Compilation and C

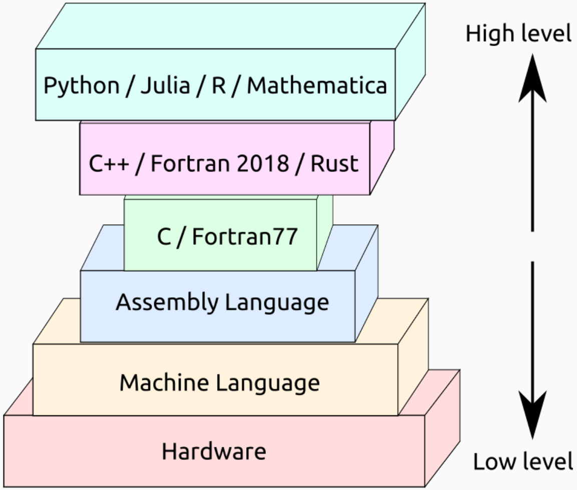
A. Scemama

21/11/2024

Laboratoire de Chimie et Physique Quantiques, Univ. Toulouse/CNRS

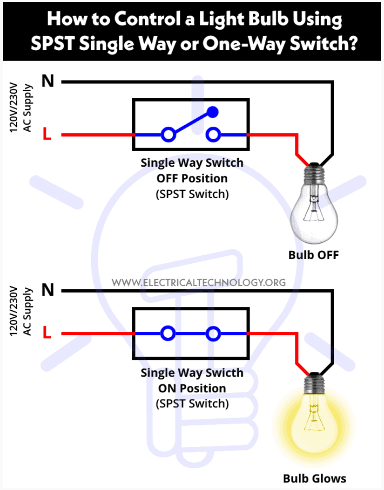
Introduction to hardware and binary representation

Introduction



On/Off

On/Off switch



How to Control a Light Bulb Using SPST Single Way or One-Way Switch?
120V/230V AC Supply
N
L
Single Way Switch OFF Position (SPST Switch)
Bulb OFF
WWW.ELECTRICALTECHNOLOGY.ORG
120V/230V AC Supply
N
L
Single Way Switch ON Position (SPST Switch)
Bulb Glows

Binary notation

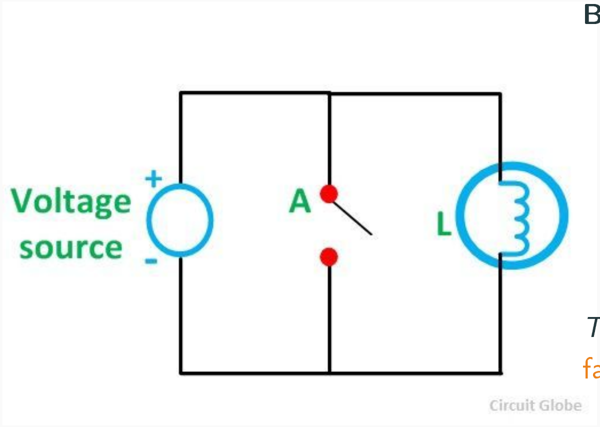
- 0: Off
- 1: On

Switch Off	$\rightarrow$ Light Off	$f(0) = 0$
Switch On	$\rightarrow$ Light On	$f(1) = 1$

The light is on is true if *the switch is on* is true.

NOT Gate

NOT Gate



Binary notation

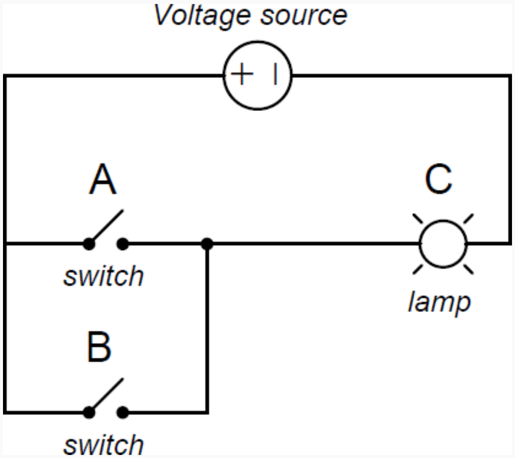
Switch Off	$\rightarrow$ Light On
Switch On	$\rightarrow$ Light Off

- $\text{not}(0) = \neg 0 = 1$
- $\text{not}(1) = \neg 1 = 0$

The light is on is true if *the switch is on* is false.

OR gate

OR gate

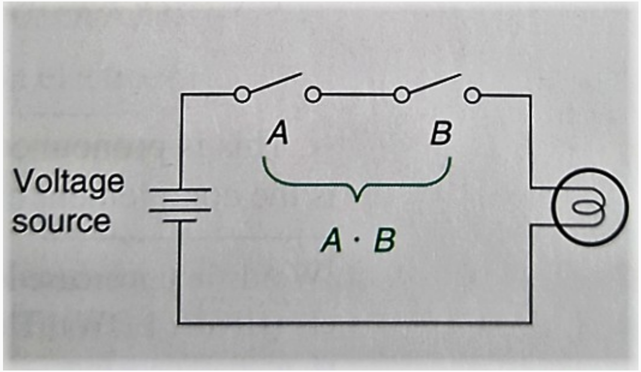


Binary operation

- $\text{or}(0,0) = 0 \vee 0 = 0$
- $\text{or}(0,1) = 0 \vee 1 = 1$
- $\text{or}(1,0) = 1 \vee 0 = 1$
- $\text{or}(1,1) = 1 \vee 1 = 1$

The light is on is true if either *switch A is on* or *switch B is on* is true.

AND gate



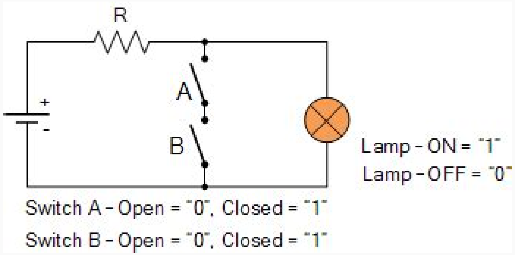
Binary operation

- $\text{and}(0,0) = 0 \wedge 0 = 0$
- $\text{and}(0,1) = 0 \wedge 1 = 0$
- $\text{and}(1,0) = 1 \wedge 0 = 0$
- $\text{and}(1,1) = 1 \wedge 1 = 1$

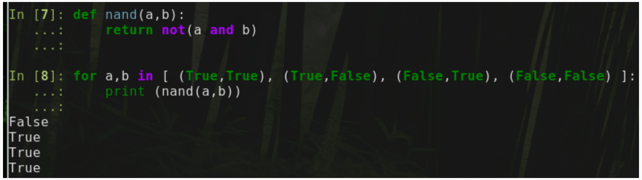
The light is on is true if both *switch A is on* and *switch B is on* are true.

NAND gate

NAND gate: Not And



R
A
B
+
-
Lamp - ON = "1"
Lamp - OFF = "0"
Switch A - Open = "0", Closed = "1"
Switch B - Open = "0", Closed = "1"



```
In [7]: def nand(a,b):
   ...:     return not(a and b)
   ...:

In [8]: for a,b in [ (True,True), (True,False), (False,True), (False,False) ]:
   ...:     print (nand(a,b))
   ...:
False
True
True
True
```

Binary operation

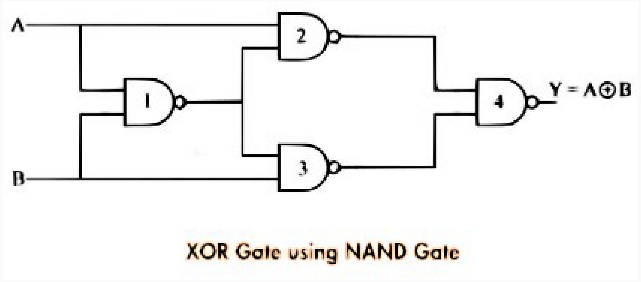
- $\text{nand}(0,0) = \neg(0 \wedge 0) = 1$
- $\text{nand}(0,1) = \neg(0 \wedge 1) = 1$
- $\text{nand}(1,0) = \neg(1 \wedge 0) = 1$
- $\text{nand}(1,1) = \neg(1 \wedge 1) = 0$

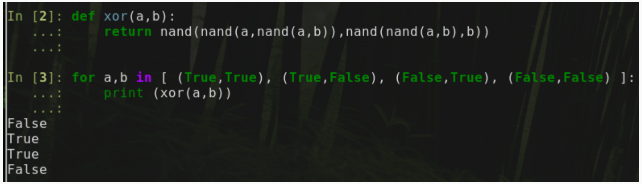
The light is on is false if both *switch A is on* and *switch B is on* are true.

Any logic function can be implemented using only NAND gates.

XOR gate

XOR gate: Exclusive OR





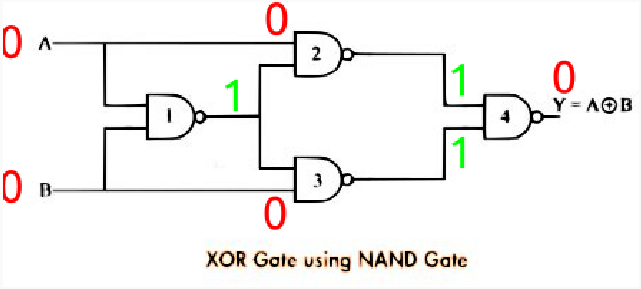
```
In [2]: def xor(a,b):
   ...:     return nand(nand(a,nand(a,b)),nand(nand(a,b),b))
   ...:

In [3]: for a,b in [ (True,True), (True,False), (False,True), (False,False) ]:
   ...:     print (xor(a,b))
   ...:
False
True
True
False
```

Binary operation

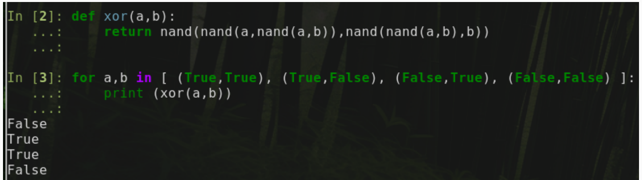
- $\text{xor}(a, b) = (a \wedge \neg b) \vee (\neg a \wedge b)$
- Either a or b is true, but not both

XOR gate

XOR gate: Exclusive OR



XOR Gate using NAND Gate



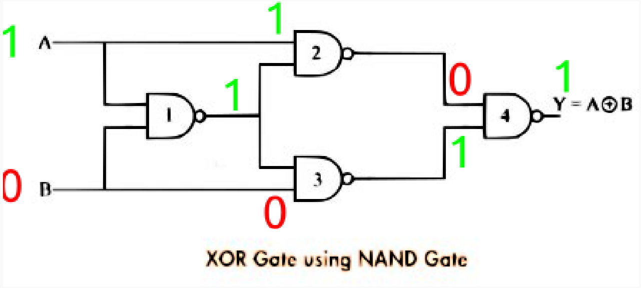
```
In [2]: def xor(a,b):
   ...:     return nand(nand(a,nand(a,b)),nand(nand(a,b),b))
   ...:

In [3]: for a,b in [ (True,True), (True,False), (False,True), (False,False) ]:
   ...:     print (xor(a,b))
   ...:
False
True
True
False
```

Binary operation

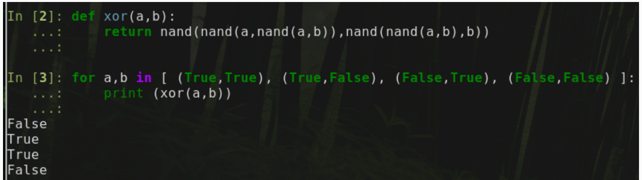
A	B	C
0	0	0
1	0	1
0	1	1
1	1	0

XOR gate

XOR gate: Exclusive OR



XOR Gate using NAND Gate



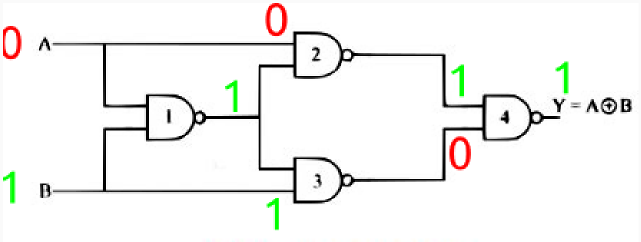
```
In [2]: def xor(a,b):
   ...:     return nand(nand(a,nand(a,b)),nand(nand(a,b),b))
   ...:

In [3]: for a,b in [ (True,True), (True,False), (False,True), (False,False) ]:
   ...:     print (xor(a,b))
   ...:
False
True
True
False
```

Binary operation

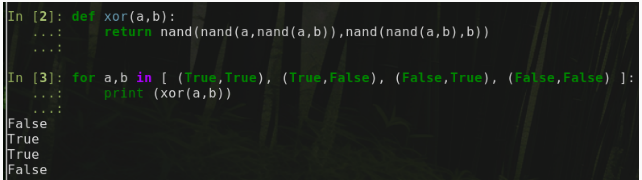
A	B	C
0	0	0
1	0	1
0	1	1
1	1	0

XOR gate

XOR gate: Exclusive OR



XOR Gate using NAND Gate



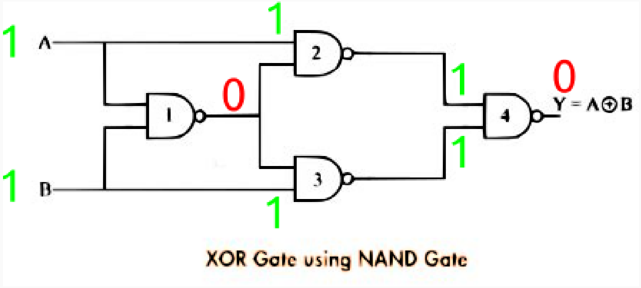
```
In [2]: def xor(a,b):
   ...:     return nand(nand(a,nand(a,b)),nand(nand(a,b),b))
   ...:

In [3]: for a,b in [ (True,True), (True,False), (False,True), (False,False) ]:
   ...:     print (xor(a,b))
   ...:
False
True
True
False
```

Binary operation

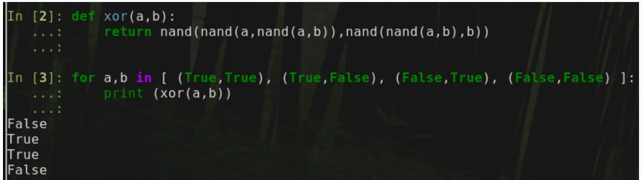
A	B	C
0	0	0
1	0	1
0	1	1
1	1	0

XOR gate

XOR gate: Exclusive OR



XOR Gate using NAND Gate



```
In [2]: def xor(a,b):
   ...:     return nand(nand(a,nand(a,b)),nand(nand(a,b),b))
   ...:

In [3]: for a,b in [ (True,True), (True,False), (False,True), (False,False) ]:
   ...:     print (xor(a,b))
   ...:
False
True
True
False
```

Binary operation

A	B	C
0	0	0
1	0	1
0	1	1
1	1	0

Decimal format

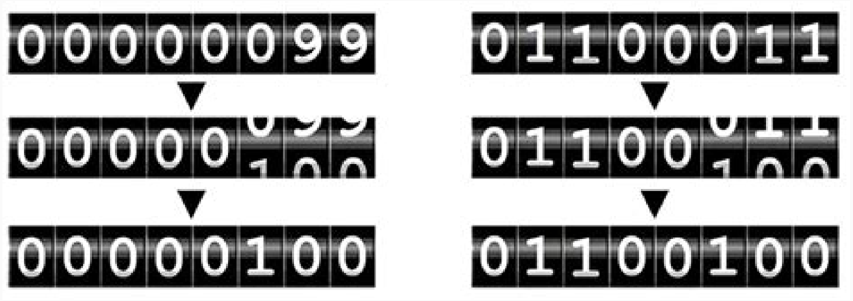
- Each digit has 10 possibilities: 0-9
- $74362 = 7 \times 10^4 + 4 \times 10^3 + 3 \times 10^2 + 6 \times 10^1 + 2 \times 10^0$

10ˆ	4	3	2	1	0
$i\times$	7	4	3	6	2

Binary format

- Each digit has 2 possibilities: 0,1
- $74362 = 2^{16}+2^{13}+2^{9}+2^{6}+2^{5}+2^{4}+2^{3}+2^{1} = 65536+8192+512+64+32+16+8+2$

2^	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
$i\times$	1	0	0	1	0	0	0	1	0	0	1	1	1	1	0	1	0



How to add two binary numbers?

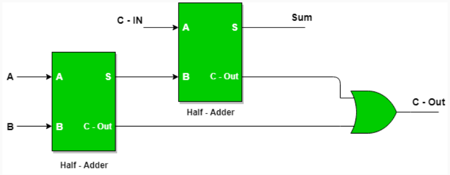
$1011 + 11 = ?$

$$\begin{array}{ccccc} & & 1 & 1 & \\ & 1 & 0 & 1 & 1 \\ + & & & 1 & 1 \\ \hline & 1 & 1 & 1 & 0 \end{array}$$

$$\begin{array}{ccccl} 0 & + & 0 & 0 & \\ 1 & + & 0 & 1 & \\ 0 & + & 1 & 1 & \\ 1 & + & 1 & 0 & \text{carry the 1} \\ \hline & & & \text{xor} & \text{and} \end{array}$$

Hardware

- Half-adder:
  - 2 inputs (bits)
  - 2 outputs (sum and carry)
  - One XOR and one AND gates
- Full adder:
  - 3 inputs (2 bits and one carry)
  - 2 outputs (sum carry)
  - 2 Half adders and an OR gate



How to subtract two binary numbers?

Introduce negative numbers using 2's complement

- The most significant is the sign bit (0: positive, 1: negative)
- Swap each bit (NOT gate), and then add 1:
    - $01101 = 13$
    - $-01101 = -13 \longrightarrow 10010 + 1 = 10011$

	000	001	010	011	100	101	110	111
Unsigned integer	0	1	2	3	4	5	6	7
Signed integer	0	1	2	3	-4	-3	-2	-1

How to subtract two binary numbers?

To subtract, add the negative number

- Number of bits has to be kept fixed
- Using 5-bits: $1011 - 11 = 1000 = 01011 + 11101$

$$\begin{array}{cccccc} & 0 & 1 & 0 & 1 & 1 \\ + & 1 & 1 & 1 & 0 & 1 \\ \hline & 0 & 1 & 0 & 0 & 0 \end{array}$$

- Using 5-bits: $11 - 1011 = -1000 = 00011 + 10101$

$$\begin{array}{cccccc} & 0 & 0 & 0 & 1 & 1 \\ + & 1 & 0 & 1 & 0 & 1 \\ \hline & 1 & 1 & 0 & 0 & 0 \end{array}$$

- $11000 \rightarrow -(00111 + 1) = -01000 = -8$

Hexadecimal format

- A compact way to write binary format
- Pack bits by groups of 4 $\Longrightarrow$ 16 possibilities
- Represent each group of 4 in base 16 : 0-9, a-f

Example: 363845035 = 00010101101011111101010110101011

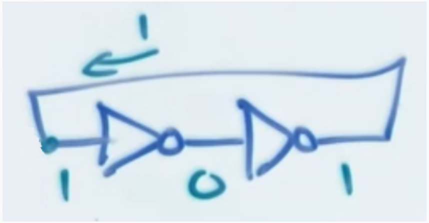
0001	0101	1010	1111	1101	0101	1010	1011
1	5	10	15	13	5	10	11
1	5	a	f	d	5	a	b

363484587 = 0x15afd5ab

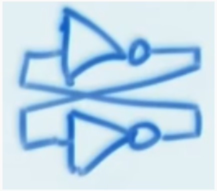
How to store a bit?

Consider this circuit

- Two inverters (NOT gates):



- Redraw as:

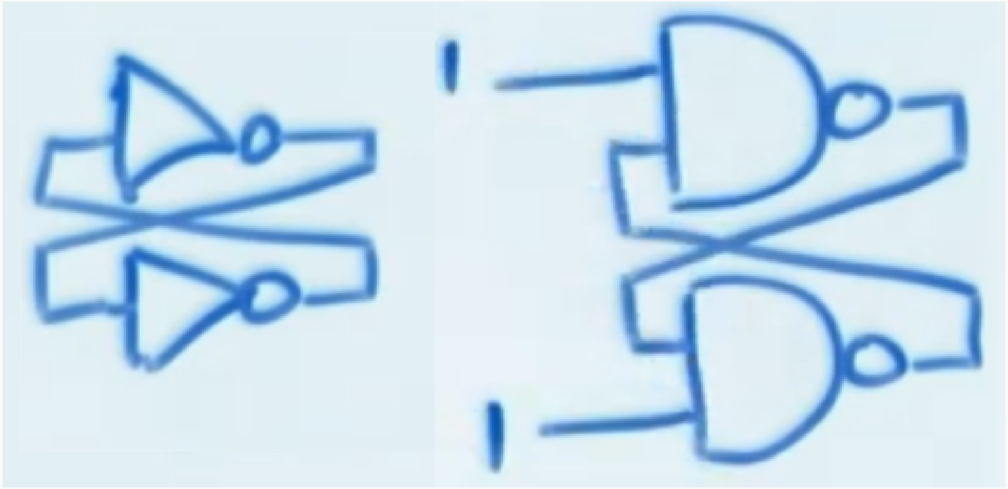


How to store a bit?

- This circuit is not programmable
- An extra input is needed to control if we store a 1 or a 0
- NAND Gates

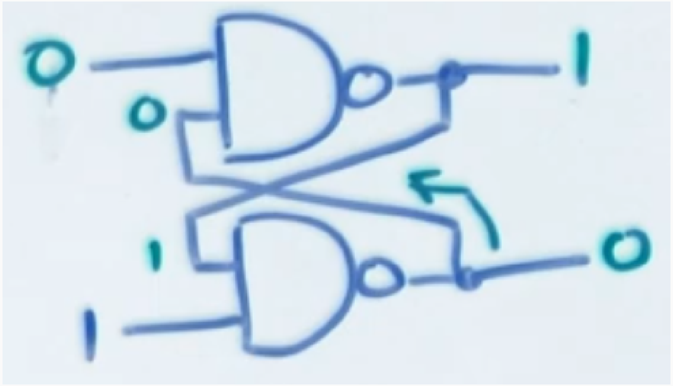
A	B	nand(AB)	
0	0	1	If A=0, nand(AB)=1
0	1	1	
1	0	1	If A=1, nand(AB)=$\neg B$
1	1	0	

How to store a bit?



How to store a bit?

Input (0,1)



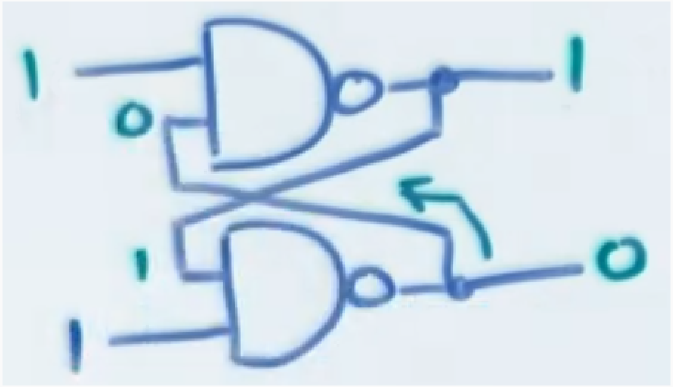
NAND Truth table

A	B	AB
0	0	1
0	1	1
1	0	1
1	1	0

Sets stored value to (1,0)

How to store a bit?

Input (1,1)



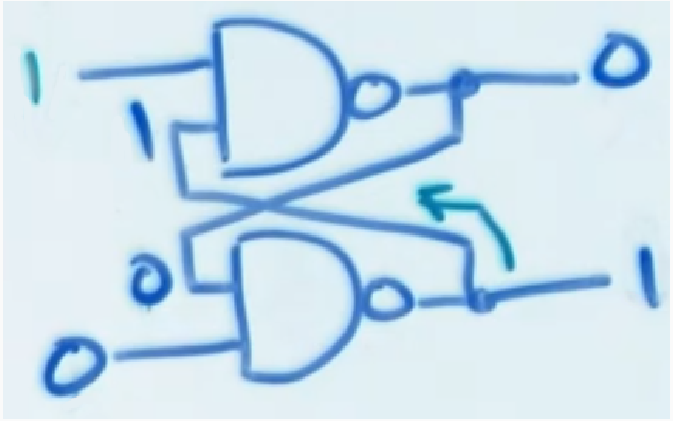
NAND Truth table

A	B	AB
0	0	1
0	1	1
1	0	1
1	1	0

Keeps (1,0) stored

How to store a bit?

Input (1,0)



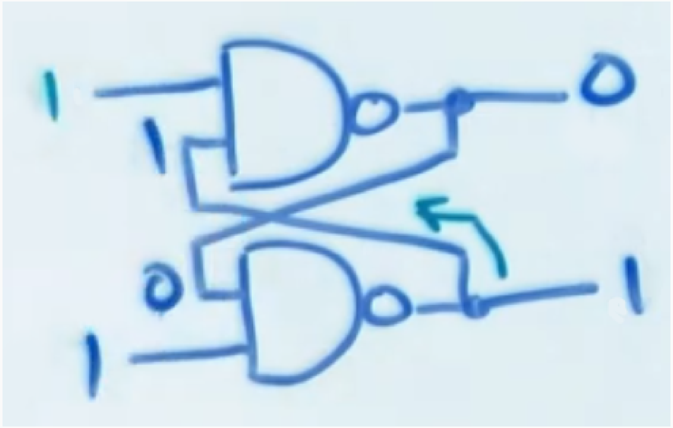
NAND Truth table

A	B	AB
0	0	1
0	1	1
1	0	1
1	1	0

Sets stored value to (0,1)

How to store a bit?

Input (1,1)



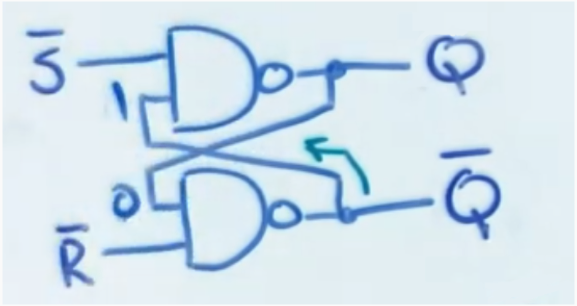
NAND Truth table

A	B	AB
0	0	1
0	1	1
1	0	1
1	1	0

Keeps (1,1) stored

How to store a bit?

The SR-Latch



$\bar{S}$	$\bar{R}$	Q	$\bar{Q}$	
0	0	-	-	
0	1	1	0	Sets Q to 1
1	0	0	1	Resets Q to 0
1	1	Q	$\bar{Q}$	Stored state

Q is the stored value

Registers

- Very fast memory *on* the CPU
- Data used for arithmetic operations (data registers, operands)
- Data used for memory (address registers)
- Data used to control the execution of a program (instruction registers)
- ...

Example: C = A + B

- A is stored in a 64-bit register
- B is stored in a 64-bit register
- Both registers give their data as inputs to the adder circuit
- The result is stored in a 64-bit register C (which can be the same as A or B, or another one)

Summary

- One bit corresponds to an on/off state
- Numbers can be represented by arrays of bits
- Operations on numbers can be performed by combining logic gates
- Numbers can be stored in registers

Data Representation

Types

Take the following sequence of bits: `1011010011101011`.

I can decide to interpret it as:

- A 16-bit unsigned integer:
  $2^{15} + 2^{13} + 2^{12} + 2^{10} + 2^{7} + 2^{6} + 2^{5} + 2^{3} + 2^{1} + 2^{0} = 46315$
- A 16-bit signed integer: -19221
- Two 8-bit unsigned integers: $(\texttt{10110100},\ \texttt{11101011}) = (180, 235)$
- Two characters : (’, ë)

The Type indicates how to interpret a sequence of bits.

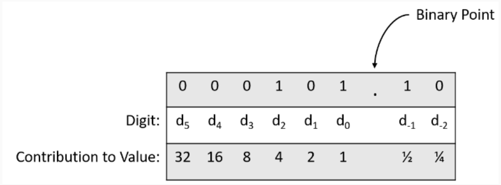
Real numbers

- $\mathbb{R}$ is continuous: Real numbers require an infinite number of bits $\implies$ they can't be represented
- We use a discretization of the real numbers (a "grid of points")
- To each 32-bit (or 64-bit) integer we associate a 32-bit (or 64-bit) floating point number
- Regular grid:
  - We choose a minimum and maximum value: range $= [x_{\min}, x_{\max}[)$
  - Each 1-bit increment adds $\delta = \frac{x_{\max} - x_{\min}}{N+1}$, where N is the largest representable integer
  - The binary number n corresponds to the value $x_{\min} + n \times \delta$

Rational numbers

Fixed-point numbers

- Integer multiplied by a scaling factor 2^{-m}
- m is the number of fraction bits


Binary Point
0 0 0 1 0 1 . 1 0
Digit: d_5 d_4 d_3 d_2 d_1 d_0 d_{-1} d_{-2}
Contribution to Value: 32 16 8 4 2 1 ½ ¼

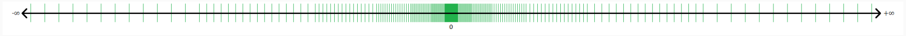
Characteristics

- If the range is chosen to be large, the precision is low
- If the range is chosen to be small, the precision is high
- Can't be used to represent both huge and tiny numbers

Rational numbers

Floating point representation

- A non-uniform grid:

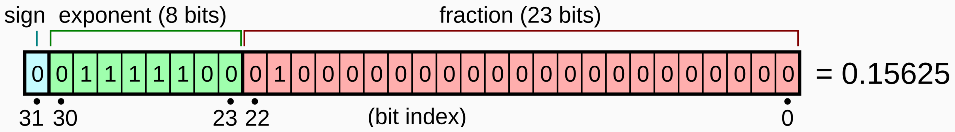


- A binary equivalent of scientific notation
- Example: `-6.63821E+04`
    - Use some bits to store 663821
    - Use some bits to store +04
    - Use a bit to store the sign

IEEE-754 Format

32-bit floating-point (FP32, single-precision)

- 1 bit: sign (S)
- 8 bits: exponent (E)
- 23 bits: fraction (F)



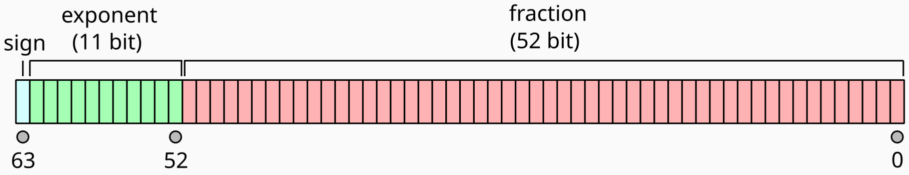
$$-1^{S} \times 2^{E-127} \times \left(1 + \sum_{i=1}^{23} F_{23-i} \times 2^{-i}\right)$$

- $S = 0$, $E = 01111100 = 124$, $F = 2097152$
- $-1^{0} \times 2^{124-127} \times \left(1 + 2^{-2}\right) = 0.15625$

IEEE-754 Format

64-bit floating-point (FP64, double-precision)

- 1 bit: sign (S)
- 11 bits: exponent (E)
- 52 bits: fraction (F)



IEEE-754 Format

Special values

- Smallest FP32: 1.1754943508 $\times 10^{-38}$

  0 00000001 00000000000000000000000
- Smallest FP64: 2.2250738585072014 10^{-308}

  0 00000000001 0000000000000000000000000000000000000000000000000000
- Largest FP32: 3.4028234664 $\times 10^{+38}$

  0 11111110 11111111111111111111111
- Largest FP64:= 1.7976931348623157 10^{+308}

  0 11111111110 1111111111111111111111111111111111111111111111111111

IEEE-754 Format

Special values

- Smallest FP32 larger than one: 1.00000011920928955
  0 01111111 00000000000000000000001
- Smallest FP64 larger than one: 1.0000000000000002
  0 01111111111 0000000000000000000000000000000000000000000000000001
- 0 00000000 00000000000000000000000 $= +0$
  1 00000000 00000000000000000000000 $= -0$
- 0 11111111 00000000000000000000000 $= +\infty$
  1 11111111 00000000000000000000000 $= -\infty$
- Not a number (NaN): Result of $\sqrt{-3.0}$
  0 11111111 00000000000000000000001 = sNaN
  0 11111111 10000000000000000000001 = qNaN

Bits/Bytes

- 1 byte (B) = 1 octet (o) = 8 bits (b)
- An ASCII character is encoded in 1 byte
- A 64-bit double precision number uses 8 bytes.
- Disk capacity or RAM is usually expressed in bytes
- Speed of disk or memory is usually expressed in bytes/second
- Network speed is usually expressed in bits/second (10Gb Ethernet)
- kilo, mega, giga, tera, peta, exa, zetta, . . .
- Often, instead of using 10^3, we use 2^{10}=1024
    - 1kb: one kilobit = 1000 bits
    - 1kB: one kilobyte = 8000 bits
    - 1KiB: one kibibyte = 1024 bytes = 8192 bits
    - 1MB: one megabyte = 1000^2 bytes = 8 000 000 bits
    - 1MiB: one mebibyte = 1024^2 bytes = 8 388 608 bits

Summary

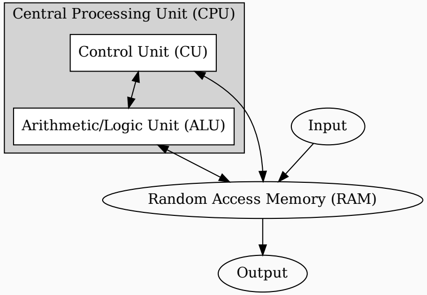
- The type tells us how to interpret bytes
- All the calculations made by a computer use a discretization of real numbers
- There is a finite number of floats used to describe real numbers
- There is a one-to-one mapping between the integers and floats
- Each operation (add, multiply, . . .) has a rounding error

Von Neumann Architecture

Von Neumann architecture

- Both program instructions and data are stored in memory
- Consequences:
    - Simpler hardware design
    - Programs can be created by other programs (compilers)
    - Scripting languages

Von Neumann architecture


Central Processing Unit (CPU)
Control Unit (CU)
Arithmetic/Logic Unit (ALU)
Input
Random Access Memory (RAM)
Output

- ALU: Performs bitwise or arithmetic operations
- Control unit: Decodes instructions fetched from memory and controls the flow of data within the CPU
  - Instruction cycle
    - Fetch: Retrieve the next instruction from memory.
    - Decode: Interpret the instruction.
    - Execute: Carry out the operation, which could involve the ALU or memory access.

Memory

Random Access Memory (RAM)

- Stores both data and instructions for access by the CPU
- Organized as a large array of cells, where each cell holds a small unit of data, typically one byte (8 bits)
- Each cell has a unique address: like a “house number” that specifies the exact location of the data within the memory space

Memory read and write

- To read the data stored at address `0xde102f`:
  - the CPU sends this address over the address bus to the memory controller
  - the memory controller then retrieves the value from the cell at address `0xde102f` and sends it back to the CPU.
- To write data to memory:
  - the CPU sends the data along with the address of the memory cell where the data should be stored

Summary

In Von Neumann architecture, code and data are stored in the same memory

Programs

What is a *program* ?

- A program is a sequence of instructions
- Each instruction is encoded in a binary format (machine language):

00101000	10001001	11001000
Opcode	Operand 1	Operand 2

Instructions

- Set a register to a constant
- Copy data from memory to a register (or backwards)
- Read/Write from hardware devices
- Add, subtract, multiply, divide the value of 2 registers, storing the result in a register
- Bitwise logic operations
- Comparisons: $<$, $>$, $=$
- Floating-point instructions
- Conditional branch (if statement)
- Call another block of code (function call)
- ...

Instruction sets

- Instruction Set Architecture: Defines the mapping between instructions and their binary format

Binary	00101000	10001001	11001000
Human-readable	mov	%rax	%rcx

- Two CPUs using the same ISA are binary-compatible
- Intel and AMD CPUs use the x86 ISA: (`x86_64`, `amd64`, `Intel64`)
- Smartphones and recent Apple CPUs use the ARM ISA

`x86_64` ISA

Instruction sets evolve over time

- 2004: SSE, SSE2 (128-bit FP vectorization)
- 2008: SSE4.2, SSE3, SSSE3
- 2011: AVX (256-bit FP vectorization, 3 operands)
- 2013: AVX2 (FMA)
- 2015: AVX-512 (512-bit FP vectorization)
- 2019: AVX-512 VNNI (Vector Neural Networks Instructions)
- 2020: AVX-512 BF16 (lower-precision bfloat16 floating-point numbers)

Assembly

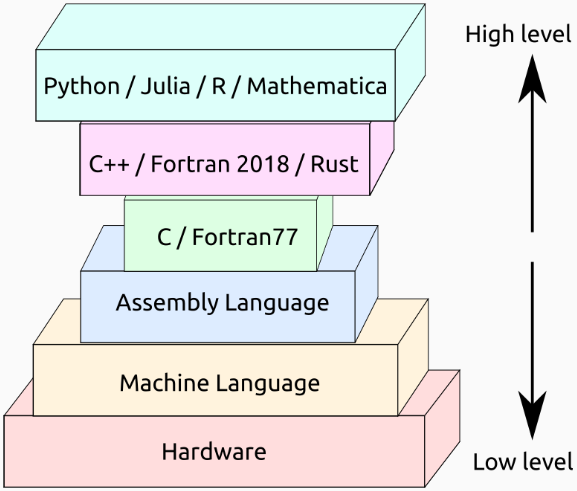
Assembly language is the human-readable form of machine language

Polynomial $a\,x^2 + b\,x + c$ in `x86_64` assembly:

```
f2 0f 59 c3        mulsd  %xmm3,%xmm0    ; x*a -> a
f2 0f 59 c3        mulsd  %xmm3,%xmm0    ; x*a -> a
f2 0f 59 d9        mulsd  %xmm1,%xmm3    ; b*x -> x
f2 0f 58 c3        addsd  %xmm3,%xmm0    ; x+a -> a
f2 0f 58 c2        addsd  %xmm2,%xmm0    ; c+a -> a
c3                 ret                   ; returns a
```

`%xmm0`: a `%xmm1`: b `%xmm2`: c `%xmm3`: x

High/low level



Summary

- A program is a sequence of instructions
- Different hardware may use different ISA
- Assembly is a human-friendly way to write binary code

The C compiler

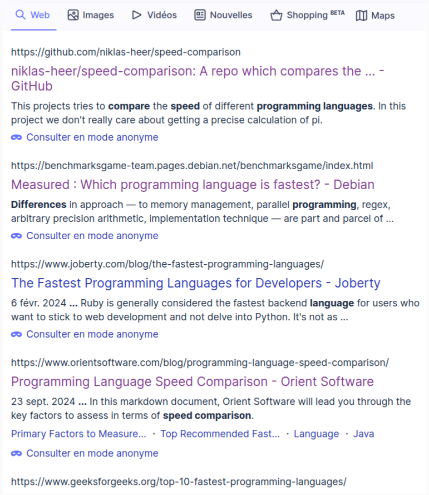
Compiler

Compiler
Computer program that translates computer code written in one programming language (the source language) into another language (the target language).

Examples

Compiler	Source	Target
NASM	x86 Assembly	Machine Language
GCC	C	Machine Language
ICX	C	Machine Language
Gfortran	Fortran	Machine Language
F2C	Fortran	C
objdump	Machine Language	Assembly
JavaC	Java	Java Bytecode
JSofOCaml	OCaml	Javascript

Does it make sense to say "a language is fast"



https://github.com/niklas-heer/speed-comparison
niklas-heer/speed-comparison: A repo which compares the ... - GitHub
This projects tries to **compare** the **speed** of different **programming languages**. In this project we don't really care about getting a precise calculation of pi.
Consulter en mode anonyme

https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html
Measured : Which programming language is fastest? - Debian
Differences in approach — to memory management, parallel **programming**, regex, arbitrary precision arithmetic, implementation technique — are part and parcel of ...
Consulter en mode anonyme

https://www.joberty.com/blog/the-fastest-programming-languages/
The Fastest Programming Languages for Developers - Joberty
6 févr. 2024 ... Ruby is generally considered the fastest backend **language** for users who want to stick to web development and not delve into Python. It's not as ...
Consulter en mode anonyme

https://www.orientsoftware.com/blog/programming-language-speed-comparison/
Programming Language Speed Comparison - Orient Software
23 sept. 2024 ... In this markdown document, Orient Software will lead you through the key factors to assess in terms of **speed comparison**.
Primary Factors to Measure... · Top Recommended Fast... · Language · Java
Consulter en mode anonyme

https://www.geeksforgeeks.org/top-10-fastest-programming-languages/

- No:
  - A language is not fast: only machine code is executed.
  - One compiler for this language produces efficient machine code.
- Yes:
  - Some languages are designed in such a way that compilers can produce very efficient code
  - Interpreted languages (Bash, Python, Ruby) are slower than compiled languages

Stages of compilation

Four main stages:

1. Preprocessing: text substitutions performed in the source code.
2. Compilation
3. Assembly
4. Linking

Stages of compilation

Four main stages:

1. Preprocessing: text substitutions performed in the source code.
2. Compilation
3. Assembly
4. Linking

Preprocessing

- `gcc -E` outputs the preprocessed source file

Directives

- `#include`: Inserts the content of a file
- `#define`: Defines a macro
- `#ifdef`, `#elif#`, `#else`, `#endif`: Conditional compilation
- `#error`, `#warning`: User-defined warnings and errors

#include

```c
#include <stdio.h>

int main(void)
{
    printf("Hello World!\n");
    return 0;
}
```

- `#include`: Inserts the content of the file `stdio.h`, which contains the definition of the `printf` function
- `#include <...>`: Search for the file into the default locations (`/include`, `/usr/include`, `C_INCLUDE_PATH`, `CPATH`, `gcc -I/path/to/includes`)
- `#include "..."`: Look for a user-defined file in the local project

Macros

PI is replaced with 3.141592653589793 before the actual compilation

```
#define PI 3.141592653589793
float circumference = 2. * PI * radius;
// float circumference = 2. * 3.141592653589793 * radius;
```

```
#define A(i,j) a[(i)+(j)*n]
#define B(i,j) b[(i)+(j)*n]
for (int j=0 ; j<n ; j++) {
  for (int i=0 ; i<n ; i++) {
    A(i,j) = A(i,j) + B(j,i);
    // a[(i)+(j)*n] = a[(i)+(j)*n] + b[(j)+(i)*n]
  }
 }
```

Conditional compilation

- Selects which branch to insert in the source code

```
#ifdef FAST
  x = fast_version_of_f(a,b,c);
#else
  x = slow_version_of_f(a,b,c);
#endif
```

```
#if DEBUG >= 2
  printf("x = %f\n", x);
#endif
```

- Macros can be defined by `#define`, or from the compilation command line: `gcc -DFAST` defines the `FAST` macro

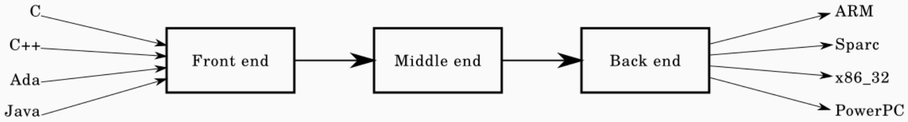
Stages of compilation

Four main stages:

1. Preprocessing: text substitutions performed in the source code.
2. Compilation
3. Assembly
4. Linking

Compilation

- Checks for syntax errors
- Translate the preprocessed source code into assembly language
- Three-stage Compiler structure



- To generate an assembly file, use `gcc -S example.c -o example.s`

Stages of compilation

Four main stages:

1. Preprocessing: text substitutions performed in the source code.
2. Compilation
3. Assembly
4. Linking

Assembly

- Converts the assembly code into the binary format
- The produced file is an object file (e.g., `example.o`)
    - `.data`: Initialized data
    - `.bss`: Uninitialized data
    - `.rodata`: Read-only data (constants)
    - Symbol table:
        - Defined Symbols: Symbols that are declared in the current file, such as function names or global variables.
        - Undefined Symbols: Symbols that are used in the current file but defined elsewhere (e.g., in a different source file or a library).
    - Relocation information: Information needed by the linker to adjust addresses and references when combining multiple object files
    - Debugging information: identifiers for functions, variables, etc.
- `gcc -c` produces an object file

Stages of compilation

Four main stages:

1. Preprocessing: text substitutions performed in the source code.
2. Compilation
3. Assembly
4. Linking

Linking

- Combines multiple object files and libraries into a single executable
- Symbol Resolution: The linker resolves symbols, which are references to variables or functions defined in different files.
- Address Binding: The linker assigns memory addresses to variables and functions and resolves external references to ensure everything points to the right location in the executable.
- Static Linking: Libraries are combined *into* the executable, resulting in a larger file that is self-contained.
- Dynamic Linking: The executable relies on external shared libraries (e.g., .so or .dll files), making the file smaller but dependent on these libraries at runtime.

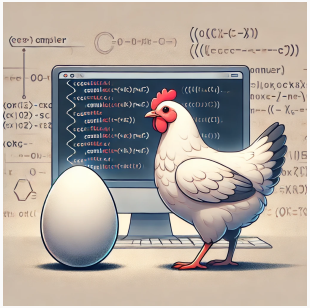
Summary

Why use a Compiler?

- Because we don't want to write assembly
- The compilers can now produce more efficient assembly than humans
- We want our code to be architecture-independent

Origins

- Gcc is written in C, ocamlc is written in OCaml, the rustc compiler is written in Rust, . . . How did the first compiler emerge?



Exercise

- Write the following function in a file dot.c

```c
#include <stdlib.h>
double my_dot_product(double* A, double* B, size_t n) {
  double result = 0.0;
  for (size_t i = 0; i < n; i++) {
    result += A[i] * B[i];
  }
  return result;
}
```

- Transform it into assembly code using

```
gcc -O0 -S -fverbose-asm dot.c -o dot_gcc_O0.S
gcc -O3 -S -fverbose-asm dot.c -o dot_gcc_O3.S
gcc -O3 -march=native -S -fverbose-asm dot.c -o dot_gcc_O3_native.S
icx -O3 -march=native -S -fverbose-asm dot.c -o dot_icc_O3_native.S
```

- Look at the different assemblies produced

Introduction to the C programming language

Hello World

- Create the file `hello.c`

```c
#include <stdio.h>
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

- Compile the file using gcc, and run the program

```
gcc hello.c -o hello
./hello
```

- `main()`: Main function, entry point of the program
- `printf()`: Formatted print to the standard output
- `#include <stdio.h>`: Required for `printf`
- `return 0`: Successful execution return code

Hello World

```
$ ./hello && echo Success || echo Failed with code $?
Hello, World!
Success
```

Change `return 0` into `return 2`

```
$ ./hello && echo Success || echo Failed with code $?
Hello, World!
Failed with code 2
```

Comments

- A comment is a piece of text that is ignored by the compiler
- Used to explain code, and to make it more readable
- Single-line comments start with two forward slashes (//)
- Multi-line comments start with /* and end with */.

Variables and Data types

Basic data types

Type	Description
`int`	signed integer able to represent at least [-32767 ; +32767]
`unsigned int`	unsigned integer able to represent at least [0 ; +65535]
`long`	at least [-2147483647 ; +2147483647]
`long long`	at least [-9223372036854775807; +9223372036854775807]
`size_t`	unsigned integer large enough to count bytes
`float`	32-bit floating point
`double`	64-bit floating point
`char`	8-bit character

Variables and Data types

Fixed-width integers

- Defined in `<stdint.h>`
- `int8_t`, `int16_t`, `int32_t`, `int64_t`: 8-, 16-, 32-, 64-bit wide signed integers
- `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`: 8-, 16-, 32-, 64-bit wide unsigned integers

Variables and Data types

Boolean variables

- `0` evaluates to `false`
- Non-zero evaluates to `true`
- Equality: `==`, non-equality: `!=`
- Comparison `<`, `>`, `<=`, `>=`
- Negation: `!`
- Logical and : `&&`, or: `||`
- Short-circuit:
    - `&&` stops evaluating at the first false expression
    - `||` stops evaluating at the first true expression

Variables and Data types

Example

```c
#include <stdio.h>
int main() {
  int    a = 5;
  float  b = 3.14;
  char   c = 'A';
  double d = 2.71828;
  printf("a=%d; b=%f; c=%c; d=%f\n", a, b, c, d);
  return 0;
}
```

Types conversion

Exercise: Check the output of the following program

```
#include <stdio.h>
int main() {
  int a = 5;
  float b = 5.;
  printf("a as int  : %d\n", a);  // Good
  printf("a as float: %f\n", a);  // Bad
  printf("b as float: %f\n", b);  // Good
  printf("b as int  : %d\n", b);  // Bad
  // With type conversions
  printf("a converted to float: %f\n", (float) a);
  printf("b converted to int  : %d\n", (int) b);
  return 0;
}
```

Printf

- See `man 3 printf`
- `\n` : End of line character
- Modifiers:

Specifier	Data Type	Example
`%d`	`int`	`printf("%d", 42);`
`%u`	`unsigned int`	`printf("%u", 42);`
`%f`	`float`	`printf("%f", 3.14);`
`%lf`	`double`	`printf("%f", 3.14);`
`%c`	`char`	`printf("%c", ’A’);`
`%s`	String	`printf("%s", "Hello");`
`%x`	`unsigned integer (hexadecimal)`	`printf("%x", 255);`
`%p`	Pointer address	`printf("%p", &variable);`

Printf

Precision and Field Width Modifiers

- Field Width: minimum number of characters to print

```
printf("%5d", 42);  // Output: "   42" (padded with spaces)
```

- Precision: number of digits to display after the decimal point for floating-point numbers.

```
printf("%.2f", 3.14159);  // Output: "3.14"
```

- Syntax: `%[width].[precision][specifier]`

```
printf("%8.3f", 3.14159);  // Output: "   3.142" (width: 8, prec: 3)
```

Scanf

- scanf is similar to printf, but for reading data
- It takes a format string that specifies the type of input values to read in:

```
#include <stdio.h>
int main() {
  int a;
  double b;
  printf("Input an integer and a double: ");
  scanf("%d %lf", &a, &b);
  printf("Read : %d %lf\n", a, b);
  return 0;
}
```

- We will see later on why a & symbol precedes the variable names in scanf.

Basic arithmetic operations

```c
#include <stdio.h>
int main() {
  int a = 5; int b = 3; int c; double d;
  c = a+b;  // Add
  printf("%d = %d + %d\n", c, a, b);
  c = a/b;  // Integer Division
  printf("%d = %d / %d\n", c, a, b);
  c = a%b;  // Modulo
  printf("%d = %d %% %d\n", c, a, b);
  d = (double) a / (double) b; // FP division
  printf("%f = %d / %d\n", d, a, b);
  return 0;
}
```

Binary operators

- << Shift left
- >> Shift right
- & Binary and
- | Binary or
- ~ Binary not
- ^ Binary xor

```
// 4 (100) shifted right 2 times = 001
printf("%x %x\n", 4, 4 >> 2);  // 4 1
// 1 (001) shifted left 2 times = 100
printf("%x %x\n", 1, 1 << 2);  // 1 4

printf("%x %x %x\n", 5, 3, 5 & 3); // 5 3 1
printf("%x %x %x\n", 2, 4, 2 | 4); // 2 4 6
printf("%x %x\n", 6, ~6); // 6 fffffff9
```

Incrementation

- `i+=1`: increments `i` by 1
- `i+=k`: increments `i` by `k`
- `i-=k`: decrements `i` by `k`
- `i++`: post-increments `i` by 1
- `i--`: post-decrements `i` by 1
- `++i`: pre-increments `i` by 1
- `--i`: pre-decrements `i` by 1

```
int i = 5;
int result = i++;
// result is assigned 5, then i becomes 6

int i = 5;
int result = ++i;
// i becomes 6 first, then result is assigned 6
```

Conditionals

```
if ( boolean_expression_1 ) {
    /* true body 1 */
}
else if ( boolean_expression_2 ) {
    /* true body 2 */
}
else {
    /* false body */
}
```

Conditionals

```c
#include <stdio.h>
int main() {
  int number = 10;
  if (number > 5) {
    printf("Number is greater than 5\n");
  } else {
    printf("Number is not greater than 5\n");
  }
  return 0;
}
```

While loop

```c
#include <stdio.h>
int main() {
  int number = 5;
  while (number < 10) {
    printf("Number is %d\n", number);
    number++;
  }
  return 0;
}
```

For loop

```
for (initialization; condition; update) {
    /* Code to be executed in each iteration */
}
```

- Initialization: Sets the starting value of the loop control variable. This step is executed once, before the loop begins.
- Condition: Evaluates whether the loop should continue running. If the condition is true, the loop executes; if false, the loop terminates.
- Update: Modifies the loop control variable after each iteration. This step happens at the end of each loop cycle.

For loop

```c
#include <stdio.h>
int main() {
  for (int i = 0 ; i < 5 ; i++) {
    printf("i = %d\n", i);
  }
  return 0;
}
```

- `int i = 0`: Declares a local integer variable `i`, initialized equal to zero
- `i < 5`: if i<5 is true, iterate
- `i++`: Increment `i` at the end of each iteration
- A for loop can be exited before the termination condition is reached with the `break` statement

For loop

Exercise:

- Write a program that displays the prime numbers in the [0;200] range

For loop (Solution)

```c
#include <stdio.h>
int main() {
  for (int num=2 ; num<201 ; num++) {  // 0 and 1 are not prime numbers
    int is_prime = 1;  // Assume num is prime until proven otherwise
    for (int i = 2; i < num; i++) { // Check for divisors
      if (num % i == 0) {
        is_prime = 0;  // num has a divisor >=2
        break;  // Exit loop over i
      }
    }
    if (is_prime) {
      printf("%d\n", num);
    }
  }
}
```

Functions

```c
#include <stdio.h>

double divide(int a, int b) {
  return (double) a / (double) b;
}

int main() {
  double c = divide(3, 4);
  printf("divide(3,4) = %f\n", c);
  return 0;
}
```

- `divide` has to be specified before it is used
- The arguments `(3,4)` are copied into variables `a` and `b` (pass by *value*)

Functions

```c
#include <stdio.h>

double divide(int a, int b);   // Function Prototype

int main() {
  double c = divide(3, 4);
  printf("divide(3,4) = %f\n", c);
  return 0;
}

double divide(int a, int b) {    // Function defined after being used
  return (double) a / (double) b;
}
```

Functions

my_program.c

```c
#include <stdio.h>

// Function Prototype:
double divide(int a, int b);

int main() {
  double c;
  c = divide(3, 4);
  printf("divide(3,4) = %f\n", c);
  return 0;
}
```

divide.c

```c
double divide(int a, int b) {
  return (double) a / (double) b;
}
```

Compile as:

```
gcc -o divide my_program.c divide.c
```

Functions

my_program.c

```c
#include <stdio.h>
#include "my_arith.h"

int main() {
  double c;
  c = divide(3, 4);
  printf("divide(3,4) = %f\n", c);
  return 0;
}
```

divide.c

```c
double divide(int a, int b) {
  return (double) a / (double) b;
}
```

my_arith.h

```c
double divide(int a, int b);
```

Compile as:

```
gcc -o divide my_program.c divide.c
```

Back to compilation

Compiling multiple files

- Generate the executable

```
gcc  -o divide  my_program.c  divide.c  # produces divide
```

- First generate objects, then link

```
gcc  -c my_program.c          # produces my_program.o
gcc  -c divide.c              # produces divide.o
gcc  my_program.o  divide.o  # produces divide
```

- Compiling files into objects:
  - Recompiling only what has changed
  - Simultaneous compilation of independent files $\Longrightarrow$ speed

Example

- Similarly to `divide.c`, we can create `add.c`, `subtract.c`, and `multiply.c`

`add.c`

```c
double add(int a, int b) {
  return (double) a + (double) b;
}
```

`divide.c`

```c
double divide(int a, int b) {
  return (double) a / (double) b;
}
```

`subtract.c`

```c
double subtract(int a, int b) {
  return (double) a - (double) b;
}
```

`multiply.c`

```c
double multiply(int a, int b) {
  return (double) a * (double) b;
}
```

Example

- In the header file my_arith.h, we insert the prototypes of all the functions, with some comments explaining what the functions do:

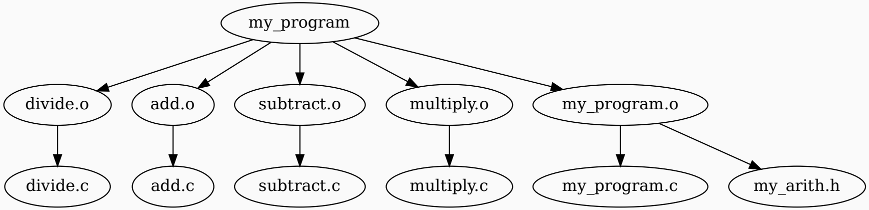
```
double divide (int a, int b);
// Returns the FP-division of a and b

double add (int a, int b);
// Returns the FP-addition of a and b

double subtract (int a, int b);
// Returns the FP-subtraction of a and b

double multiply (int a, int b);
// Returns the FP-multiplication of a and b
```

Example

- In the main file, `#include "my_arith.h"`
- Compile all the files into objects, and link them:

```
#!/bin/bash

SOURCE_FILES="add.c subtract.c multiply.c divide.c my_program.c"
OBJECT_FILES=${SOURCE_FILES//.c/.o}

for FILE in $SOURCE_FILES ; do
    gcc -c $FILE
done
gcc  -o my_program  $OBJECT_FILES
```

Dependencies



Makefile

We can create a Makefile:

```
CC=gcc
SRC=add.c subtract.c multiply.c divide.c my_program.c
OBJ=$(patsubst %.c, %.o, $(SRC))

my_program: $(OBJ)
        $(CC) -o my_program $(OBJ)

%.o: %.c                # All .o files depend on the corresponding .c file
        $(CC) -c $<     # $< is the name of the .c file

my_program.o: my_arith.h my_program.c
```

Makefile

- Syntax:

```
target: list of dependencies
        command
```

- Running the `make` command will compile all the files in the correct order, and link the objects.
- If a file is modified, only what is necessary will be recompiled
- Try the following with the Makefile of the previous slide

```
make
make
touch add.c ; make
touch my_arith.h subtract.c ; make
```

Static Library

A library is a binary file that contains a collection of functions.

Let's create a static library:

```
ar  rcs  libmy_arith.a  add.o subtract.o multiply.o divide.o
```

- `ar` (archive) bundles multiple object files into a static library `libmy_arith.a`
- `r`: Inserts or replaces the object files into the archive
- `c`: Creates the library if it does not exist.
- `s`: Generates an index for faster linking.

Static Library

- Compile the code with the library:

```
gcc  my_program.c  libmy_arith.a
```

  or

```
gcc  my_program.c  -L.  -lmy_arith
```

- `-L.`: Search libraries in the current directory
- `-lmy_arith`: link with library named `libmy_arith.a`
- The library and the main program are now linked together, as before

Shared Library

Let's create a shared (or dynamic) library:

- Compile the source files with the `-fPIC` (Position-Independent Code) option.

```
for FILE in add.c subtract.c multiply.c divide.c ; do
    gcc -fPIC -c $FILE
done
```

- Link with the `-shared` option, and the `.so` suffix

```
gcc  -shared  -o libmy_arith.so  add.o subtract.o multiply.o divide.o
```

Shared Library

- Compile the code with the library:

```
gcc my_program.c -L. -lmy_arith
```

- `-L.`: Search libraries in the current directory
- `-lmy_arith`: link with library named `libmy_arith.so`
- The library is not embedded in the program, but stays separated. It will be loaded dynamically when the program executes.
- When the program is executed, it will search for `libmy_arith.so` in the default system paths (`/lib`, `/usr/lib`, . . .) and in the `LD_LIBRARY_PATH` environment variable.

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PWD
./my_program
```

Shared Library

- You can put all your dynamic libraries in a special location (`$HOME/.local/lib`) and add `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$HOME/.local/lib` to your `~/.bashrc` file.
- As the library functions are not copied inside the exectuable, it is possible to update the library without recompiling the binary

Shared Library application 1

- Modify the divide function to handle division by zero
- Recompile the library:

```
gcc  -fPIC  divide.c
gcc  -shared  -o libmy_arith.so  add.o subtract.o multiply.o divide.o
```

- Run again the code
- If you have multiple codes linked with `libmy_arith.so`, all of them will be updated automatically

Shared Library application 2

- You create a program that can make a plot of any function defined as

```
double to_plot(double x);
```

- You let the users write the function `to_plot` into a shared library
- They can now use your code to plot their function, without recompiling your code

Shared Library application 3

- Your code uses the Zlib library for compression
- A security breach was found in Zlib, and the developers release an important update affecting important services
- So you update your system
- Your program will now use the updated library, without the need to recompile

Summary

- Different functions can be put in different files
- Makefiles can handle dependencies in compilation to recompile only what is necessary
- Functions can be grouped into libraries for re-use in other programs

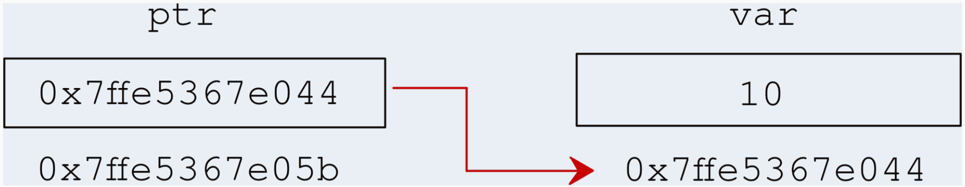
Back to C

Pointers

- A pointer is a variable that contains as its value the memory address of another variable.
- The type of a pointer variable is the type of the data it points to followed by a `*` : a pointer to an `int` variable has type `int*`
- The address of a variable is obtained with the `&` operator (reference): `&a` denotes the address of variable `a`
- To get the value a pointer points to, we use the `*` (dereference) operator: `*b` returns the value stored at address contained in pointer `b`
- The `NULL` value points to the address `0x0`, which is invalid. It is often used to specify that a pointer points to nothing
- Pointing to an invalid address generates a Segmentation Fault

Pointers examples

```c
int  var = 10;
int* ptr = &var;

printf("%d",  var);  // Outputs the value of var (10)
printf("%p", &var);  // Outputs the memory address of var (0x7ffe5367e044)
printf("%p",  ptr);  // Outputs the value of ptr (0x7ffe5367e044)
printf("%d", *ptr);  // Outputs the value pointed by ptr (10)
```



Arrays

- An array is a collection of values with the same data type.
- It is declared using the [] operator
- The m-th element of the array has index i=m-1 and is accessed using [i]

```c
double v[3];
for (int i=0 ; i<3 ; i++) {
  v[i] = (double) i / 3.;
  printf("v[%d] = %f\n", v[i]);
}
```

- Warning: The first element is v[0] and the last element is v[2]

Arrays

- Initialization:

```
double v[3] = {1.0, 2.0, 3.0};   double v[]  = {1.0, 2.0, 3.0};
```

- In memory, the values of an array are stored in contiguous addresses: `&v[i] = &v[i-1] + 1`
- The array name alone is equivalent to the address of the array: `&v[0] == v`

```
double v[3]; double *p = v;
for (int i=0 ; i<3 ; i++) {
  v[i] = (double) i / 3.;
  printf("v[%d] = %f\n", *(p+i));
}
```

- Adding an integer `n` to a pointer `p` of type `t*` increments the address by `n*sizeof(t)` bytes

2-dimensional arrays

- It is an array of arrays:

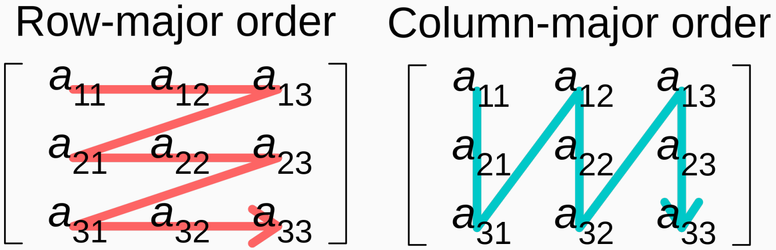
```
int arr[2][4] = {
    {10, 11, 12, 13},
    {14, 15, 16, 17}
};
```

- `arr` is an array with 2 elements, each one being an `int` array with 4 elements
- `arr[i][j]` picks the (j-1)-th element of array `arr[i]`, which is the (i-1)-th element of array `arr`
- As arrays are stored with contiguous addresses:

```
{{10, 11, 12, 13}, {14, 15, 16, 17}} == {10, 11, 12, 13, 14, 15, 16, 17};
```

Row/column major

- In a matrix, A_{ij} represents the element of row i and column j
- When matrices are stored in 2D-arrays, we often *decide* that `a[i][j]` $= A_{ij}$
- In that case, the rows of the matrices are contiguous in memory, and the matrix is stored with row-major ordering
- Row-major ordering is natural in C, while column-major ordering is natural in Fortran


Row-major order
Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Strings

- Strings are arrays or characters, terminated with a NULL character ('\0')
- The size of a string needs to be dimensioned with one more character for the NULL character

```
char[] mystring = "hello";
```

- Copying a string is not as simple as `a=b`. Use `strncpy`.

Memory layout

Memory is divided into several regions with distinct purposes:

- Code Segment: Stores the program's compiled instructions (read-only).
- Global/Static Segment: Stores global and static variables.
- Stack: Stores function-local variables and manages function calls.
- Heap: Used for dynamically allocated memory during runtime.

The stack

The Stack: A region of memory that stores temporary data for function calls.

- Holds local variables and function parameters
- Each function call creates a new *stack frame* for its variables
- Automatically Managed: Memory is allocated and deallocated automatically as functions are called and return
- Fixed Size: Limited by system settings; can lead to a stack overflow if too large

The heap

The Heap: A region of memory used for dynamic memory allocation.

- Allows flexible memory management with `malloc` and `free` functions
- Suitable for variables whose size or lifetime is not known at compile time
- Manually Managed: The programmer must allocate and free memory
- Flexible Size: Limited by the system's available memory, but improper management can cause memory leaks

Memory allocation on the heap

- The `malloc` system call asks the system to allocate a block of memory
- The argument of malloc is a number of bytes (of type `size_t`)
- It returns a pointer to the allocated block upon success, of a `NULL` pointer upon failure
- The allocated memory is *not* initialized
- The memory is de-allocated using the `free` function

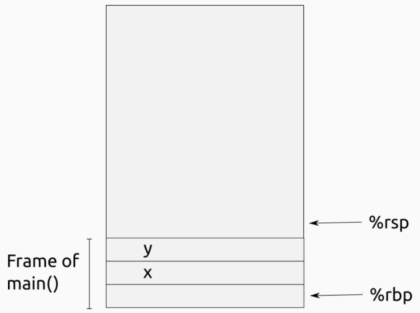
Memory allocation on the heap

```c
#include <stdlib.h> // Required for malloc

double* data = malloc(10*sizeof(double)); // Request 10 doubles
if (data == NULL) {   // Always check that the allocation was OK
    printf("Allocation failed\n");
    exit(-1);
}
for (size_t i=0 ; i<10 ; i++) {  // Initialize the array
    data[i] = (double) i+1;
}
free(data);    // Always free when not used anymore
data = NULL;   // Always set a freed pointer to NULL
```

Function calls

Stack

- Region of memory that grows and shrinks during execution
- Used for managing environments (local variables)



Example

```c
double polynomial(double x) {
  double p[3] = {3, 4, 5};
  double result;
  result = p[0]*x*x + p[1]*x + p[2];
  return result;
}

int main() {
  double x, y;
  ...                  // <-Instr ptr
  y = polynomial(x);
  printf("%f\n", y);
}
```

Function calls

Setting up the call stack by pushing onto the stack

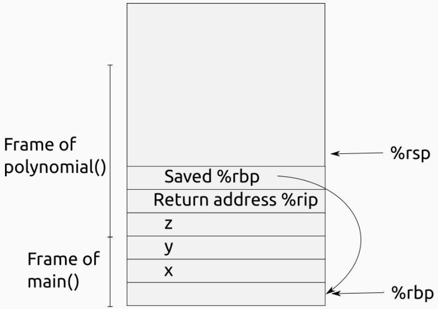
- Function arguments
- The return address: next instruction after the call
- The stack frame pointer: base pointer (%rbp) allowing to restore the calling function's stack frame upon return

Example

```c
double polynomial(double z) {
  double p[3] = {3, 4, 5}; // <-Stack frame ptr
  double result;
  result = p[0]*z*z + p[1]*z + p[2];
  return result; // Go to next instr after call
}

int main() {
  double x, y;
  ...
  y = polynomial(x);  // <-Instr ptr
  // Next instr after call: copy result in y
  printf("%f\n", y);
}
```

Function calls

Setting up the call stack by pushing onto the stack



Example

```
double polynomial(double z) {
  double p[3] = {3, 4, 5}; // <-Stack frame ptr
  double result;
  result = p[0]*z*z + p[1]*z + p[2];
  return result; // Go to next instr after call
}

int main() {
  double x, y;
  ...
  y = polynomial(x);  // <-Instr ptr
  // Next instr after call: copy result in y
  printf("%f\n", y);
}
```

Function calls

Transfer control, and set up the function's stack frame

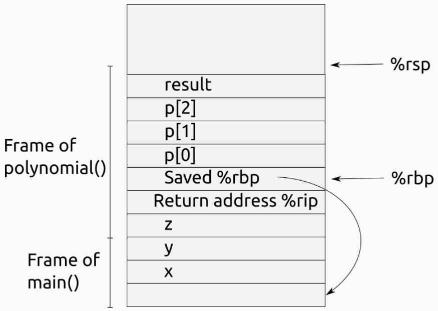
- `CALL` instruction which changes the instruction counter to point to the called function's starting address
- Set the new base pointer `%rbp = %rsp`
- Move the stack pointer (`%rsp`) to allocate space for the local variables

Example

```
double polynomial(double z) {
  double p[3] = {3, 4, 5}; // <-Stack frame ptr
  double result;
  result = p[0]*z*z + p[1]*z + p[2]; // <-Instr p
  return result; // Go to next instr after call
}

int main() {
  double x, y;
  ...
  y = polynomial(x);
  // Next instr after call: copy result in y
  printf("%f\n", y);
}
```

Function calls

Transfer control, and set up the function's stack frame



Example

```
double polynomial(double z) {
  double p[3] = {3, 4, 5}; // <-Stack frame ptr
  double result;
  result = p[0]*z*z + p[1]*z + p[2]; // <-Instr p
  return result; // Go to next instr after call
}

int main() {
  double x, y;
  ...
  y = polynomial(x);
  // Next instr after call: copy result in y
  printf("%f\n", y);
}
```

Function calls

Execute function, and return

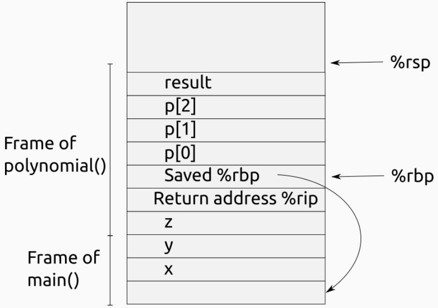
- Store return value (`%rax`)
- Deallocate local variables: set `%rsp` to `%rbp`
- Restore `%rbp` to the caller's one
- Set the next instruction to `%rip`
- Clean up the stack: Move `%rsp` back to the top of the stack by popping function arguments

Example

```
double polynomial(double z) {
  double p[3] = {3, 4, 5}; // <-Stack frame ptr
  double result;
  result = p[0]*z*z + p[1]*z + p[2]; // <-Instr p
  return result; // Go to next instr after call
}

int main() {
  double x, y;
  ...
  y = polynomial(x);
  // Next instr after call: copy result in y
  printf("%f\n", y);
}
```

Function calls

Execute function, and return

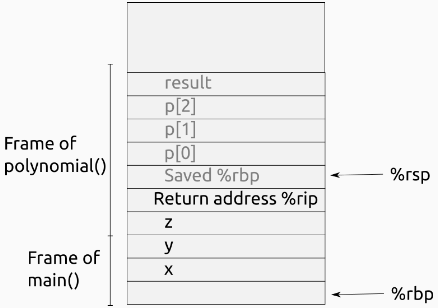


Example

```
double polynomial(double z) {
  double p[3] = {3, 4, 5}; // <-Stack frame ptr
  double result;
  result = p[0]*z*z + p[1]*z + p[2]; // <-Instr p
  return result; // Go to next instr after call
}

int main() {
  double x, y;
  ...
  y = polynomial(x);
  // Next instr after call: copy result in y
  printf("%f\n", y);
}
```

Function calls

Execute function, and return

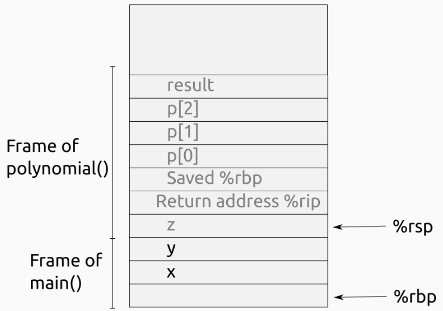


Example

```
double polynomial(double z) {
  double p[3] = {3, 4, 5}; // <-Stack frame ptr
  double result;
  result = p[0]*z*z + p[1]*z + p[2]; // <-Instr p
  return result; // Go to next instr after call
}

int main() {
  double x, y;
  ...
  y = polynomial(x);
  // Next instr after call: copy result in y
  printf("%f\n", y);
}
```

Function calls

Execute function, and return



Example

```
double polynomial(double z) {
  double p[3] = {3, 4, 5}; // <-Stack frame ptr
  double result;
  result = p[0]*z*z + p[1]*z + p[2]; // <-Instr p
  return result; // Go to next instr after call
}

int main() {
  double x, y;
  ...
  y = polynomial(x);
  // Next instr after call: copy result in y
  printf("%f\n", y);
}
```

Passing by value

- Recall that for `scanf`, we passed the addresses of the arguments:

```
scanf("%d %lf", &a, &b);
```

- When calling a function, the arguments are passed by value: they are copied in the function, so they can't be modified
- To modify a variable, the value passed to the function has to be the *address* of the variable
- Note: In Fortran, the variable are by default passed by address

Formatted I/O

Writing

```
#include <stdio.h>

FILE* output_file = fopen("output.txt", "w");  // Open output.txt for writing

if (output_file == NULL) {
  printf("Error opening input file for reading");
  exit(-1);
}

for (int i=0 ; i<10 ; i++) {
  fprintf(output_file, "%d %lf\n", i, (double) i);  // Write into output_file
 }

fclose(output_file);
output_file = NULL;
```

Formatted I/O

Reading

```c
#include <stdio.h>
FILE* input_file = fopen("input.txt", "r");  // Open input.txt for reading
if (input_file == NULL) {
  printf("Error opening input file for writing");
  exit(-1);
}

int return_code = 0;
while (1) {
  int i; double x;
  return_code = fscanf(input_file, "%d %lf\n", &i, &x);  // Read from input_file
  if (return_code == EOF) break;
  printf("%d %lf\n", i, x);
 }
fclose(input_file);
```

Binary I/O

- Formatted I/O is slow: It takes time to make the binary-decimal conversions
- Binary I/O takes the binary representation and writes it directly in the file

```c
#include <stdio.h>
#define N 1000
double data[N];

FILE* input_file = fopen("input.bin", "rb");  // Open input.bin for reading in binary mode
size_t fread(data, sizeof(double), N, input_file);
fclose(input_file);

FILE* output_file = fopen("output.bin", "wb");  // Open output.bin for writing in binary mode
size_t fwrite(data, sizeof(double), N, output_file);
fclose(output_file);
output_file = NULL;
```

Structures

- A struct is a user-defined type which contains a collection of data of same or different types:

```
struct atom_struct {
  char* Symbol;
  unsigned int atomic_number;
  double mass;
  double coordinates[3];
};
```

Structures

```
struct atom_struct helium;
helium.Symbol = "He";
helium.atomic_number = 2;
helium.mass = 4.002;
helium.coordinates = { 0. ; 1. ; 0. };

struct atom_struct * molecule;
molecule = malloc(4*sizeof(struct atom_struct));
```

- In memory, the elements of the struct have contiguous addresses `&helium == &(helium.Symbol)`

Structures

With pointers, `(*a).x` can be rewritten as `a->x`

```
struct atom_struct* helium = malloc(sizeof(struct atom_struct));
helium->Symbol = "He";
helium->atomic_number = 2;
helium->mass = 4.002;
helium->coordinates = { 0. ; 1. ; 0. };
```

Problems with structures

Data alignment

- Consider the struct

```
struct mystruct {
  char c;
  double d;
};
```

- The struct will be allocated on an 8-byte boundary address
- c will be on a 1-byte boundary (as it is on an 8-byte boundary)
- d will be misaligned, as it will be 1-byte away from an 8-byte boundary
- To improve the data access, it is preferable to to put d before c such that they both have the proper alignment

Problems with structures

Data alignment

- Consider the struct:

```
struct mystruct {
  double d;
  char c;
};
```

- If we make an array of those, the elements of the first struct will be well aligned, but the second struct will be mis-aligned (as the struct is 9 bytes long

- To solve this problem, we can use padding: adding dummy variables to make the struct a multiple of 8 bytes

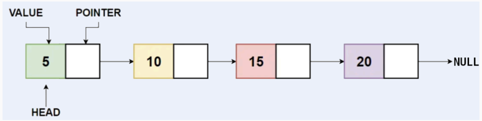
```
struct mystruct {
  double d;
  char c;
  char padding[7];
};
```

Problems with structures

- To optimize memory access, the data should be ordered by decreasing size of the data types:
    - 8-byte variables: `double`, `int64_t`, ...
    - Pointers and `size_t`: `double*`, `int64_t*`, `char*`, `float*`, ...
    - 4-byte variables: `float`, `int32_t`, ...
    - 1-byte variables: `char`, ...
- Otherwise, the elements will be misaligned in memory

Structure example: Linked list

Linked list



Each element contains

- data
- a pointer to the next element

```
typedef struct element {
  struct element* next;
  int data;
} elem;

elem* head = malloc(sizeof(elem));
head->data = 5; head->next = NULL;

elem* second = malloc(sizeof(elem));
second->data = 10; second->next = NULL;
head->next = second;
```

Recursive functions

A recursive function is a function that calls itself. For example:

$$N! = \begin{cases} 1 & \text{if } N = 0 \\ N \times (N-1)! & \text{otherwise} \end{cases}$$

Exercises:

- Write the factorial function, and a program that displays the first 10 values

$$1!, \dots, 10!$$

- Write a recursive function that searches for the first negative integer in a liked list, and returns 0 if not found

Recursive functions (factorial)

Solution

```c
#include <stdint.h>
#include <stdio.h>
uint64_t factorial(uint64_t n) {
  if (n == 0) return 1;
  else return n * factorial(n-1);
}

int main() {
  for (int i=0 ; i<10 ; i++) {
    printf("%d : %ld \n", i, (long) factorial(i));
  }
}
```

Recursive functions (linked list)

Solution

```
int search(elem* e) {
  if (e == NULL) {
    return 0;
  }
  else if (e->data < 0) {
    return e->data;
  }
  else {
    return search(e->next);
  }
}
```

Functions with nothing to return

```c
void my_function() {
  printf("Hello!\n");
}

int main() {
  my_function();   // Call the function with no lvalue
  return 0;
}
```

- A function that returns nothing has a `void` return type
- It is the equivalent of a `subroutine` in Fortran
- Instead of returning `void`, you can return some extra information to inform the user that the function succeeded or failed. For example, `printf` returns the number of characters printed (but this is rarely checked).

A few tips for malloc/free

- If you make a function that allocates some memory and returns an allocated object, the caller will need to know that it has to take care of freeing this memory (bad!)
- It is very hard to ensure that there is no memory leak when arrays are allocated in some functions and freed somewhere else.
- Whenever possible, allocatate arrays and free them int the same function
- If you have a return statement in the middle of a function, remember to free the allocated arrays before you return:

```
a = malloc(...);
...
if (condition) {
   free(a);
   return result;
}
...
```

Returning arrays

You may need to write functions that return an array. For this, you should

- Allocate an array for the result in the calling function
- Pass this array to the function

It is best to always pass the dimensions of the array to the functions so the function can check that the array is large enough to hold the data.

```
#include <string.h>

char* strcpy (char* dest, const char* src);
char* strncpy(char* dest, const char* src, size_t n);
```

`strncpy` can check that it writes in the bounds of the destination array, but `strcpy` can't.

References

References

- Dive into systems
- W3Schools C
- Introduction to x86 assembly
- What Every Programmer Should Know About Memory
- Computer Organization and Design
- CERT C Coding standard
- Wikipedia
- Man pages