Floating point numbers

- Keywords: *float*, *double*, *long double*
- Stored as specified in *IEEE 754 Standard* TL;DR
- Special values for ∞, $-\infty$, NaN
- Useful for fractions and very large numbers
- Type a decimal point instead of a comma!

Example:

```
float x = 0.125;                /* Precision: 7 to 8 digits */
double y = 111111.111111;       /* Precision: 15 to 16 digits */
```

Characters

- Keyword: *char*
- Can be *signed*(default) or *unsigned*
- Size: 1 Byte (8 Bit) on almost every architecture
- Intended to represent a single character
- Stores its *ASCII* number (e.g. 'A' $\Rightarrow$ 65)

You can define a *char* either by its ASCII number or by its symbol:

```
char a = 65;
char b = 'A';    /* use single quotation marks */
```

Scopes

You begin a block with a '{' and end it with a '}':

```
1 int i = 4;
2 {
3     i = 3;          /* valid */
4     int j = 5;
5 }
6 j = 2;              /* invalid, j is not in scope */
```

- Program area in which an identifier may be used
- Referring to it anywhere else causes compilation errors
- Starts at the line of declaration
- Ends at the end of the block, in which the variable was declared

Shadowing identifiers

When redeclaring identifiers inside a block, they refer to a new variable:

```
1 int i = 3;          /* Defining i with value 3 */
2 {
3     i = 2;          /* "Outside" i is now 2 */
4     int i = 4;      /* New "inside" i */
5     i = 7;          /* Changes "inside" i only */
6 }
7                     /* "Outside" i is still 2 */
```

Valid identifiers

- Consist of English letters (no *ß*, *ä*, *ö*, *ü*), numbers and underscore (_)
- Start with a letter or underscore
- Are case sensitive (*number* differs from *Number*)
- Must not be reserved words (e.g *int*, *return*)

Style:

- Stay in one language (English recommended)
- Decide whether to use *camelCaseIdentifiers* or *snake_case_identifiers*.
- When nesting blocks, indent every inner block by one additional tab!

Speaking identifiers

```c
/* calculate volume of square pyramid */
int a, b, c;
a = 3;
b = 2;
c = (1 / 3) * a * a * b;
```

```c
/* calculate volume of square pyramid */
int length, height, volume;
length = 3;
height = 2;
volume = (1 / 3) * length * length * height;
```

Use speaking identifiers.

Please, use speaking identifiers.[1]

[1] Seriously, use speaking identifiers.

printf() with placeholders

The string you pass to *printf* may contain placeholders:

```
int a = 3;
int b = 5;
float c = 7.4;
printf("a: %d\n", a);
printf("b: %d\nc: %f\n", b, c);
```

Output:

```
a: 3
b: 5
c: 7.4
```

You can insert any amount of placeholders. For each placeholder, you have to pass a value of the corresponding type.

Example placeholders

The placeholder determines how the value is interpreted. To avoid compiler warnings, only use the following combinations:

type	**description**	**type of argument**
%c	single character	char, int (if <= 255)
%d	decimal number	char, int
%u	unsigned decimal number	unsigned char, unsigned int
%X	hexadecimal number	char, int
%ld	long decimal number	long
%f	floating point number	float, double

Variable input

scanf() is another useful function from the standard library.

- Like *printf()*, it is declared in *stdio.h*
- Like *printf()*, it has a format string with placeholders
- You can use it to read values of primitive datatypes from the command line

Example:

```
int i;
scanf("%d", &i);
```

After calling *scanf()*, the program waits for the user to input a value in the command line. After pressing the *return* key, that value is stored in *i*.

Note:

- *scanf()* uses the same placeholders as *printf()*
- You must type an *&* before each variable identifier (more about this later)
- If you read a number (using %d, %u etc.), interpretation
  - Starts at first digit
  - Ends before last non digit character
- If you use %c, the first character of the user input is interpreted (this may be a ' ' as well!)

Never trust the user: they may enter a blank line while you expect a number, which means your input variable is still undefined!

Let's calc

Each expression (variables or values) you type in gets evaluated.
You can use operators to combine existing expressions to new ones.

- +, -, *, / as all of you should know
- % is the modulo (remainder) operator
- *, /, % get evaluated before +, -
- Operations in ( ) are of higher precedence

```
1 int a = 5;        /* value 5 */
2 int b = a + 3;    /* value 8 */
3 char c = 120;     /* 120 */
4 char d = 2 * c;   /* why is that not 240? */
```

You see, it's not that easy

- Variables may overflow
- You shall not divide by zero
- Integer division differs from floating point division
- You can use operators between different data types
  - mixing different sizes
  - mixing integer and floating point variables

```
int i1 = 42, i2 = 23;
short s = 13;
float f = 3.14;

i1 / i2;     /* results in 1, not a real division */
i1 + s;      /* int and short, result is int */
i1 / f;      /* result is float, actual division */
```

Syntactic sugar

As you have seen, you can use any expression on the right side of the assignment operator.
This expression often contains the variable it is assigned to.

To avoid redundancy, C offers the following short forms:

```
a += 4;      /* a = a + 4; */
a -= 4;      /* a = a - 4; */
a *= b;      /* a = a * b; */
b /= 42;     /* b = b / 42; */
b %= 2;      /* b = b % 2; */
c++;         /* c = c + 1; */
++c;         /* c = c + 1; */
c--;         /* c = c - 1; */
--c;         /* c = c - 1; */
```