Why C?

- It allows to
  - directly interface with the kernel (through syscalls) and with the hardware
  - have direct control of the memory
  - (performance)

- It is the language in which most of (all?) the operating systems' kernels are written

- Drawbacks: very easy to make (hard-to-find) mistakes
  - In particular, memory corruption!
  - Any complex software written in C that is completely bug-free (and secure)?

Hello World

```c
#include <stdio.h>

int main (int argc, char* argv[]){
	printf( "Hello world!\n" );
	return 0;
}
```

C vs Java

- They have a similar syntax

- Both C and Java have functions/methods, but C is not Object-Oriented

- Manual handling of memory
  - Manual allocation and deallocation (no garbage collector)
  - Manual handling of strings
  - No array boundary checks
  - Pointers (instead of object references)

- C is compiled to an OS-specific format

- C for Java Programmers by George Ferguson @ URCS: https://www.cs.rochester.edu/u/ferguson/

Compilation

- Use the text editor you prefer to edit source files
- Compile:

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

- Run:

  ```
  ./hello
  ```

- Compile & Run:

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

- Alternative, use an IDE, such as Eclipse CDT
  http://www.eclipse.org/downloads/packages/release/photon/r/eclipse-ide-cc-developers

Variable Declaration

```
int global;
int function(){
        int local;
    float f;
    int a = 32;
    char c = 'a';
        ...
}
```

- Function/global scope

Types (in Linux 64bit)

short	2 byte
int	4 byte
long	8byte
float	4 byte
double	8 byte
char	1byte

sizeof(<type>) returns the size (in bytes) of a type

short, int, long, char can be signed (default) or unsigned

For instance:

(signed) int: -2,147,483,648 to 2,147,483,647 (included)

unsigned int: 0 to 4,294,967,295 (included)

Arrays

```
int a[5];
int numbers[] = { 1, 2, 3 };
char letters[] = { 'a', 'b', 'c', 'd'};

letters[2] = 'x';
```

- There is no way to know the size of an array

printf

```
printf( <format_string>, [<variables>, ...] );
```

```
int var;
var = 5;
printf("The value of var is: %d", var);
→ The value of var is: 5
```

```
    int age = 20;
char name[] = "Chris";
printf("Hello %s, your age is: %d", name, age);
→ Hello Chris, your age is: 20
```

printf

```
printf("The value of var is: %d", var);
```

%d	int
%lu	unsigned long
%c	char
%s	string (char* or char[])
%lx	unsigned long, printed using hexadecimal notation
%p	Memory address (pointer), printed using hexadecimal notation

- Many many options: https://en.wikipedia.org/w/index.php?title=Printf_format_string#Format_placeholder_specification

Defining structures and new types

```
#include <stdio.h>
struct point {
        int x;
        int y;
};

typedef struct line {
        int x1;
        int y1;
        int x2;
        int y2;
} line_type;
```

```
int main (int argc, char* argv[]){
    struct point p;
    p.x = 5;   p.y = 7;
    printf("coordinates %d %d\n", p.x, p.y);

    line_type newline;
    printf("the size of the type 'line_type' is
        %d byte(s)\n", sizeof(line_type));
}
```

output:

```
coordinates 5 7
the size of the type 'line_type' is 16 byte(s)
```

Functions

- It is necessary to write the function's prototype at the beginning of the source file
- In function declaration/prototype, you need to specify the types of:
  - Parameters
  - Return value (void means no return value)

- Primitive Types (and user-defined struct) are passed by copy
- Arrays are passed by reference
- Use pointers to pass any variable by reference

Function Prototypes

```
#include <stdio.h>
typedef struct Point {
	int x;
	int y;
} point_type;

void print_point(point_type);
void change_array(int[], int, int);
void print_int_array(int[], int);

int global_variable = 10;
```

Functions

```
int global_variable = 10;

int main (int argc, char* argv[]){
	point_type p;
	p.x = 5; p.y = 7;
	print_point(p);
	global_variable++;

	printf("global_variable: %d\n",
				global_variable); //12

	int a[] = {10, 11, 12};
	change_array(a, 1, 33);
	print_int_array(a, 3);

	//now a = {10, 33, 12}
}
```

```
void print_point(point_type p){
	global_variable++;
	printf("Point coordinates:
		x=%d, y=%d\n", p.x, p.y);
}

void change_array(int array[],
			int index, int new_value){
	array[index] = new_value;
}

void print_int_array(int array[],
			int length){
	for(int i=0; i<length; i++){
		printf("index array[%d]=%d\n",
			i,array[i]);
	}
}
```

Passing Arguments by Value – Pitfalls

```
#include <stdio.h>
void swap_variables(int, int);

int main (int argc, char* argv[]){
        int a = 10;
        int b = 20;
        printf("a=%d, b=%d\n", a, b); //a=10, b=20
        swap_variables(a, b);
        printf("a=%d, b=%d\n", a, b); //a=10, b=20
}

void swap_variables(int a, int b){
        int tmp = b;
        b = a;
        a = tmp;
}
```

Pointers

A pointer is a variable containing a memory address (typically of another variable)

Declaration:

```
int* pointer;
```

Assign a value:

```
pointer = &variable; //& means: address of
```

Access pointed value:

```
*pointer = 11; //* means: follow the pointer value
```

Pointers

Variable Name	Address	Value
...	...	...
var1	db28	1
var2	db2c	2
pointer	db30	...
...	...	...

Pointers

Variable Name	Address	Value
...	...	...
var1	db28	1
var2	db2c	2
pointer	db30	db28
...	...	...

```
pointer = &var1;
```

Pointers

Variable Name	Address	Value
...	...	...
var1	db28	11
var2	db2c	2
pointer	db30	db28
...	...	...

```
*pointer = 11;
```

Pointers

Variable Name	Address	Value
...	...	...
var1	db28	11
var2	db2c	2
pointer	db30	db2c
...	...	...

pointer = &var2;

Pointers

Variable Name	Address	Value
...	...	...
var1	db28	11
var2	db2c	21
pointer	db30	db2c
...	...	...

```
*pointer = 21;
```

Passing Arguments by Reference using Pointers

```
#include <stdio.h>
void swap_variables(int*, int*);

int main (int argc, char* argv[]){
        int a = 10;
        int b = 20;
        printf("a=%d, b=%d\n", a, b); //a=10, b=20
        swap_variables(&a, &b);
        printf("a=%d, b=%d\n", a, b); //a=20, b=10
}

void swap_variables(int* a, int* b){
        int tmp = *b;
        *b = *a;
        *a = tmp;
}
```

Passing Arguments by Reference using Pointers

```
#include <stdio.h>


int main (int argc, char* argv[]){
        int age;
        printf("Enter your age:");
        scanf("%d", &age); //<------ notice the &
        //now age has the value inserted by the user
        printf("You inserted %d as your age\n", age);
}
```

malloc and free

We can manually allocate memory using malloc
If we want memory for 10 integers:

int* integers = (int*) malloc(10*sizeof(int));

malloc formally returns a void* pointer, we need to cast it

No garbage collector, we need to manually use free:
free(integers);

Pointers and Array

Arrays and pointers are similar:
array == &(array[0])

Suppose that:
int index;
int array[10];
    int* pointer = array;

Now pointer points to
    the first element of array

Therefore (using pointer arithmetics):
pointer+index == &(array[index])   Gives address
*(pointer+index) == array[index]   Gives access to value

Pointer Arithmetic	Address	Array Indexing
pointer+0	db00	&(array[0])
pointer+1	db04	&(array[1])
pointer+2	db08	&(array[2])
...	...	...

↕ sizeof(int)==4

Arrow Operator   ->

```
#include <stdio.h>
#include <stdlib.h> //required for malloc and free
typedef struct Point {
        int x;
        int y;
};

int main (int argc, char* argv[]){
        Point * point = (Point *) malloc(sizeof(Point));
        point->x = 5; //same as (*point).x
        printf("%d %d\n", point->x, (*point).x); //5 5

        //printf("%d %d\n", *point.x, *(point.x)); //error!
        free(point);
}
```

Strings

A string is an array of characters terminated by the NULL byte: '\x00'

A NULL byte is automatically added to string literals (such as char* a = "string")

If using malloc, remember to allocate 1 byte more for the terminator!

Use already implemented string functions:
strcat, strlen, strncpy, ...

Strings

```
//...
#include <string.h>
int main (int argc, char* argv[]){
	char* string = "string";
	int string_size = strlen(string);
	printf("The string size is: %d\n", string_size);

	char* string_copy;
	string_copy = (char*) malloc(string_size+1);

	for(int index=0; index<(string_size+1); index++){
		char cc = string[index];
		printf("%d: %02x\n", index, cc);
		string_copy[index] = string[index];
	}
}
```

Files

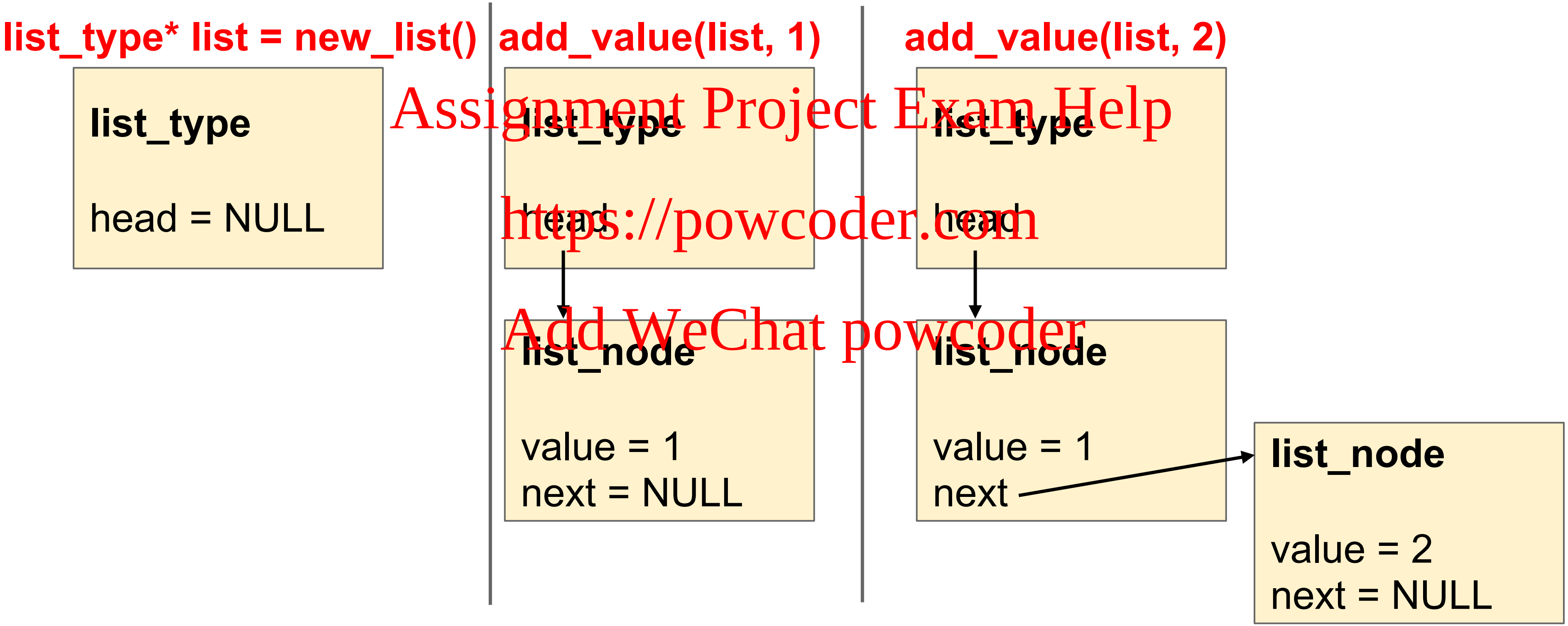
C offers functions to read/write files:
fopen, fclose, fread, …

You can also access low-level OS syscalls:
open, read, write, …

Linked Lists

A data structure to store an arbitrary amount of ordered elements

We need to “manually” handle it with pointers



list_type* list = new_list()

list_type

head = NULL

add_value(list, 1)

list_type

head

list_node

value = 1

next = NULL

add_value(list, 2)

list_type

head

list_node

value = 1

next

list_node

value = 2

next = NULL

gdb: a command-line debugger

To use it easily, compile your program with no optimizations and with debugging information:

```
gcc -O0 -ggdb -o program program.c
gdb ./program
```

r: run
b: set breakpoint
c: continue
n: execute next line
print variable: print the current value of variable
bt: backtrace (print call stack)