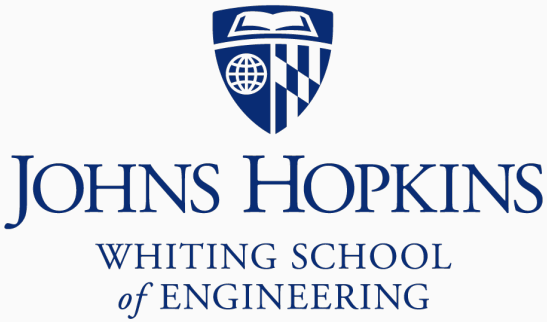
Variables, types, operators

Ben Langmead

ben.langmead@gmail.com

www.langmead-lab.org



Source markdown available at github.com/BenLangmead/c-cpp-notes

Variables, types, operators

What does this program do?

```c
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%.2f", y); // print up to 2 decimal places
    return 0;
}
```

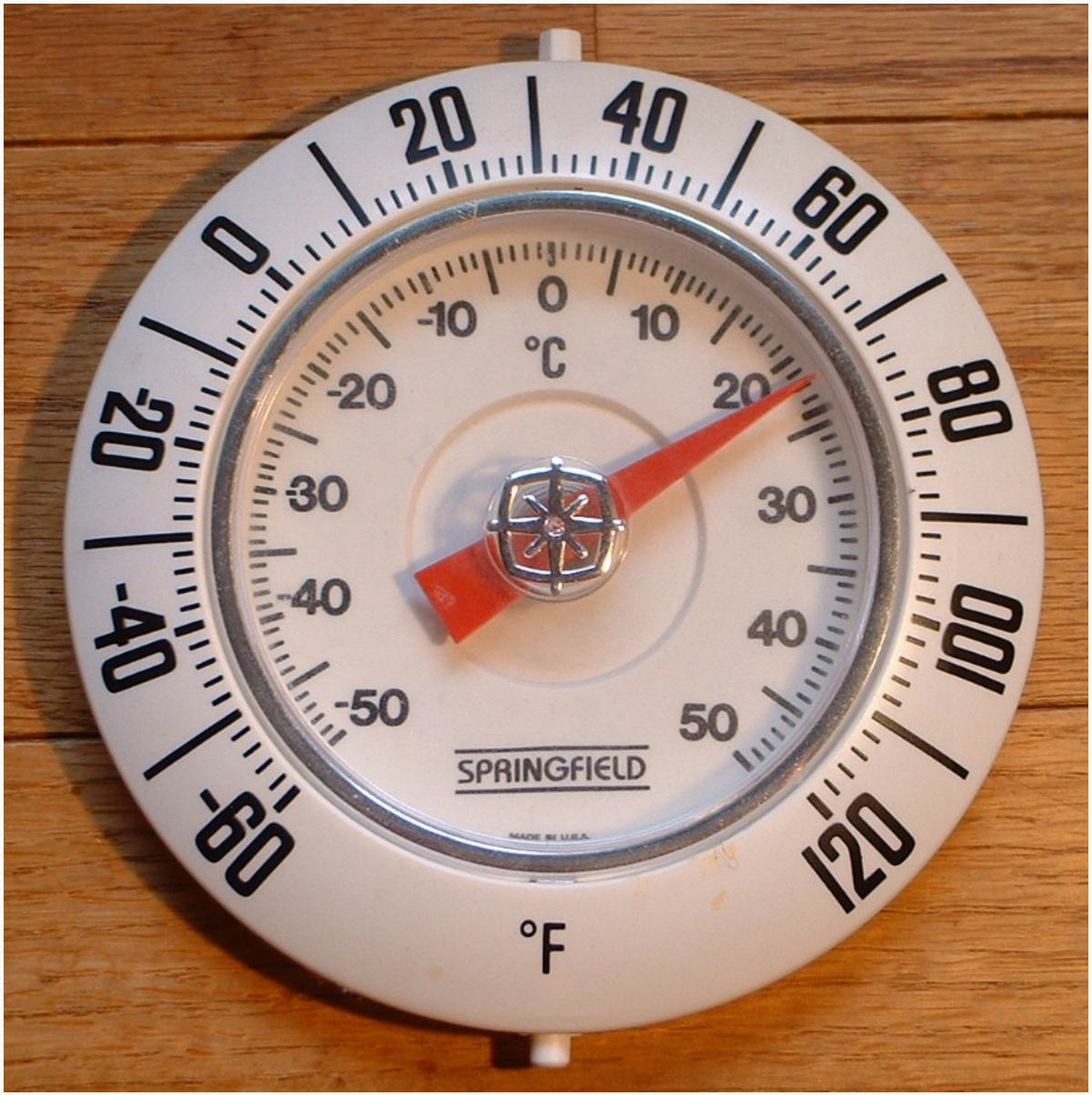
Mystery program

```c
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%.2f", y); // print up to 2 decimal places
    return 0;
}
```

```
$ gcc mysterious.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
21.67
```

Mysterious program



https://en.wikipedia.org/wiki/Fahrenheit#/media/File:Raumthermometer_Fahrenheit%2BCelsius.jpg

Mysterious program

```c
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%.2f", y); // print up to 2 decimal places
    return 0;
}
```

```
$ gcc mysterious.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
21.67
```

Mystery program

```c
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%0.2f", y); // print up to 2 decimal places
    return 0;
}
```

- `x` and `y` are *variables*
- `int` and `float` are their *types*
- `5.0 / 9.0 * (x - 32)` converts fahrenheit to celsius
- `printf` prints result as a decimal number

Variables & Assignment

```
int num_students;
```

- Variable declaration gives a *type* (`int`) and *name* (`num_students`)
- A variable has a *value* that may change throughput the program's lifetime

= is the *assignment operator*, which modifies a variable's value

```
num_students = 32; // assign
printf("Student added\n");
num_students = 33; // assign again
printf("2 students dropped\n");
num_students = 31; // assign again
```

Assignment

It is good practice to declare and assign *at the same time*:

```
int num_students = 32;
```

This is also called *initializing* the variable

A variable that has been declared but not yet assigned has an "undefined" value

Mystery program

What's a good way to make this mystery program less mysterious?

```c
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%0.2f", y); // print up to 2 decimal places
    return 0;
}
```

Less mysterious program

```c
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    int fahrenheit = 71;
    float celsius = 5.0 / 9.0 * (fahrenheit - 32);
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}
```

- Give variables meaningful names
- // explanatory comments

Types

- Integer types
  - `int`: signed integer
  - `unsigned`: unsigned integer
- Floating-point (decimal) types
  - `float`: single-precision floating point number
  - `double`: double-precision floating point number

Review

```
int num_students = 35;
```

- Variables have a name, a value and a type
- Type determines the kind of values it's allowed to have
- Good practice: give (*assign*) a variable a value at the same time as you *declare* it
- Give variables meaningful names

Types

Integer types

- `int`: positive or negative integer
- `unsigned int`: positive integer (or 0)

```
#include <stdio.h>

int main() {
    int x = -10;
    unsigned int y = 333;
    printf("%d %u\n", x, y);
    return 0;
}
```

```
$ gcc ints.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
-10 333
```

Types

Floating-point (decimal) types

- `float`: single-precision floating point number
- `double`: double-precision floating point number

```
#include <stdio.h>

int main() {
    float w = 10000.0;
    double z = 3.141592;
    printf("%f %.3f %e\n", w, z, w);
    // %e: scientific notation, %.3f: print 3 decimal places
    return 0;
}
```

```
$ gcc floats.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
10000.000000 3.142 1.000000e+04
```

Types

`char`: character type

- Holds a 1-byte character, `'A'`, `'B'`, `'$'`, `'\n'`...
- chars are basically integers, as we'll see

Types

Some other languages support a “Boolean” (true or false) type

In C, integers function as booleans

- `0`=`false`, non-`0`=`true`

Types

Type	Stands for	Size*	Range
`char`	Character	1 byte	-128 – 127
`short`	Short integer	2 bytes	-32,768 – 32,767
`int`	Integer	4 bytes	-2^{31} – $(2^{31}-1)$
`long`	Long integer	8 bytes	-2^{63} – $(2^{63}-1)$
`float`	Floating point (single precision)	4 bytes	1.2e-38 – 3.4e+38
`double`	Floating point (double precision)	8 bytes	2.3e-308 – 1.7e+308

* Size can vary depending on platform

Types

Unsigned types can't represent negative numbers, but range to somewhat higher positive numbers:

Type	Stands for	Size*	Range
`unsigned char`	Unsigned character	1 byte	0 – 255
`unsigned int`	Unsigned integer	4 bytes	0 – 2^{32}
`unsigned long`	Unsigned long integer	8 bytes	0 – 2^{64}
`size_t`	(same)	8 bytes	0 – 2^{64}

No such thing as `unsigned float` or `unsigned double`

More on *numeric representations* and *numeric precision* later

We typically use `int`, `double` and `char`

Operators

```
3 + 4
```

- 3 and 4 are *operands*, + is *operator*
- 3 and 4 are *constants* (not variables)

```
num_students + 4
```

- num_students and 4 are operands, + is operator
- num_students is a variable

Arithmetic operators

Operator	Function	Expression	Result
`+`	Addition	`4 + 3`	`7`
`-`	Subtraction	`7 - 6`	`1`
`*`	Multiplication	`4 * 5`	`20`
`/`	Integer division	`7 / 2`	`3`
`/`	Floating-point division	`7.0 / 2.0`	`3.5`
`%`	Integer remainder	`11 % 3`	`2`

Mysterious program

```c
#include <stdio.h>

int main() {
    int x = 71;
    float y = 5.0 / 9.0 * (x - 32);
    printf("%0.2f", y); // print up to 2 decimal places
    return 0;
}
```

```
$ gcc mysterious.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
21.67
```

Less mysterious program

```c
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    int fahrenheit = 71;
    float celsius = 5.0 / 9.0 * (fahrenheit - 32);
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}
```

- Output is correct (double-check with google)

```
$ gcc convert_fc.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
21.67
```

Mistake?

```c
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    int fahrenheit = 71;
    //float celsius = 5.0 / 9.0 * (fahrenheit - 32);
    float celsius = 5.0 / 9.0 * fahrenheit - 32;
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}
```

Mistake?

```c
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    int fahrenheit = 71;
    float celsius = 5.0 / 9.0 *  fahrenheit - 32 ;
    //                was: 9.0 * (fahrenheit - 32);
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}
```

```
$ gcc convert_fc_badprec.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
7.44
```

- Mistake because multiplication & division have higher *precedence* than subtraction

Operator precedence

Higher in the table means higher precedence

Operator	Function	Associativity
`()`	Function call / parentheses	left to right
`[]`	Array subscript	
`.`, `->`	Member selection	
`++`, `--`	Postincrement/postdecrement	
`++`, `--`	Preincrement/predecrement	right to left
`!`	Logical negation (not)	
`~`	Bitwise complement (not)	
`*`, `&`	Pointer dereference, reference	
`*`, `/`, `%`	Multiplication, division, modulus	left to right
`+`, `-`	Addition, subtraction	left to right
`<<`, `>>`	Bitwise left- and right shift	left to right
`<`, `<=`	Relational operators	left to right
`>=`, `>`		

Operator precedence

Know where to look up the rules

- en.cppreference.com/w/c/language/operator_precedence

. . . and use parentheses when in doubt

Program variant

```c
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    int fahrenheit = 71;
    int base = 32;
    float factor = 5.0 / 9.0;
    float celsius = factor * (fahrenheit - base);
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}
```

Result is same

```
$ gcc convert_fc_var1.c -std=c99 -pedantic -Wall -Wextra
$ ./a.out
21.67
```

Using const well

```c
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    int fahrenheit = 71;
    const int base = 32; // can't be modified
    const float factor = 5.0 / 9.0; // can't be modified
    float celsius = factor * (fahrenheit - base);
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}
```

const

```
const int base = 32;
```

const before a type says the variable *cannot be modified later*

const

```c
#include <stdio.h>

// Convert 71 degrees fahrenheit to celsius, print result
int main() {
    const int fahrenheit = 71;
    const float celsius = 5.0 / 9.0 * fahrenheit - 32;
    printf("%0.2f", celsius); // print up to 2 decimal places

    celsius = 5.0 / 9.0 * 07 - 32; // oops! reassigning to a const
    printf("%0.2f", celsius); // print up to 2 decimal places
    return 0;
}
```

const

```
$ gcc convert_fc_var3.c -std=c99 -pedantic -Wall -Wextra
convert_fc_var3.c: In function 'main':
convert_fc_var3.c:9:13: error: assignment of read-only variable 'celsius'
     celsius = 5.0 / 9.0 * 07 - 32; // oops! reassigning to a const
             ^
```

Logical & relational operators

Relational & logical operators:

Operator	Function	Example	Result
== / !=	Equals / does not equal	`1 == 1`	true (non-0)
< / >	Less / greater	`5 < 7`	true (non-0)
<= / >=	Less / greater or equal	`5 >= 7`	false (0)
&&	Both true (*AND*)	`1 && 0`	false (0)
||	Either true (*OR*)	`1 || 0`	true (non-0)
!	Opposite (*NOT*)	`!(1 || 0)`	false (0)