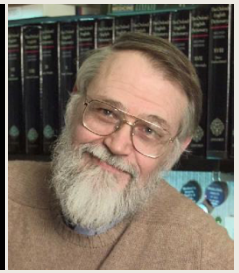
Introduction to C

Programming Workshop in C (67316)
Fall 2017
Lecture 1
24.10.2017

History



He has said that if stranded on an island with only one programming language it would have to be C

C → C++ → Java

C	C++	Java
70's – Development of UNIX. (Ritchie & Kernighan – AT&T Bell)	80's – Large efficient code. (Stroustrup – Bell)	90's – Language for the web (Sun Microsystems)

Simple to convert to machine code	⟷	VM as a middle layer
Fast and Efficient	⟷	Secure and Safe
Low level coding	⟷	High level coding

C – 32 Keywords only

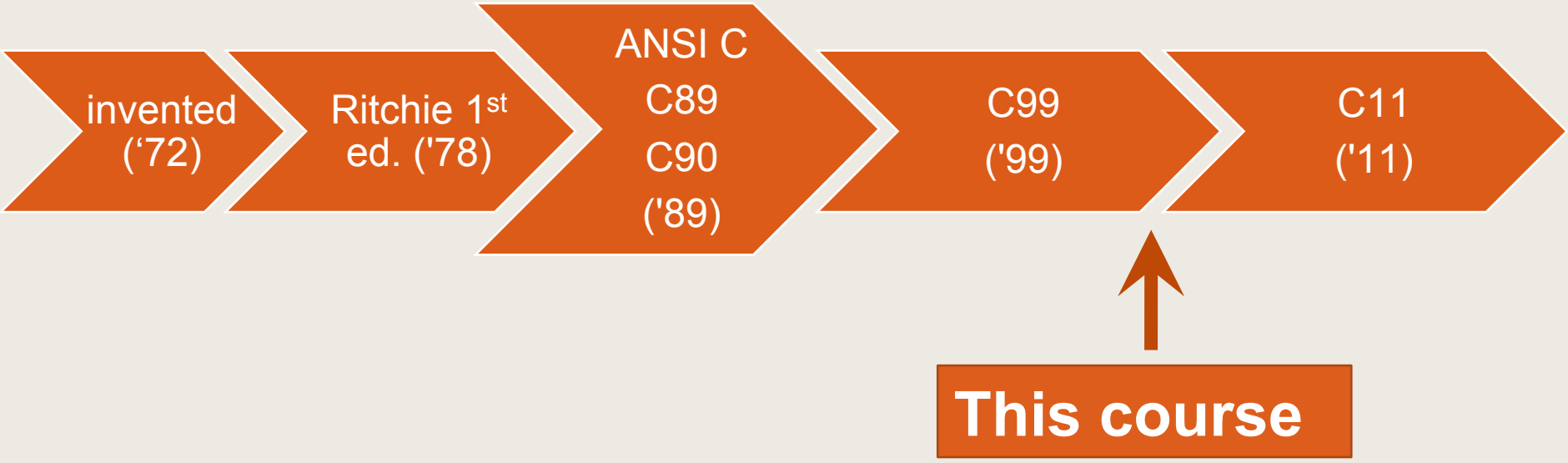
History – C:

- First "standard”
- Default int

- enums
- return struct
- fun. prototypes
- void

- single line comments //
- VLA
- Variadic macros
- inline

- multithreading
- Anonymous structs
- _Generic



C – Design Decisions

Efficient & Simple

- Deals with objects that computers do (e.g.: ints, no lists)
- Tightly controls memory & CPU usage
- Can understand & use ENTIRE language

Type checking

- But: “Programmers know what they are doing”

Portable (OS, HW) code

- Standard language definition
- Standard library

C – Design Decisions

Disadvantages

- May be time-consuming and error-prone to code some tasks (e.g. home-made text parsing)
- Low level coding means it's easier to make fatal mistakes (e.g. invalid memory pointers)

C – What is it used for?

Systems programming

- Operating systems, such as Linux
- microcontrollers: cars & planes
- embedded processors: phones, portable electronics
- DSP processors: digital audio & TV systems
- where there are tight limits on memory and CPU time

C – beware:

No run-time checks

- Array boundary overruns
- Illegal pointers

No memory management

- Programmer has to manage memory
  - Get memory from OS
  - Release it when done using it

Note: You can work in C without dealing with memory, like we will do in the beginning of our course

C++ - OO extension of C

Classes & methods

- OO design of classes

(More) Generic programming

- Templates allow for code reuse

Stricter type system

Some run-time checks & memory control

The evolution of coding

Last century

This century

code your function

```
int gcd(int n1, int n2)
{
  if (n2 != 0) {
    return gcd(n2, n1%n2);
  } else {
    return n1;
  }
}
```

google it

cppreference.com

Page | Discussion

C++ | Numerics library

std::gcd

Defined in header <numeric>

```
template< class M, class N>
constexpr std::common_type_t<M, N> gcd(M m, N n);
```

Computes the greatest common divisor of the integers m and n.

stackoverflow Question

Stack Overflow is a community of 7.9

GCD function for C

6

1

Q 1. Problem 5 (evenly divi
sites and found this code:

```
#include<stdio.h>
int gcd(int a, int b)
{
    while (b != 0)
    {
        a %= b;
        a ^= b;
        b ^= a;
        a ^= b;
    }

    return a;
}
```

The evolution of coding

- The programs are constantly growing
- We need to learn how to read code (not only write!)
- Debugging is a big part of coding

Year	Version	Source lines of code (millions)
2001	Linux kernel 2.4.2	2.4
2003	Linux kernel 2.6.0	5.2
2009	Linux kernel 2.6.29	11.0
2009	Linux kernel 2.6.32	12.6
2010	Linux kernel 2.6.35	13.5
2012	Linux kernel 3.6	15.9
2015	Linux kernel pre-4.2	20.2

https://en.wikipedia.org/wiki/Source_lines_of_code

Hello World!

First Program in C

```
// This line is a comment,
/* and those lines also.
Next line includes standard I/O header file */
#include <stdio.h> //part of the C Standard Library

// main function – program entry point.
// Execution starts here
int main()
{ // {…} define a block
    printf("Hello class!\n");
    return 0;
}
```

Note:
Coding guidelines are online. The slides are not a reference for coding style.

Compiling & Running…

```
> gcc hello.c
> a.out
Hello class!
```


```
> gcc -std=c99 -Wall hello.c -o hello
> hello
Hello class!
```

Debugging

```
> gcc -g -Wall hello.c -o hello
> hello
Hello class!
> gdb hello
```

add -g option to enable debugging

Some useful commands:

- `break` *`linenumber`* – create breakpoint at specified line
- `break` *`file:linenumber`* – create breakpoint at line in file
- `run` – run program
- `c` – continue execution
- `next` – execute next line
- `step` – execute next line or step into function
- `quit` – quit gdb
- `print` *`expression`* – print current value of the specified expression
- `help command` – in-program help

Why coding style matters?

- Coding style is how your code looks
- Coding style is personal
  - CamelCase: `aRatherLongSymbolName`
  - snake_case: `a_rather_long_symbol_name`
- Big projects require standards for coding style
- These conventions usually include file organization, indentation, comments, declarations, statements, white space, naming conventions, programming practices...
- Coding style guides are an important part of writing code as a professional

Variables

Variables

Statically typed – each variable has a type which is known at **compile time**

Declare by: <type> <name>

```
int x;
int x,y;
```

Important word in C.
Arbitrary initial value (=garbage)
can be very confusing to debug

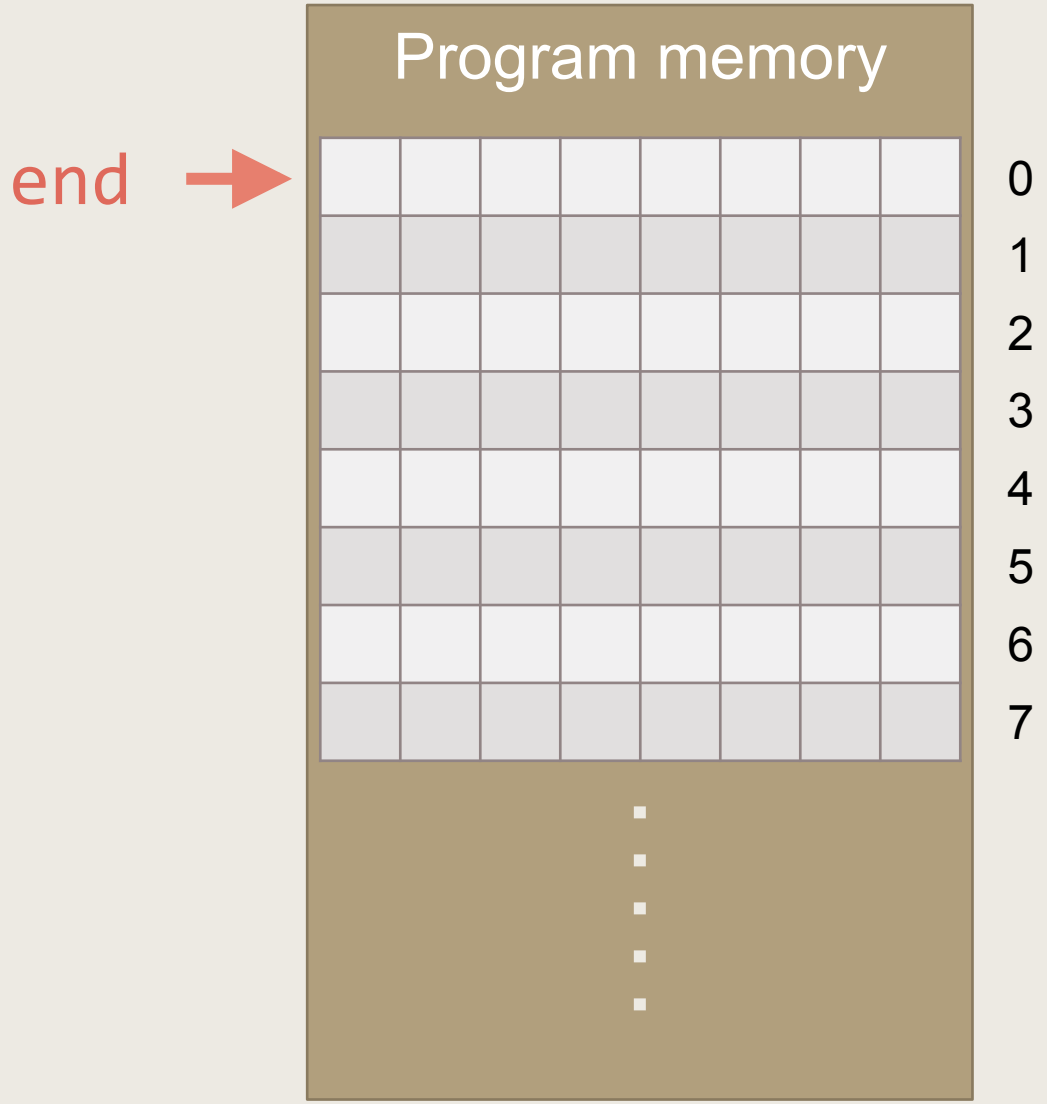
```
int x=0;
```

Optionally **initialize** (otherwise, for local variables, it is **undefined!**)

Numeric data types in C

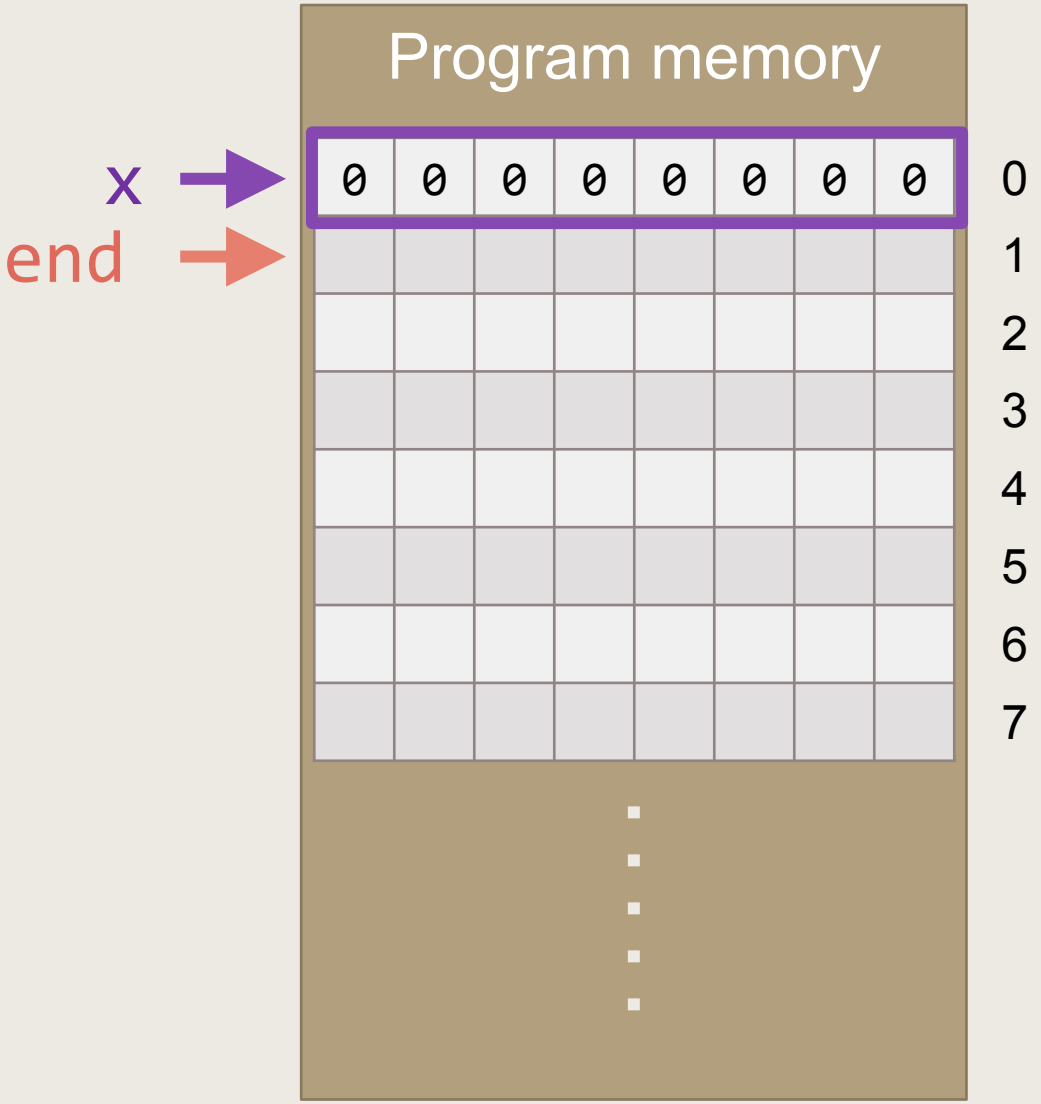
```
char c = ‘A’;
short s = 0;
int x = 1;
long y = 9;

unsigned char c = ‘A’;
unsigned short s = 0;
unsigned int x = 1;
unsigned long y = 9;

float x = 0.0;
double y = 1.0;

// more on initialization
char a = ‘A’, b = ‘B’;
int x = y = 1; // equivalent to int x = (y = 1);
```

Defining new variables



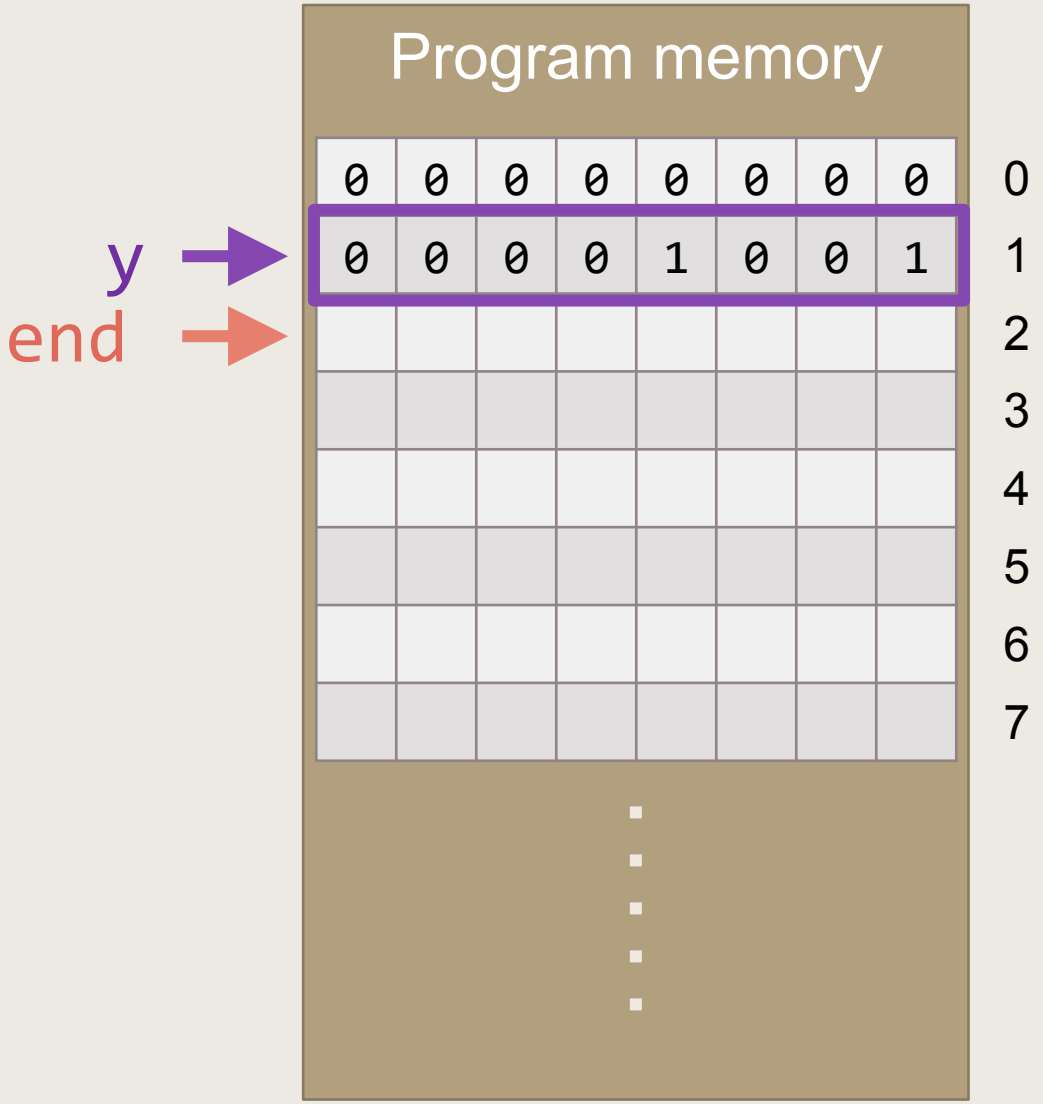
Defining new variables

```
...
int x = 0;
```



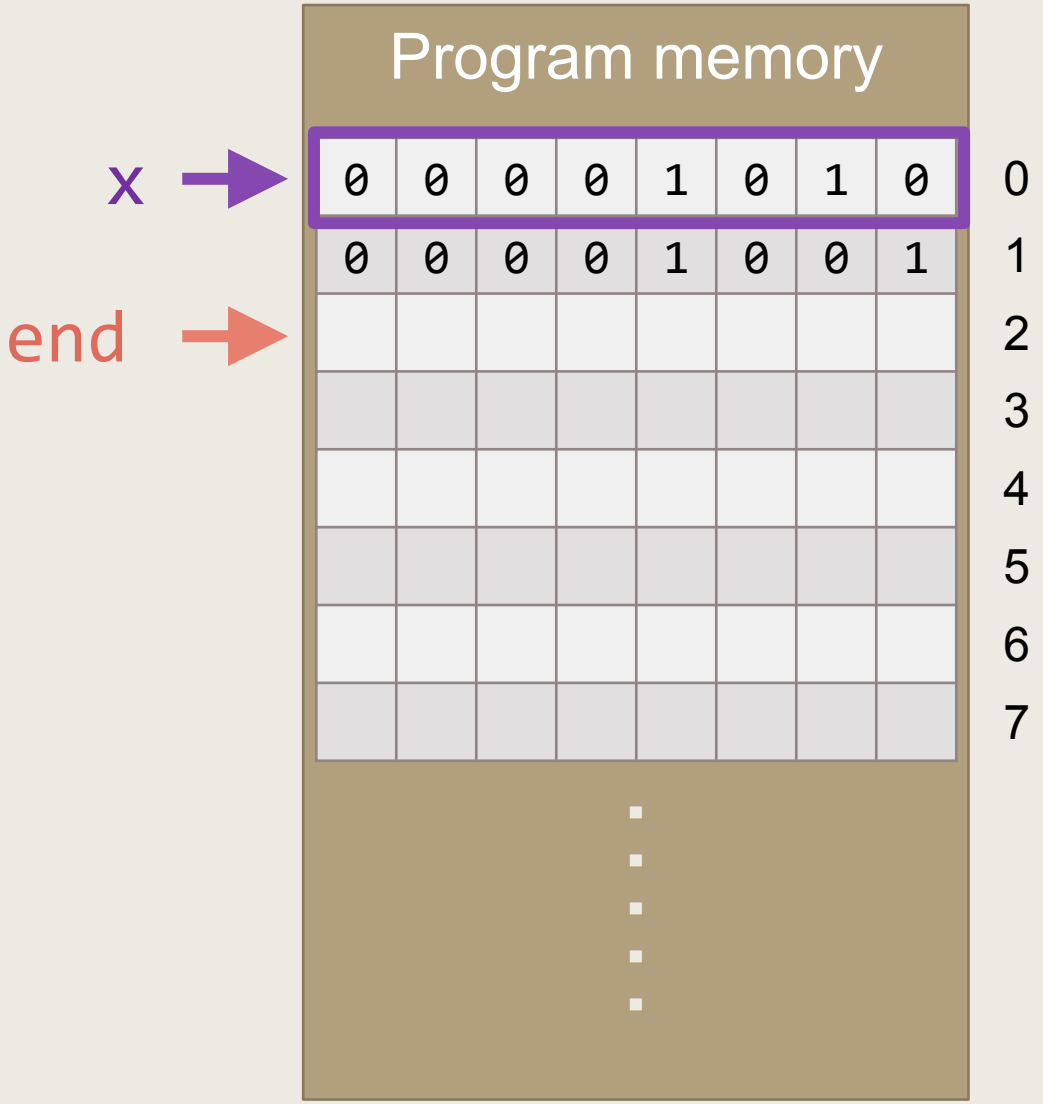
Defining new variables

```
...
int x = 0;
int y = 9;
```

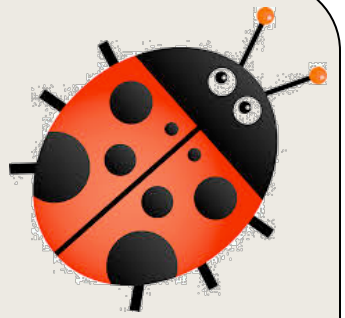


Reading and setting variables

```
...
int x = 0;
int y = 9;
x = y + 1;
```



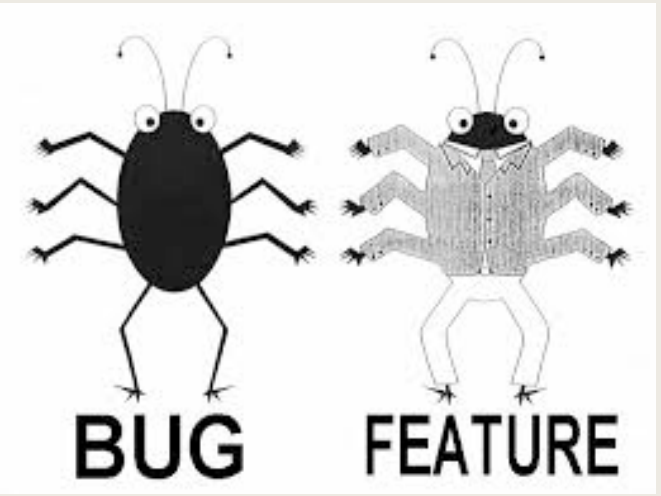
Beware of overflow



```
...
unsigned char x = 0;
x = x - 1; // x = 255
```

Fun fact - Gandhi and Civilization game

- In the *Civilization* India's supposedly-peaceful leader Gandhi has been famous for dropping nukes
- Each leader in the game had an "aggression" score
- Gandhi had the lowest score of 1
- When a player adopted democracy in *Civilization*, their aggression would be automatically reduced by 2



http://kotaku.com/

Primitive types and `sizeof` operator

```
// Queries size of the object or type
unsigned long numberOfBytes =
                    sizeof(unsigned char);
```

The individual sizes are machine/compiler dependent.

The following is guaranteed:

```
sizeof(char)<sizeof(short)<=sizeof(int)<=sizeof(long)
```

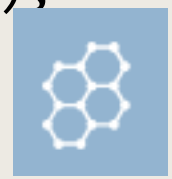
and

```
sizeof(char)<sizeof(short)<=sizeof(float)<=sizeof(double)
```

What are the sizes of standard variables?

```c
/* A program that prints variable sizes */
#include <stdio.h>

int main()
{
  // Basic primitive types:
  printf("Size of char %lu\n", sizeof(char));
  printf("Size of short %lu\n", sizeof(short));
  printf("Size of int %lu\n", sizeof(int));
  printf("Size of long %lu\n", sizeof(long));
  printf("Size of float %lu\n", sizeof(float));
  printf("Size of double %lu\n", sizeof(double));

  // Other types:
  printf("Size of long double %lu\n",sizeof(long double));
  return 0;
}
```

l - is a *length* modifier meaning "long"
u - is a *specifier* meaning "unsigned decimal integer"
see printf Documentation for more options



Variables

Where to declare?

1. Inside a block (C89 - block beginning), visible only in block
2. Outside all blocks – global - will be visible everywhere

```
int x=0; // global
int main()
{
   int x=1; //local hides global
   {
      int x=2; //local hides outer scope
      //x is 2
   }
   //x is 1 again!
}
```

Scopes

- **Code block** defined with “**{**” and “**}**”
- **Nesting** is possible
- Only declarations inside current or outer scopes are visible
- Declarations in inner scope **hide** declarations in outer scopes
- Outmost scope (global) has no brackets
- Keep in mind that a function is also a scope

Scopes

```
int y=5,x=0;
{
    int x=y;
    // x is 5
    {
        int y=0;
        // x is ?
    }
}
// x is ?
```

Scopes

```
int y=5,x=0;
{
    int x=y;
    // x is 5
    {
        int y=0;
        // x is 5
    }
}
// x is 0
```

Operators

Operator Types in C

- Arithmetic operators: + - * / %
- Increment and decrement operators ++ --
- Relational operators: < <= > >= == !=
- Logical operators: && || !
- Bitwise operators: & | ^ << >> -
- Assignment operators: += -= *= /= %= ...

- can be binary (such as +), unary (such as ++), or ternary

Arithmetic operators

- **The order of evaluation is as in algebraic expressions:**
  - **brackets first, followed by * and /, followed by +**
  - **from left to right**

Operator	Meaning	Examples
+	addition	`int x = y + 3;`
-	subtraction	`int x = y - 3;`
*	multiplication	`float z = x * y;`

/	division	`float x = 3 / 2; // = 1`
		`float y = 3.0 / 2; // = 1.5`
		`int z = 3.0 / 2; // = 1`

%	remainder	`int x = 3 % 2;`

Increment and decrement operators

- **++ and -- operators are shortcuts:**

`x++;`	`x = x + 1;`	
`y = x++;`	`y = x; x = x + 1;`	x is evaluated **before** it is increment
`y = ++x;`	`x = x + 1; y = x;`	x is evaluated **after** it is incremented