The C Programming Language

AIV PROG3: Roberto De Ioris

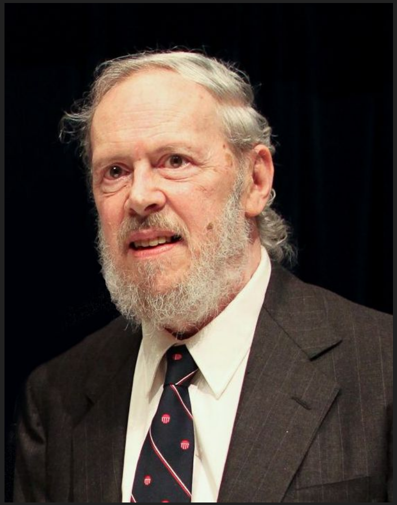
Part 0x00: History

A Language for creating UNIX (1972)

- Developed by Dennis Ritchie between 1972 and 1973 at Bell Labs
- It is a successor of the ‘B’ language
- Its original purpose was quick development of UNIX operating system tools
- After its implementation was mature enough, they rewrote the UNIX kernel with C
- The original implementation was for the PDP-11 (https://en.wikipedia.org/wiki/PDP-11_architecture)
- Since 1989 it is a standard (by both ANSI and ISO)
- In the 80’s became very popular as a way to increase productivity in areas where assembly was the only choice (like games ;)
- Most of the modern languages are based on its syntax or are written in C themselves ;)
- Even after 50 years, it is probably one of the most important programming language (for good and bad reasons)

Dennis Ritchie (1941-2011)

- With Ken Thompson created UNIX
- Created the first version of the C Language
- With Brian Kernighan wrote the book “The C Programming Language” (known as K&R)



C Language features/characteristics

- Procedural general-purpose
- Statically (weakly) typed
- Very easy to learn, very hard to master
- While considered a low-level language, it is technically (and definitely) a high level one (compared to Assembly)
- Can basically run everywhere (there are compilers even for the 6502!)
- Can embed assembly
- Basically all of the programming languages out there have a way for calling C functions.

Part 0x01: Syntax

The “Function” abstraction

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

int useless_multiplier(int value0, int value1)
{
    return value0 * value1;
}

void do_math()
{
    int total = useless_adder(1, 2, 3) + useless_multiplier(0x10, 0x20);
    total += 1;
    total = useless_multiplier(useless_adder(0xA, 0xB, 0xC), 0x17);
}
```

Declaration before use!

```
int useless_adder(int a, int b, int c);
int useless_multiplier(int value0, int value1);

void do_math()
{
    int total = useless_adder(1, 2, 3) + useless_multiplier(0x10, 0x20);
    total += 1;
    total = useless_multiplier(useless_adder(0xA, 0xB, 0xC), 0x17);
}

int useless_adder(int a, int b, int c)
{
    return a + b + c;
}

int useless_multiplier(int value0, int value1)
{
    return value0 * value1;
}
```

The 32 Keywords (magenta: types, cyan: very rare)

- auto
- break
- case
- char
- const
- continue
- default
- do
- double
- else
- enum
- extern
- float
- for
- goto
- if
- int
- long
- register
- return
- short
- signed
- sizeof
- static
- struct
- switch
- typedef
- union
- unsigned
- void
- volatile
- while

The Types

- Each object (variable) has 3 informations: storage, type and name
- Storage defines where the object is created (stack, .data section, .readonly section, …)
- Storage can be ‘auto’, ‘register’, ‘static’, ‘extern’
- ‘auto’ means the object is created in the current scope (and destroyed at the end of the scope). It is implicit (in fact very probably you will never see it) in functions local variables. You should have recognized by now that the scope is an abstraction for the stack
- ‘register’ means the object is stored in a register. (this is generally a bad idea and very few times it gives a true advantage, it is a relic of an era where compilers did not do any kind of optimization)
- ‘static’ instructs the compiler to put the object in one of the program sections (.data, .readonly, .bss…). It means the object will be alive for the whole program life. Note: the keyword changes meaning based on where it is used (in a stack: the object is persistent, out of stack: not visible to other files)
- ‘extern’ means the object is stored in some other file (ex. you declare and set in foo.c and you want to access it from bar.c). This is generally used for global variables (the ones you define out of functions)
- The official types are: char (1 byte), short (2 bytes), int (4 bytes), long (4/8 bytes), long long (8 bytes), float and double. All the integer ones can be signed (implicit) or unsigned (must be explicit).
- The void type means ‘nothing’ and it has various uses (like functions without a return value)

The “long” mess

- C born on 16 bits machines (in fact ‘int’ used to be 2 bytes)
- ‘long int’ was the type for 32 bits value
- Nowadays (well at least in the last 30 years) ‘int’ is 32 bits, but there is no consensus on the ‘long’ size
- Theoretically ‘long’ should represent the size of the biggest value storable in the CPU, but historically ‘long’ was a modifier for int (for upgrading it from 2 bytes to 4).
- Currently we have two different approaches: The Microsoft one where long is 4 bytes everywhere, and the rest of the world where long is 8 bytes on 64 bits architectures and 4 bytes on 32 (so long is platform-dependent)
- Currently the best thing you can do is to use ‘int’ and ‘long long’ and forget about ‘long’.
- There are better approaches to the problem offered by the C standard library (we are going to see them later)

Please use const, really…

```
int useless_adder(const int a, const int b, const int c);
int useless_multiplier(const int value0, const int value1);

void do_math()
{
    int total = useless_adder(1, 2, 3) + useless_multiplier(0x10, 0x20);
    total += 1;
    total = useless_multiplier(useless_adder(0xA, 0xB, 0xC), 0x17);
}

int useless_adder(const int a, const int b, const int c)
{
    return a + b + c;
}

int useless_multiplier(const int value0, const int value1)
{
    return value0 * value1;
}
```

Pointers

- While variables are values stored at memory addresses, pointers are memory addresses stored at memory addresses
- The & symbol will return the address of a variable
- The * symbol marks a type as a pointer and will 'dereference' a pointer by reading back its value (when applied to a variable)

```
void hello()
{
    int a = 100;
    int *ptr_of_a = &a;
    int *another_ptr_of_a = ptr_of_a;

    int a_copy_of_a = *ptr_of_a;
}
```

0x00000000_00000004 (int a)	100
0x00000000_00000008 (int *ptr_of_a)	0x00000000_00000004
0x00000000_00000010 (int *another_ptr_of_a)	0x00000000_00000004
0x00000000_00000018 (int a_copy_of_a)	100

Indexing (any pointer is indexable)

```c
void hello()
{
    int a = 100;
    int *ptr_of_a = &a;
    int a_copy_of_a = ptr_of_a[0];

    int array_of_ints[4];
    array_of_ints[0] = 100;
    array_of_ints[1] = 120;
    array_of_ints[2] = 200;
    array_of_ints[3] = 300;

    int *array_of_ptrs_of_ints[2];
    array_of_ptrs_of_ints[0] = ptr_of_a;
    array_of_ptrs_of_ints[1] = &a_copy_of_a;

    char array_of_chars[5] = {50, 60, 70, 80, 90};

    char array_of_chars_incomplete[4] = {50, 60, 70};
    array_of_chars_incomplete[3] = 80;

    unsigned char let_the_compiler_compute_the_size[] = {1, 2, 3, 4, 5, 6, 7, 8};
}
```

Pointers Math

```c
int hello()
{
    int values[] = {1, 2, 3};

    int *ptr_to_value_0 = values;
    int *ptr_to_value_1 = values + 1;
    int *ptr_to_value_2 = values + 2;

    int *another_ptr_to_value1 = ++ptr_to_value_0;
    int *another_another_ptr_to_value1 = ptr_to_value_0++;
    int *another_ptr_to_value2 = ptr_to_value_0;

    char list[] = {0, 4, 8, 12};

    char* ptr_to_12 = list + 3;
}
```

Strings

- Strings in C are pointers to char
- Strings must be 0 terminated
- The double quote syntactic sugar allows you to create strings on the fly (in readonly memory!)

```
void hello()
{
    char* i_am_a_string = "Hello World";

    char writable_string[] = {'c', 'i', 'a', 'o', 0};
    writable_string[0] = 'C'; // will not crash

    i_am_a_string[0] = 'h'; // will crash as i_am_a_string is in a readonly segment
}
```

Strings (improved)

- If you can, always try to make the compiler warns about memory issues, const is your ally

```
void hello()
{
    const char* i_am_a_string = "Hello World";

    char writable_string[] = {'c', 'i', 'a', 'o', 0};
    writable_string[0] = 'C'; // will not crash

    i_am_a_string[0] = 'h'; // will not compile as i_am_a_string is const
}
```

Casting

- C is a weakly typed language, it means you can cast to everything you want without checks
- This means you can make any kind of mess…

```
void hello()
{
    char items[] = {1, 2, 3};

    int *fake_int_ptr_to_chars = (int *)items;

    int welcome_to_the_crash = fake_int_ptr_to_chars[2]; // pointer math will go out of the items bounds
}
```

Enums

```
enum weapons
{
    Sword,
    Spear,
    Bow,
    Axe,
    Max_Weapons
};

enum classes
{
    Warrior = 100,
    Elf = 200,
    Witcher,
    Mage,
    Max_Class
};

int main()
{
    enum weapons current_weapon = Sword;
    for (int i = 0; i < Max_Weapons; i++)
    {
        if (i == Axe)
        {
            // picked an axe
        }
    }
}
```

Complex Types: Structs

```c
struct vec3
{
    float x;
    float y;
    float z;
};

struct transform
{
    struct vec3 position;
    struct vec3 rotation;
    struct vec3 scale;
};

int main()
{
    struct vec3 position = {0, 0, 0};

    struct vec3 *position_ptr = &position;

    float x = position.x;
    float y = position_ptr->y;

    float *z_ptr = &position.z;

    struct transform player_transform = {{0, 0, 0}, {0, 0, 0}, {1, 1, 1}};

    float scale_z = player_transform.scale.z;

    struct transform a_copy_of_transform = player_transform; // note here we are copying 9 floats... remember this for later slides...
}
```

Structs alignment: sizeof() is your friend

- All the fields of a structure must be aligned (the compiler will automatically do it)
- The structure final size must be a multiple of the biggest native type included (the compiler will automatically do it)

```
struct aligned_data
{
    int first;                      // offset 0
    unsigned short second;          // offset 4
    short third;                    // offset 6
    char fourth;                    // offset 8
    char fifth;                     // offset 9
    unsigned short sixth;           // offset 10
    int seventh;                    // offset 12
    unsigned long long eighth;      // offset 16
};

struct unaligned_data
{
    char first;   // offset 0
    int second;   // offset 1 UNALIGNED ! will be 4
    char third;   // offset 8
    char fourth;  // offset 9
};
```

typedef

```c
typedef float f32;

typedef struct vec3
{
    f32 x;
    f32 y;
    f32 z;
} vec3_t;

typedef struct transform
{
    vec3_t position;
    vec3_t rotation;
    vec3_t scale;
} transform;

int main()
{
    vec3_t position = {0, 0, 0};

    vec3_t *position_ptr = &position;

    f32 x = position.x;
    f32 y = position_ptr->y;

    f32 *z_ptr = &position.z;

    transform player_transform = {{0, 0, 0}, {0, 0, 0}, {1, 1, 1}};

    f32 scale_z = player_transform.scale.z;

    transform a_copy_of_transform = player_transform; // note here we are copying 9 floats... remember this for later slides...
}
```

Pointers of Pointers

```c
void hello()
{
    int numbers[] = {10, 20, 30, 40};
    int *numbers_ptr = numbers;
    int **numbers_ptrptr = &numbers_ptr;
}
```

0x00000000_00000004 (int numbers[0])	0x0000000A
0x00000000_00000008 (int numbers[1])	0x00000014
0x00000000_0000000C (int numbers[2])	0x0000001E
0x00000000_00000010 (int numbers[3])	0x00000028
0x00000000_00000014 (int *numbers_ptr)	0x00000000_00000004
0x00000000_0000001C (int **numbers_ptrptr)	0x00000000_00000014

Unions

```
union super_data
{
    int integer;
    double floating;
    char filename[8];
};

struct position_t
{
    float x;
    float y;
    float z;
};

struct color_t
{
    float r;
    float g;
    float b;
};

typedef union float3
{
    struct position_t position;
    struct color_t color;
} float3;

int main()
{
    union super_data data;

    data.integer = 0x4F414943; // this will write "CIAO" in data.filename

    float3 blue;
    blue.position.y = 1; // we are setting blue.color.g too!
}
```

Part 0x02: Compiling and Linking

Common Compilers

- GCC (Gnu)
- MSVC (Microsoft)
- Intel C/C++ Compiler
- clang (LLVM)
- TCC
- …

Why clang ?

- The LLVM project invested years (and money) of research in compilers theory and optimizations
- LLVM is the base of Apple Swift, Julia, Rust, and many other projects
- Its C/C++ compiler (clang) is one of the most advanced and complete
- clang is compatible with gcc (you can easily switch between the two)
- Unreal Engine uses it for all of the platforms (except Windows, even if in the future this is going to change…)
- VisualStudio already allows you to use clang instead of MSVC
- Apple adopted clang years ago (switching from gcc)

Compiling with clang (and the LLVM suite)

clang.exe -c -o yourfile.o yourfile.c *compile yourfile.c into yourfile.o with default optimizations*

clang.exe -O0 -c -o yourfile.o yourfile.c *compile yourfile.c into yourfile.o without optimizations*

clang.exe -O3 -c -o yourfile.o yourfile.c *compile yourfile.c into yourfile.o with maximum optimizations*

clang.exe -g -O0 -c -o yourfile.o yourfile.c *compile yourfile.c into yourfile.o without optimizations and with debug symbols*

llvm-objdump.exe -d --x86-asm-syntax=intel yourfile.o *show disassembly of object file yourfile.o using the intel syntax (default is AT&T)*

llvm-objdump.exe -x yourfile.o *show sections, symbols and relocations of object file yourfile.o*

The Process Environment

- When a process starts, a very specific environment is built around it
- Each process has 3 virtual character devices associated: stdin, stdout, stderr for doing I/O (generally with terminals)
- A set of key-value strings (environment variables) that are inherited (copied) from the system. The process is free to overwrite them without creating problems to the other processes. The processes generated (children) by the current process will inherit its environment variables
- A list of ‘arguments’ (generally passed from the command line)
- At the end the process has to return an ‘exit code’ (an integer) reporting its status (like ‘everything went well’). 0 means OK.

main()

- This is the default entry point for C
- its signature/declaration is `int main();` the return value is the exit code of the process
- Its true signature is `int main(int argc, char **argv);`
- Its true true signature is `int main(int argc, char **argv, char **environ);`
- It is treated as a special function, given that it has a default return value of 0 (you are free to not call return even if it is a non-void function, this is the only case officially allowed)
- Linking an executable without a main() will trigger an error
- Obviously it is not required for a library

Linking

- LLVM supports various linkers (and has its own almost-experimental one)
- On Windows it uses by default the Microsoft linker (link.exe) distributed with VisualStudio (this is going to change again)
- It can create executables, as well as shared libraries
- Even if you run ‘clang.exe’ it is just a wrapper that will call link.exe (or lld-link.exe that is the LLVM implementation for Windows)

Linking with clang

clang.exe -o final.exe yourfile.o anotherfile.o *link yourfile.o and anotherfile.o into final.exe*

clang.exe -Xlinker /subsystem:console -o final.exe yourfile.o anotherfile.o *link yourfile.o and anotherfile.o into final.exe marking it as a Windows console application (it will spawn a terminal if not executed from another console application)*

clang.exe -Xlinker /subsystem:windows -o final.exe yourfile.o anotherfile.o *link yourfile.o and anotherfile.o into final.exe marking it as a Windows graphics application (it will never spawn a terminal)*

clang Shortcut (compile and link)

- This is useful when experimenting, in all of the other cases it is better to split compilation in various phase to reduce time (read: you do not need to recompile files that did not changed)

```
clang -o hello.exe hello.c
```

Part 0x03: Under the hood

Stack Frames

- Function’s local variables, as well as arguments and return values goes into the stack
- The stack used by a function is called “stack frame” (magenta blocks in next slides)
- Always remember that the stack pointer (rsp) must be 16 bytes aligned
- Optionally (for simplifying debugging) the end of the stack frame (aka frame pointer) can be stored in the rbp register (technically it is the top of the calling frame)
- If a function does not call other functions it is called a “leaf” function, and it has a way more-relaxed usage of the stack (generally only the 16 bytes alignment is enforced). Non-leaf functions requires (in addition to the 16 bytes alignment) the 32 bytes of shadow space (technically this is the amount of storage required for storing the 4 function arguments that are passed via registers [RCX, RDX, R8, R9]; This helps debuggers).
- Note that those are the calling conventions for X86-64 on Windows

A “leaf” function

```c
int i_am_a_leaf_function(int a, int b, int c)
{
    int d = a + b;
    if (d == 17)
    {
        return d * c;
    }
    return d % c;
}
```

rsp (caller rsp - 8 - 24)	pad for alignment
rsp + 4	int d
rsp + 8	int a (from rcx/ecx)
rsp + 12	int b (from rdx/edx)
rsp + 16	int c (from r8/r8d)
rsp + 20	int return value (to rax/eax)
caller rsp - 8	return address

```
0000000000000000 <i_am_a_leaf_function>:
        0: 48 83 ec 18                  sub     rsp, 24
        4: 44 89 44 24 10               mov     dword ptr [rsp + 16], r8d
        9: 89 54 24 0c                  mov     dword ptr [rsp + 12], edx
        d: 89 4c 24 08                  mov     dword ptr [rsp + 8], ecx
        11: 8b 44 24 08                 mov     eax, dword ptr [rsp + 8]
        15: 03 44 24 0c                 add     eax, dword ptr [rsp + 12]
        19: 89 44 24 04                 mov     dword ptr [rsp + 4], eax
        1d: 83 7c 24 04 11              cmp     dword ptr [rsp + 4], 17
        22: 0f 85 12 00 00 00           jne     0x3a <i_am_a_leaf_function+0x3a>
        28: 8b 44 24 04                 mov     eax, dword ptr [rsp + 4]
        2c: 0f af 44 24 10              imul    eax, dword ptr [rsp + 16]
        31: 89 44 24 14                 mov     dword ptr [rsp + 20], eax
        35: e9 0d 00 00 00              jmp     0x47 <i_am_a_leaf_function+0x47>
        3a: 8b 44 24 04                 mov     eax, dword ptr [rsp + 4]
        3e: 99                          cdq
        3f: f7 7c 24 10                 idiv    dword ptr [rsp + 16]
        43: 89 54 24 14                 mov     dword ptr [rsp + 20], edx
        47: 8b 44 24 14                 mov     eax, dword ptr [rsp + 20]
        4b: 48 83 c4 18                 add     rsp, 24
        4f: c3                          ret
```

A “leaf” function (other types)

```
long long i_am_a_leaf_function(int a, int b, unsigned char c)
{
    int d = a + b;
    if (d == 17)
    {
        return d * c;
    }
    return d % c;
}
```

```
0000000000000000 <i_am_a_leaf_function>:
        0: 48 83 ec 18                sub     rsp, 24
        4: 44 88 44 24 0f             mov     byte ptr [rsp + 15], r8b
        9: 89 54 24 08                mov     dword ptr [rsp + 8], edx
        d: 89 4c 24 04                mov     dword ptr [rsp + 4], ecx
       11: 8b 44 24 04                mov     eax, dword ptr [rsp + 4]
       15: 03 44 24 08                add     eax, dword ptr [rsp + 8]
       19: 89 04 24                   mov     dword ptr [rsp], eax
       1c: 83 3c 24 11                cmp     dword ptr [rsp], 17
       20: 0f 85 17 00 00 00          jne     0x3d <i_am_a_leaf_function+0x3d>
       26: 8b 04 24                   mov     eax, dword ptr [rsp]
       29: 0f b6 4c 24 0f             movzx   ecx, byte ptr [rsp + 15]
       2e: 0f af c1                   imul    eax, ecx
       31: 48 98                      cdqe
       33: 48 89 44 24 10             mov     qword ptr [rsp + 16], rax
       38: e9 13 00 00 00             jmp     0x50 <i_am_a_leaf_function+0x50>
       3d: 8b 04 24                   mov     eax, dword ptr [rsp]
       40: 0f b6 4c 24 0f             movzx   ecx, byte ptr [rsp + 15]
       45: 99                         cdq
       46: f7 f9                      idiv    ecx
       48: 48 63 c2                   movsxd  rax, edx
       4b: 48 89 44 24 10             mov     qword ptr [rsp + 16], rax
       50: 48 8b 44 24 10             mov     rax, qword ptr [rsp + 16]
       55: 48 83 c4 18                add     rsp, 24
       59: c3                         ret
```

rsp (caller rsp - 8 - 24)	int d
rsp + 4	int a (from rcx/ecx)
rsp + 8	int b (from rdx/edx)
rsp + 15	unsigned char c (from r8/r8b)
rsp + 16	long long return value (to rax)
caller rsp - 8	return address

A “non-leaf” function (pure unsigned 64 bits)

```c
unsigned long long i_am_a_leaf_function(unsigned long long a, unsigned long long b, unsigned long long c)
{
    unsigned long long d = a + b;
    d = useless_function(d);
    if (d == 17)
    {
        return d * c;
    }
    return d % c;
}
```

```
0000000000000010 <i_am_a_leaf_function>:
          10: 48 83 ec 48                              sub      rsp, 72
          14: 4c 89 44 24 38                           mov      qword ptr [rsp + 56], r8
          19: 48 89 54 24 30                           mov      qword ptr [rsp + 48], rdx
          1e: 48 89 4c 24 28                           mov      qword ptr [rsp + 40], rcx
          23: 48 8b 44 24 28                           mov      rax, qword ptr [rsp + 40]
          28: 48 03 44 24 30                           add      rax, qword ptr [rsp + 48]
          2d: 48 89 44 24 20                           mov      qword ptr [rsp + 32], rax
          32: 48 8b 4c 24 20                           mov      rcx, qword ptr [rsp + 32]
          37: e8  00 00 00 00                call      <useless_function>
          3c: 48 89 44 24 20                           mov      qword ptr [rsp + 32], rax
          41: 48 83 7c 24 20 11                        cmp      qword ptr [rsp + 32], 17
          47: 0f 85 15 00 00 00                        jne      0x62 <i_am_a_leaf_function+0x52>
          4d: 48 8b 44 24 20                           mov      rax, qword ptr [rsp + 32]
          52: 48 0f af 44 24 38                        imul     rax, qword ptr [rsp + 56]
          58: 48 89 44 24 40                           mov      qword ptr [rsp + 64], rax
          5d: e9 13 00 00 00                           jmp      0x75 <i_am_a_leaf_function+0x65>
          62: 48 8b 44 24 20                           mov      rax, qword ptr [rsp + 32]
          67: 31 c9                                    xor      ecx, ecx
          69: 89 ca                                    mov      edx, ecx
          6b: 48 f7 74 24 38                           div      qword ptr [rsp + 56]
          70: 48 89 54 24 40                           mov      qword ptr [rsp + 64], rdx
          75: 48 8b 44 24 40                           mov      rax, qword ptr [rsp + 64]
          7a: 48 83 c4 48                              add      rsp, 72
          7e: c3                                       ret
```

rsp (caller rsp - 8 - 72)	32 bytes shadow space
rsp + 32	unsigned long long d
rsp + 40	unsigned long long a (from rcx)
rsp + 48	unsigned long long b (from rdx)
rsp + 56	unsigned long long c (from r8)
rsp + 64	unsigned long long return value (to rax)
caller rsp - 8	return address

Pointers of Pointers

```c
void ptrofptr()
{
    const char *string0 = "Hello";
    const char *string1 = " ";
    const char *string2 = "World";

    const char *strings[3];
    strings[0] = string0;
    strings[1] = string1;
    strings[2] = string2;

    const char **strings_ptr = strings;
    for (int i = 0; i < 3; i++)
    {
        const char *string = strings_ptr[i];
    }
}
```

rsp (caller rsp - 8 - 88)	pad for alignment
rsp + 8	const char *string
rsp + 20	int i
rsp + 24	const char **strings_ptr
rsp + 32	const char *strings[0]
rsp + 40	const char *strings[1]
rsp + 48	const char *strings[2]
rsp + 64	const char *string2
rsp + 72	const char *string1
rsp + 80	const char *string0
caller rsp - 8	return address

```
0000000000000000 <ptrofptr>:
          0:  48 83 ec 58              sub     rsp, 88
          4:  48 8d 05 00 00 00 00     lea     rax, [???]
          b:  48 89 44 24 50           mov     qword ptr [rsp + 80], rax
         10:  48 8d 05 00 00 00 00     lea     rax, [???]
         17:  48 89 44 24 48           mov     qword ptr [rsp + 72], rax
         1c:  48 8d 05 00 00 00 00     lea     rax, [???]
         23:  48 89 44 24 40           mov     qword ptr [rsp + 64], rax
         28:  48 8b 44 24 50           mov     rax, qword ptr [rsp + 80]
         2d:  48 89 44 24 20           mov     qword ptr [rsp + 32], rax
         32:  48 8b 44 24 48           mov     rax, qword ptr [rsp + 72]
         37:  48 89 44 24 28           mov     qword ptr [rsp + 40], rax
         3c:  48 8b 44 24 40           mov     rax, qword ptr [rsp + 64]
         41:  48 89 44 24 30           mov     qword ptr [rsp + 48], rax
         46:  48 8d 44 24 20           lea     rax, [rsp + 32]
         4b:  48 89 44 24 18           mov     qword ptr [rsp + 24], rax
         50:  c7 44 24 14 00 00 00 00  mov     dword ptr [rsp + 20], 0
         58:  83 7c 24 14 03           cmp     dword ptr [rsp + 20], 3
         5d:  0f 8d 23 00 00 00        jge     0x86 <ptrofptr+0x86>
         63:  48 8b 44 24 18           mov     rax, qword ptr [rsp + 24]
         68:  48 63 4c 24 14           movsxd  rcx, dword ptr [rsp + 20]
         6d:  48 8b 04 c8              mov     rax, qword ptr [rax + 8*rcx]
         71:  48 89 44 24 08           mov     qword ptr [rsp + 8], rax
         76:  8b 44 24 14              mov     eax, dword ptr [rsp + 20]
         7a:  83 c0 01                 add     eax, 1
         7d:  89 44 24 14              mov     dword ptr [rsp + 20], eax
         81:  e9 d2 ff ff ff           jmp     0x58 <ptrofptr+0x58>
         86:  48 83 c4 58              add     rsp, 88
         8a:  c3                       ret
```

Consts FTW

```
int dummy_function()
{
    int a = 17;
    return a;
}

int dummy_const_function()
{
    const int a = 17;
    return a;
}
```

```
0000000000000000 <dummy_function>:
           0: 50                            push     rax
           1: c7 44 24 04 11 00 00 00       mov      dword ptr [rsp + 4], 17
           9: 8b 44 24 04                   mov      eax, dword ptr [rsp + 4]
           d: 59                            pop      rcx
           e: c3                            ret
           f: 90                            nop

0000000000000010 <dummy_const_function>:
          10: 50                            push     rax
          11: c7 44 24 04 11 00 00 00       mov      dword ptr [rsp + 4], 17
          19: b8 11 00 00 00                mov      eax, 17
          1e: 59                            pop      rcx
          1f: c3                            ret
```

Trust the compiler optimizations!

```
int multiplier(int a)
{
    return a * 17;
}
```

```
0000000000000000 <multiplier>:
       0: 50                               push    rax
       1: 89 4c 24 04                      mov     dword ptr [rsp + 4], ecx
       5: 6b 44 24 04 11                   imul    eax, dword ptr [rsp + 4], 17
       a: 59                               pop     rcx
       b: c3                               ret
```

```
0000000000000000 <multiplier>:
       0: 89 c8                            mov     eax, ecx
       2: c1 e0 04                         shl     eax, 4
       5: 01 c8                            add     eax, ecx
       7: c3                               ret
```

Part 0x04: The Preprocessor

#include directive

Preprocesses and Inserts the content of the specified file in the preprocessed source

#include “file” // relative to the directory of the currently processed file

#include <file> // searched in “Include Paths”

It is very common to use includes for declarations (so you can reuse them)

You can add “Include Paths” using the -I option:

clang -c -o foo.o -I includeFirst/ -I Foo/AnotherInclude foo.c

#define directive

```c
#define print puts
#define hello(message) print(message)
#define print_type(t) print(#t)
#define print_type_verbose(t) print("The type is " #t ".")
#define print_type_and_size(t) printf("the type is " #t " and has size %llu\n", sizeof(t))

#define zalloc(size) calloc(size, 1)
#define call_with_type(func, type, value) func##_##type##_exec(value)

#define call_with_type_verbose(func, type, value) print("about to call " #func "_" #type "_exec(" #value ")")

#include <stdio.h>

float hello_float_exec(const float argument)
{
    print("i am hello_float_exec");
    return argument * 2;
}

int main()
{
    hello("starting the program...");
    print_type_verbose(unsigned long long);
    print_type_and_size(unsigned long);
    print_type_and_size(unsigned int);
    call_with_type(hello, float, 0x100);
    call_with_type_verbose(hello, float, 0x100);
    return 0;
}
```

#undef

Undefine a #define :)

Check the next slide for understanding how much it can be useful

```
#define SLOW_MODE 1
#undef SLOW_MODE
```

#if #else #elif #endif

```
#define SLOW_MODE 1
#undef SLOW_MODE

#if SLOW_MODE == 1
int number_of_buttons = 8;
#elif SLOW_MODE == 2
int number_of_buttons = 16;
#elif SLOW_MODE < 0
int number_of_buttons = 0xFFFF;
#else
int number_of_buttons = 32;
#endif
```

#ifdef #ifndef

```
#define SLOW_MODE 1

#ifdef SLOW_MODE
int number_of_buttons = 8;
#endif

#ifndef SLOW_MODE
int number_of_buttons = 0;
#endi
```

#error #warning

```
#ifdef WINDOWS
#include <Windows.h>
#endif

#ifdef LINUX
#include <unistd.h>
#endif

#ifdef SWITCH
#warning Nintendo Switch support is experimental
#include <nintendo.h>
#endif

#ifdef ANDROID
#error android not supported
#endif
```

Macros

```
__FILE__ // the currently processing file

__LINE__ // the currently processing line

__DATE__ // current date as string

__TIME__ // current time as string
```

Governing the Preprocessor from clang

```c
#include <stdio.h>

int main()
{
#ifdef HELLO
    puts("hello");
#endif
    return 0;
}
```

```
clang -o test.exe -DHELLO test.c

clang -o test.exe -UHELLO test.c

clang -o test.exe -DHELLO=99 test.c
```

Getting preprocessor output

clang.exe -E file.c

will print on the stdout the result of the preprocessor phase (the format reported is # linenum filename flags, where flags can be 1:start of a new file, 2:return from a file, 3:from a system header file, 4:wrapped in extern [you can safely ignore 3 and 4]):

```
#include "includeme.h"
#define hello foo

void func()
{
    hello();
}

clang.exe -E preproc.c
# 1 "preproc.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 326 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "preproc.c" 2
# 1 "./includeme.h" 1
void from_another_file()
{

}
# 2 "preproc.c" 2


void func()
{
              foo();
}
```

Part 0x05: The C Standard Library

A User Space World

- The code seen until now makes use of the standard CPU/Turing Machine features. We have a bunch of addresses and data.
- For integrating with external hardware (like I/O devices, graphics cards…) we need to talk with the operating system
- Given that each operating system has its rules and behaviours, the C compilers offer a set of ready-to-use system/platform independent functions for managing common tasks.
- Under the hood those functions are obviously system/platform specific, but we can safely ignore that aspect most of the time
- The C standard library is offered in the form of ‘header files’ (read: declarations) that you can include in your code.
- All of those functions implementations are generally in a shared library that is automatically linked in by the C linker (obviously you can disable that behaviour if you are developing for a baremetal environment)

C Standard headers

- assert.h
- ctype.h
- errno.h
- float.h
- limits.h
- locale.h
- math.h
- setjmp.h
- signal.h
- stdarg.h
- stddef.h
- stdint.h (only from C99, but super-useful for us)
- stdio.h
- stdlib.h
- string.h
- time.h

#include <stddef.h>

- The ***size_t*** type is exposed by this header and it is mapped to the size of the maxim theoretical object storable in the specific CPU architecture.
- This means on a 32 bit system it will be ‘unsigned int’ while on 64 it will be ‘unsigned long long’
- There is even a ***ssize_t*** variant for signed values (Posix only)

#include <stdio.h>

```c
#include <stdio.h>

int main(int argc, char **argv, char **environ)
{
    puts("Please input a char and press enter/return..." );
    int inputed_char = getchar();

    printf("char inputed %d\n", inputed_char);

    for (int i = 0; i < argc; i++)
    {
        printf("Argument %d: %s\n", i, argv[i]);
    }

    char **envs = environ;
    char *env;
    while ((env = *envs++))
    {
        printf("%s\n", env);
    }

    return 0;
}
```

Working with the Heap

- Til now we used memory in the stack
- This is easy, fast and does not require any communication with the operating system.
- But: the stack is a limited resource (generally a couple of Megabytes are allocated, albeit configurable); the stack content is virtually ‘freed’ at the exit of the scope/frame that managed it; sometime we want memory areas out of the stack that are fully user-controlled
- The stack works only for data we know the size before. For dynamically sized data we cannot use it (there are tricks for this, but are not portable and they are generally fragile)
- The Heap is the collection of (writable) memory areas the process has access to. In the original Heap concept, it was just a single growing/shrinking area. Nowadays there can be multiple areas mapped around the process address space.
- Working with the Heap allows to allocate huge amounts of memory, but you need to remember to free them when not used; and you need to remember that the vast majority of time you need a syscall for asking the operating system for new pages.

The NULL macro

- The C Standard Library exposes a #define NULL ((void *)0)
- It is basically a cast for the value 0 as a pointer
- The address 0 (or virtual page 0) is always mapped by the operating system to an invalid entry, so any attempt to access it will trigger a crash.
- Functions returning pointers, uses NULL (by convention) for signaling an error condition (remember to check the return value for NULL before working with the returned pointer!)

#include <stdlib.h>

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    const size_t amount_of_memory_to_allocate = 8 * 1024 * 1024;
    unsigned char *bytes_on_heap = malloc(amount_of_memory_to_allocate);
    if (!bytes_on_heap)
    {
        printf("unable to allocate memory\n");
        return -1;
    }

    for (size_t i = 0; i < amount_of_memory_to_allocate; i++)
    {
        bytes_on_heap[i] = rand();
    }

    free(bytes_on_heap);
    return 0;
}
```

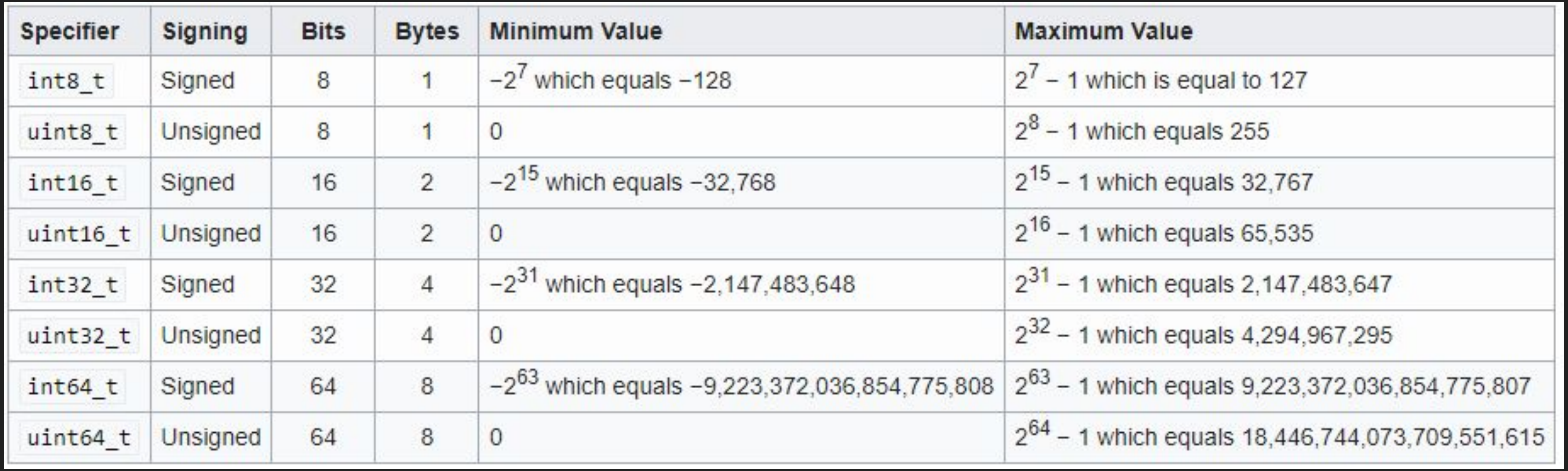
Notes on allocations

- Modern operating systems apply a lazy allocation of pages
- This means that if you are asking for 8 Megabytes, only the MMU entries for the area are prepared. They will be mapped to physical pages on demand (and page by page)
- A COW (Copy On Write) approach may be in place, where the same page is mapped when reading (on the beginning) and then it is replaced when writing.
- While C assumes no initialization is in place for malloc() (read you will find garbage in it), the operating system needs to clear the pages before assigning them to a process. This is a security measure (otherwise a process can read pages previously used by other processes)
- This mechanism can be skipped for pages assigned to the same process
- For 'tiny' allocations the default malloc() behaviour is to increase the Heap area, for bigger ones a new mapping is generally created.
- The C library will try to reuse the same areas whenever possible.
- The calloc(size_t nmemb, size_t size) is like malloc but will grant zeroing of the memory areas (again COW can be in place, so a single zero-filled page can be used for the whole mapping at the beginning)

void* realloc(void *ptr, size_t size)

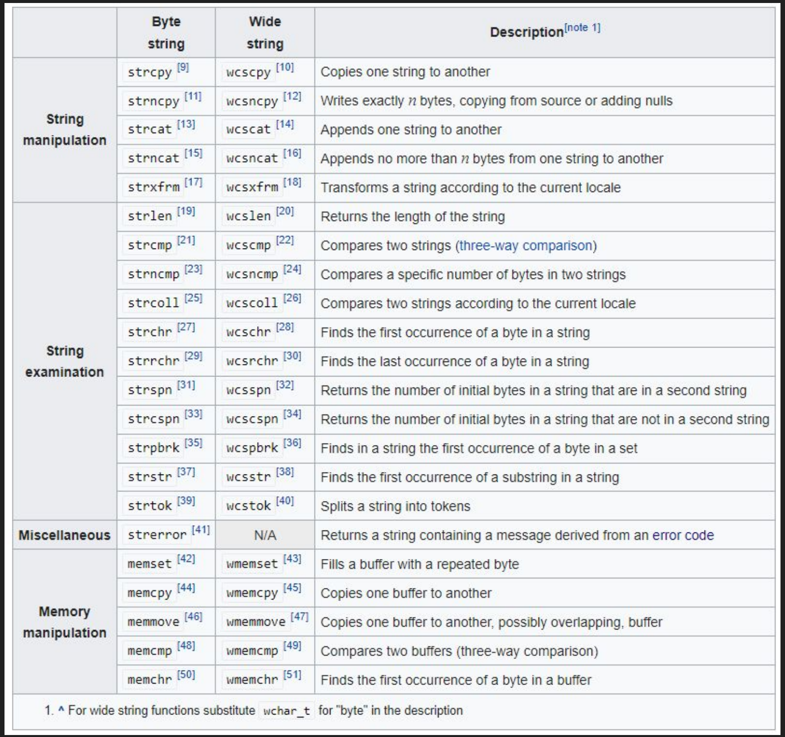
- This function tries to ‘increase’ or ‘shrink’ a memory area (preserving the content)
- Pay attention to the return value, as the function may return NULL in case of failure
- If ptr is NULL the function behaves like malloc()
- If size is 0 and ptr is not NULL the function behaves like free()
- If realloc() is forced to create a new area (read: the return address will be different from ptr) an implicit free() for the old area will be called.

#include <stdint.h>



Specifier	Signing	Bits	Bytes	Minimum Value	Maximum Value
`int8_t`	Signed	8	1	-2^{7} which equals −128	$2^{7} - 1$ which is equal to 127
`uint8_t`	Unsigned	8	1	0	$2^{8} - 1$ which equals 255
`int16_t`	Signed	16	2	-2^{15} which equals −32,768	$2^{15} - 1$ which equals 32,767
`uint16_t`	Unsigned	16	2	0	$2^{16} - 1$ which equals 65,535
`int32_t`	Signed	32	4	-2^{31} which equals −2,147,483,648	$2^{31} - 1$ which equals 2,147,483,647
`uint32_t`	Unsigned	32	4	0	$2^{32} - 1$ which equals 4,294,967,295
`int64_t`	Signed	64	8	-2^{63} which equals −9,223,372,036,854,775,808	$2^{63} - 1$ which equals 9,223,372,036,854,775,807
`uint64_t`	Unsigned	64	8	0	$2^{64} - 1$ which equals 18,446,744,073,709,551,615

#include <string.h>



	Byte string	Wide string	Description[note 1]
String manipulation	strcpy [9]	wcscpy [10]	Copies one string to another
	strncpy [11]	wcsncpy [12]	Writes exactly n bytes, copying from source or adding nulls
	strcat [13]	wcscat [14]	Appends one string to another
	strncat [15]	wcsncat [16]	Appends no more than n bytes from one string to another
	strxfrm [17]	wcsxfrm [18]	Transforms a string according to the current locale
String examination	strlen [19]	wcslen [20]	Returns the length of the string
	strcmp [21]	wcscmp [22]	Compares two strings (three-way comparison)
	strncmp [23]	wcsncmp [24]	Compares a specific number of bytes in two strings
	strcoll [25]	wcscoll [26]	Compares two strings according to the current locale
	strchr [27]	wcschr [28]	Finds the first occurrence of a byte in a string
	strrchr [29]	wcsrchr [30]	Finds the last occurrence of a byte in a string
	strspn [31]	wcsspn [32]	Returns the number of initial bytes in a string that are in a second string
	strcspn [33]	wcscspn [34]	Returns the number of initial bytes in a string that are not in a second string
	strpbrk [35]	wcspbrk [36]	Finds in a string the first occurrence of a byte in a set
	strstr [37]	wcsstr [38]	Finds the first occurrence of a substring in a string
	strtok [39]	wcstok [40]	Splits a string into tokens
Miscellaneous	strerror [41]	N/A	Returns a string containing a message derived from an error code
Memory manipulation	memset [42]	wmemset [43]	Fills a buffer with a repeated byte
	memcpy [44]	wmemcpy [45]	Copies one buffer to another
	memmove [46]	wmemmove [47]	Copies one buffer to another, possibly overlapping, buffer
	memcmp [48]	wmemcmp [49]	Compares two buffers (three-way comparison)
	memchr [50]	wmemchr [51]	Finds the first occurrence of a byte in a buffer

1. ^ For wide string functions substitute `wchar_t` for "byte" in the description

The amazing memcpy(void *dest, const void *src, size_t n)

- While part of the standard library it is implicitly used whenever the compiler needs to copy more data than the ones fitting in a register (included XMM registers that are 128 bits)
- This happens because memcpy is heavily optimized by using tons of tricks for reducing the steps required for copying (read: it is not a plain for cycle over an array of bytes)
- So do not try to reimplement it ;)
- … But try to use it whenever you can

System Errors

errno is a global variable exposed by the c library where the operating system specific error is written

void perror(const char*) prints on the stderr a string representation of the error

char *strerror(int) returns a string based on the system error passed as argument

Working with Files (standard API, there are better choices…)

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
    if (argc < 2)
    {
        fprintf(stderr, "Please specify the file to read\n");
        return -1;
    }

    const char *filename = argv[1];

    FILE *handle = fopen(filename, "rb"); // ask the kernel to open a file (syscall -> filesystem -> block device)
    if (!handle)
    {
        fprintf(stderr, "Unable to open file %s\n", filename);
        return -1;
    }

    if (fseek(handle, 0, SEEK_END)) // move the file cursor at the end [syscall]
    {
        perror("fseek()"); fclose(handle);
        return -1;
    }

    long filesize = ftell(handle); // read the current cursor position (technically it is the file size)[syscall]
    fprintf(stdout, "File size: %ld bytes\n", filesize);

    char *data = malloc(filesize + 1); // allocate memory for the file content + 1 byte for the final 0 (we consider the content a string) [potential syscall]
    if (!data)
    {
        perror("malloc()"); fclose(handle);
        return -1;
    }

    if (fseek(handle, 0, SEEK_SET)) // reset the cursor to the initial position [syscall]
    {
        perror("fseek()"); fclose(handle); free(data);
        return -1;
    }

    int ret = fread(data, 1, filesize, handle); // read the whole content of the file into 'data' [syscall]
    if (ret != filesize)
    {
        fprintf(stderr, "Unable to read data from %s (returned value: %d)\n", filename, ret);
        fclose(handle); free(data);
        return -1;
    }

    data[filesize] = 0;

    fprintf(stdout, "%s\n", data);

    free(data);
    fclose(handle);
    return 0;
}
```

Numeric Conversions

```
#include <stdlib.h>

int atoi(const char *nptr);

long atol(const char *nptr);

long long atoll(const char *nptr);

double atof(const char *nptr);

double strtod(const char *nptr, char **endptr);

float strtof(const char *nptr, char **endptr);

long double strtold(const char *nptr, char **endptr);

#include <stdio.h>

int sprintf(char *str, const char *format, ...);

int snprintf(char *str, size_t size, const char *format, ...);
```

Hex Strings Example

```c
#include <ctype.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

unsigned long long hex_to_ull(const char *string)
{
    unsigned long long value = 0;
    unsigned long long multiplier = 1;
    size_t string_length = strlen(string);
    for (int i = string_length - 1; i >= 0; i--)
    {
        char nibble = toupper(string[i]);

        if (nibble >= '0' && nibble <= '9')
        {
            value += (nibble - '0') * multiplier;
        }
        else if (nibble >= 'A' && nibble <= 'F')
        {
            value += (10 + (nibble - 'A')) * multiplier;
        }
        else
        {
            printf("unknown char!\n");
            return 0;
        }

        multiplier *= 16;
    }

    return value;
}

int main()
{
    unsigned long long value = hex_to_ull("1FF03");
    printf("%llu %llx\n", value, value);

    printf("%llu\n", hex_to_ull("00"));
    printf("%llu\n", hex_to_ull("01"));
    printf("%llu\n", hex_to_ull("DEADBEEF"));

    return 0;
}
```

Part 0x06: Common Data Structures

Linked Lists (step 0x00)

```c
#include <stddef.h> // required for NULL

struct list_node
{
    struct list_node *next;
};

struct list_node *list_get_tail(struct list_node **head)
{
    struct list_node *current_node = *head;
    struct list_node *last_node = NULL;

    while (current_node)
    {
        last_node = current_node;
        current_node = current_node->next;
    }

    return last_node;
}
```

Linked Lists (step 0x01)

```
struct list_node *list_append(struct list_node **head, struct list_node *item)
{
    struct list_node *tail = list_get_tail(head);
    if (!tail)
    {
        *head = item;
    }
    else
    {
        tail->next = item;
    }

    item->next = NULL;
    return item;
}

struct list_node *list_pop(struct list_node **head)
{
    struct list_node *current_head = *head;
    if (!current_head)
    {
        return NULL;
    }

    *head = (*head)->next;
    current_head->next = NULL;

    return current_head;
}
```

Linked Lists (step 0x02)

```c
struct string_item
{
    struct list_node node;
    const char *string;
};

struct string_item *string_item_new(const char *string)
{
    struct string_item *item = malloc(sizeof(struct string_item));
    if (!item)
    {
        return NULL;
    }
    item->string = string;
    return item;
}

int main()
{
    struct string_item *my_linked_list = NULL;
    list_append((struct list_node **)&my_linked_list, (struct list_node *)string_item_new("Hello World"));
    list_append((struct list_node **)&my_linked_list, (struct list_node *)string_item_new("Test001"));
    list_append((struct list_node **)&my_linked_list, (struct list_node *)string_item_new("Test002"));
    list_append((struct list_node **)&my_linked_list, (struct list_node *)string_item_new("Last Item of the Linked List"));

    struct string_item *string_item = my_linked_list;
    while (string_item)
    {
        printf("%s\n", string_item->string);
        string_item = (struct string_item *)string_item->node.next;
    }
    return 0;
}
```

Linked Lists Exercises

- Implement removal of item
- Use preprocessor directives to reduce the verbose casting
- Higher Level Exercise: implement reversing of the list

Doubly Linked Lists

```
struct list_node
{
    struct list_node *prev;
    struct list_node *next;
};

struct list_node *list_append(struct list_node **head, struct list_node *item)
{
    struct list_node *tail = list_get_tail(head);
    if (!tail)
    {
        *head = item;
    }
    else
    {
        tail->next = item;
    }

    item->prev = tail;
    item->next = NULL;
    return item;
}
```

Doubly Linked Lists Exercises

- Implement removal
- Implement insert after an item
- Implement insert before an item
- Higher Level Exercise: implement shuffling

Sets (part 0x00)

- A set is a list of quickly searchable values
- Each element is hashed to allow fast retrieval

```
size_t djb33x_hash(const char *key, const size_t keylen)
{
    size_t hash = 5381;

    for (size_t i = 0; i < keylen; i++)
    {
        hash = ((hash << 5) + hash) ^ key[i];
    }

    return hash;
}
```

Sets (part 0x01)

```c
struct set_node
{
    const char *key;
    size_t key_len;
    struct set_node *next;
};

struct set_table
{
    struct set_node **nodes;
    size_t hashmap_size;
};

struct set_table *set_table_new(const size_t hashmap_size)
{
    struct set_table *table = malloc(sizeof(struct set_table));
    if (!table)
    {
        return NULL;
    }
    table->hashmap_size = hashmap_size;
    table->nodes = calloc(table->hashmap_size, sizeof(struct set_node *));
    if (!table->nodes)
    {
        free(table);
        return NULL;
    }

    return table;
}
```

Sets (part 0x02)

```c
struct set_node *set_insert(struct set_table *table, const char *key, const size_t key_len)
{
    size_t hash = djb33x_hash(key, key_len);

    size_t index = hash % table->hashmap_size;

    struct set_node *head = table->nodes[index];
    if (!head)
    {
        table->nodes[index] = malloc(sizeof(struct set_node));
        if (!table->nodes[index])
        {
            return NULL;
        }
        table->nodes[index]->key = key;
        table->nodes[index]->key_len = key_len;
        table->nodes[index]->next = NULL;

        return table->nodes[index];
    }

    struct set_node *new_item = malloc(sizeof(struct set_node));
    if (!new_item)
    {
        return NULL;
    }
    new_item->key = key;
    new_item->key_len = key_len;
    new_item->next = NULL;

    struct set_node *tail = head;
    while (head)
    {
        tail = head;
        head = head->next;
    }
    tail->next = new_item;
}
```

Sets Exercises

- Implement search
- Implement removal
- Implement unique keys
- Higher Level Exercise: refactor code to allow the set implementation to reuse the linked list implementation

Dictionaries Challenge

- They are unique sets with a value
- Implement them over the set structures to support any kind of value
- Higher Level Exercise: keep count of how many collisions are in the hash map and increase the size of the hash table when there is too much ‘pressure’ (note: it requires rehashing of the whole dictionary)

Hey, what about Binary trees?

- we will cover them later in the course with practical usages

Part 0x06: Static and Shared Libraries

Static Libraries

- They are collections of functions/symbols that got embedded in the main executable
- They simplify deployment at the cost of a bigger image and the need to patch the whole executable whenever you need to fix one of the libraries
- They are just archives of object files (that will be linked to your executable)
- You can “join” a series of object files in a static library using the llvm-ar.exe command: *llvm-ar.exe rcs libfoo.lib first.o second.o*
- The rcs sequence means: r=replace or insert files in the archive, c=do not warn if a new archive is going to be created, s=generate an index of the symbols into the archive itself (under the hood the -s triggers the ‘llvm-ranlib’ command)
- Now you can just append the static library in your linker command line

Shared Libraries

- Note: on Windows you need to “mark” the functions you want to export/expose, on the other systems all is exported by default (even if you can control this)
- A common trick for portability is to define an “API tag”

```
#ifdef _WIN32
#define FOO_API __declspec(dllexport)
#else
#define FOO_API
#endif

FOO_API int hello_world()
{
    return 0x100;
}
```

clang.exe -shared -o foo.dll lib.c

Linking to a Shared Library

- Your operating system has a series of directories where libraries are searched for.
- The linker will search in them too, but if the library is not in a known path you need to specify/add it with the -L <path> option
- Shared libraries to link must be passed with the -l flag

```
clang.exe -o final.exe final.o -Lmylibraries/ -lfoo
```

On Windows this will search for foo.lib into the mylibraries/ directory

On the other systems it will search for libfoo.so

The Windows linker behaviour is pretty peculiar (and weird): even if the .dll file exposes all of the symbols informations required for linking, the microsoft linker expects a .lib files listing the exported symbols (this is not required for GCC linker as well as golink.exe, as a proof that it is again a relic from the past)

Part 0x07: The SDL Library

Why SDL?

- Abstracts operating system differences without hiding how the hardware works
- Used by lot of titles
- Available on dozens of platforms
- Unreal Engine uses it as the core for graphics and audio on Linux (given the amount of versions around)
- From a didactical perspective, its code is a bible on how the various operating systems deal with game-related hardware (GPUs, soundcards, gamepads…)

Before we start: a digression on Direct Memory Access (DMA)

- Most of modern devices (GPUs, soundcards, NICs, storage controllers…) are able to access system memory directly (without the CPU)
- At the electrical level, a procedure called “bus mastering” is in place, allowing a device to ‘own’ memory access by cooperating with some form of memory controller (historically called the northbridge in x86)
- This means your NIC can directly copy the received packets in a specific memory address and notify the CPU about the new packet with an interrupt
- At the same level your system could ask a sound card to start playing the samples stored at a specific memory address
- What about GPUs ?

GPUs and memory

- Most modern GPUs (especially for desktop/workstations and gaming consoles) have their own dedicated memory banks (but they can still access system memory)
- Even the cheaper GPUs are able to access system memory (well, for the cheapest ones the system memory is the only available)
- GPU dedicated memory is not directly accessible from the CPU
- Given that the GPU is able to access both types of memory, if you need to load data into the GPU dedicated memory, you first need to copy your bytes into system memory and then ask the GPU to copy them into its own dedicated memory
- It may looks slow, but the dedicated GPU memory is way (and way) faster in operations than the system one (especially because it needs to share it with the CPU)

Working with GPUs

Modern GPUs are really complex beasts, they are able to execute arbitrary opcodes as well as performing common Computer Graphics operations, like Rasterization and Raytracing.

The operating systems expose access to those features using low level apis (Vulkan, DirectX, Metal…). We will cover them in the future, but for now we will focus on the minimal amount of features for 2D games.

Back to SDL

- SDL (in addition to the other features related to input and audio) wraps low level GPU apis in simpler (higher level) functions
- Textures are the main concept we are going to use, given that for 2D games we generally have a bunch of images in the GPU and we ‘blit’ them to the screen/window

Another digression: Compositor

- Modern Graphical operating systems expose the ‘window’ concept using modern GPU concepts.
- Basically the content of each window is mapped to a different GPU texture, and a software known as the ‘compositor’ draw them in the correct place.
- Technically when you are ‘dragging’ a window you are blitting its texture in a different position.
- When in full screen mode, the compositor is basically turned off, and a single texture is blitted
- Everything happens transparently to the user, have you ever noticed the realtime window previews on your Windows 10 toolbar ?

Drawing in modern operating systems

- Your program needs to get access to the texture mapped to the window
- The truth is that you want two different textures, one for working (back buffer) and one for showing (front buffer)
- At each frame you swap between the two
- This group of textures is called ‘SwapChain’
- The procedure of swapping the back with the front is called ‘Present’

Getting SDL2

https://www.libsdl.org/

https://www.libsdl.org/download-2.0.php

(You need the development libraries, containing both the DLLs and the headers)

Initializing SDL

```c
#include <SDL.h>

int main(int argc, char **argv)
{
    if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) != 0)
    {
        SDL_Log("Unable to initialize SDL: %s", SDL_GetError());
        return 1;
    }

    SDL_Log("SDL is active!");

    SDL_Quit();

    return 0;
}
```

Building SDL applications on Windows

Console application:

```
clang.exe -o hello.exe -I .\SDL2-2.0.20\include -L .\SDL2-2.0.20\lib\x64 hello.c -Xlinker /subsystem:console -lSDL2main -lSDL2  -lshell32
```

Windows application:

```
clang.exe -o hello.exe -I .\SDL2-2.0.20\include -L .\SDL2-2.0.20\lib\x64 hello.c -Xlinker /subsystem:windows -lSDL2main -lSDL2  -lshell32
```

SDL2main and shell32?

- To be truly multiplatform, SDL implements its own main() functions
- At the top of SDL_main.h (included automatically by SDL.h) there is this line:
  #define main SDL_main
- This means your main is renamed as “SDL_main” and the right main implementation is taken based on the specified subsystem
- Those “mains” fixes some platform-specific behaviour to allow the same code to be built in the most possible platform-independent way
- shell32 is a Windows standard library that exposes symbols required by SDL2main (that is a static library, so it cannot resolve symbols at runtime)
- Note: this system is meaningless on Linux and Mac, where there is no difference between ‘console’ and ‘windows’ application

Can i get rid of this main() behaviour ?

Yep, just add a define before SDL inclusion:

```c
#define SDL_MAIN_HANDLED
#include <SDL.h>

int main(int argc, char **argv)
{
    if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) != 0)
    {
        SDL_Log("Unable to initialize SDL: %s", SDL_GetError());
        return 1;
    }

    SDL_Log("SDL is active!");

    SDL_Quit();

    return 0;
}
```

Now you can build with

```
clang -o hello.exe -I .\SDL2-2.0.20\include -L .\SDL2-2.0.20\lib\x64 hello.c -lSDL2
```

SDL_Init() and SDL_Quit()

It initializes the required subsystems implemented by SDL:

```
SDL_INIT_TIMER: timer subsystem
SDL_INIT_AUDIO: audio subsystem
SDL_INIT_VIDEO: video subsystem; automatically initializes the events subsystem
SDL_INIT_JOYSTICK: joystick subsystem; automatically initializes the events subsystem
SDL_INIT_HAPTIC: haptic (force feedback) subsystem
SDL_INIT_GAMECONTROLLER: controller subsystem; automatically initializes the joystick subsystem
SDL_INIT_EVENTS: events subsystem
SDL_INIT_EVERYTHING: all of the above subsystems
```

SDL_Quit() shutdowns all of the activated subsystems.

Both are wrappers for SDL_InitSubSystem() and SDL_QuitSubSystem() requiring manual specification of what to create/destroy

Creating a Window

```c
#include <SDL.h>

int main(int argc, char **argv)
{
    if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) != 0)
    {
        SDL_Log("Unable to initialize SDL: %s", SDL_GetError());
        return 1;
    }

    SDL_Window* window = SDL_CreateWindow("SDL is active!", 100, 100, 512, 512, 0);
    if (!window)
    {
        SDL_Log("Unable to create window: %s", SDL_GetError());
    }

    SDL_Delay(3000);

    SDL_Quit();

    return 0;
}
```

Why i cannot move/close the window?

Whenever you interact with a window, an “event” (or “message” in the Windows world) is generated.

Those events are enqueued and to be ‘processed’ must be dequeued.

When we try to move the window the mouse-related events are generated but the window is not dequeuing them

Dequeuing events

```
#include <SDL.h>

int main(int argc, char **argv)
{
    if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) != 0)
    {
        SDL_Log("Unable to initialize SDL: %s", SDL_GetError());
        return 1;
    }

    SDL_Window* window = SDL_CreateWindow("SDL is active!", 100, 100, 512, 512, 0);
    if (!window)
    {
        SDL_Log("Unable to create window: %s", SDL_GetError());
    }

    for(;;)
    {
        SDL_Event event;
        while(SDL_PollEvent(&event))
        {
            // do something with the event
        }
    }

    SDL_Quit();

    return 0;
}
```

The SDL_QUIT event

```c
#include <SDL.h>

int main(int argc, char **argv)
{
    if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) != 0)
    {
        SDL_Log("Unable to initialize SDL: %s", SDL_GetError());
        return 1;
    }

    SDL_Window* window = SDL_CreateWindow("SDL is active!", 100, 100, 512, 512, 0);
    if (!window)
    {
        SDL_Log("Unable to create window: %s", SDL_GetError());
    }

    int running = 1;
    while (running)
    {
        SDL_Event event;
        while(SDL_PollEvent(&event))
        {
            if (event.type == SDL_QUIT)
            {
                running = 0;
            }
        }
    }

    SDL_Quit();

    return 0;
}
```

SDL Event Types (well a subset of them)

- SDL_QUIT
- SDL_KEYDOWN
- SDL_KEYUP
- SDL_WINDOWEVENT
- SDL_MOUSEMOTION
- SDL_MOUSEBUTTONDOWN
- SDL_MOUSEBUTTONUP
- SDL_MOUSEWHEEL
- SDL_CONTROLLERBUTTONDOWN
- SDL_CONTROLLERBUTTONUP
- SDL_CONTROLLERAXISMOTION

https://wiki.libsdl.org/SDL_Event

Getting a Renderer (and our SwapChain)

```c
#include <SDL.h>

int main(int argc, char **argv)
{
    int ret = -1;
    if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) != 0)
    {
        SDL_Log("Unable to initialize SDL: %s", SDL_GetError());
        return 1;
    }

    SDL_Window *window = SDL_CreateWindow("SDL is active!", 100, 100, 512, 512, 0);
    if (!window)
    {
        SDL_Log("Unable to create window: %s", SDL_GetError());
        goto quit; // this is the only authorized way for using goto in C!
    }

    SDL_Renderer *renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED | SDL_RENDERER_PRESENTVSYNC);
    if (!renderer)
    {
        SDL_Log("Unable to create renderer: %s", SDL_GetError());
        goto quit; // this is the only authorized way for using goto in C!
    }

    ret = 0; // from now on we can exit with 0

    int running = 1;
    while (running)
    {
        SDL_Event event;
        while (SDL_PollEvent(&event))
        {
            if (event.type == SDL_QUIT)
            {
                running = 0;
            }
        }
        SDL_RenderPresent(renderer);
    }

quit:
    SDL_Quit();
    return ret;
}
```

Current Status

- We have a Window assigned to our application
- We asked for a SwapChain to the compositor
- At each frame we dequeue events and ‘swap’ the back and front buffer (‘Present’ operation)

Painting in red

```
while (running)
    {
        SDL_Event event;
        while (SDL_PollEvent(&event))
        {
            if (event.type == SDL_QUIT)
            {
                running = 0;
            }
        }
        SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
        SDL_RenderClear(renderer);
        SDL_RenderPresent(renderer);
    }
```

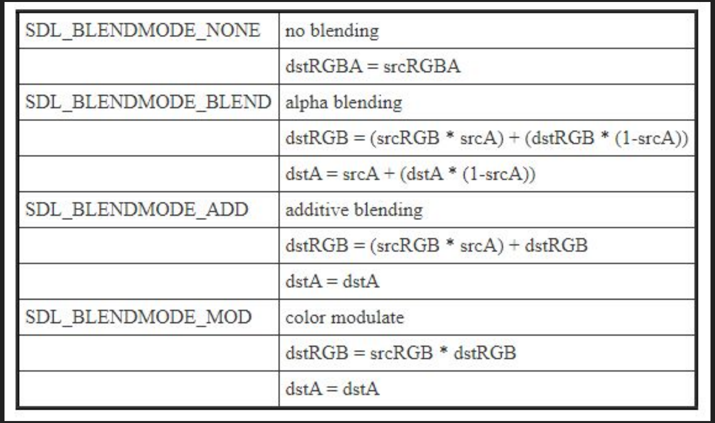
Simple point/line based drawing

```
while (running)
    {
        SDL_Event event;
        while (SDL_PollEvent(&event))
        {
            if (event.type == SDL_QUIT)
            {
                running = 0;
            }
        }

        SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
        SDL_RenderClear(renderer);

        SDL_SetRenderDrawColor(renderer, 0, 255, 255, 255);
        SDL_RenderDrawLine(renderer, 100, 100, 300, 300);

        SDL_SetRenderDrawColor(renderer, 0, 255, 0, 255);
        SDL_Rect rect = {300, 300, 200, 200};
        SDL_RenderDrawRect(renderer, &rect);

        SDL_RenderPresent(renderer);
    }
```

Rasterized triangles (for drawing filled quads) and Blending

```
while (running)
    {
        SDL_Event event;
        while (SDL_PollEvent (&event))
        {
            if (event.type == SDL_QUIT)
            {
                running = 0;
            }
        }

        SDL_SetRenderDrawBlendMode (renderer, SDL_BLENDMODE_NONE );
        SDL_SetRenderDrawColor (renderer, 255, 0, 0, 255);
        SDL_RenderClear (renderer);

        SDL_SetRenderDrawColor (renderer, 0, 255, 255, 255);
        SDL_RenderDrawLine (renderer, 100, 100, 300, 300);

        SDL_SetRenderDrawColor (renderer, 0, 255, 0, 255);
        SDL_Rect rect = {300, 300, 200, 200};
        SDL_RenderDrawRect (renderer, &rect);

        SDL_SetRenderDrawBlendMode (renderer, SDL_BLENDMODE_BLEND );
        SDL_SetRenderDrawColor (renderer, 0, 255, 0, 50);
        rect.x = 10;
        rect.y = 10;
        SDL_RenderFillRect (renderer, &rect);

        SDL_RenderPresent (renderer);
    }
```

Blending Table



SDL_BLENDMODE_NONE	no blending
	dstRGBA = srcRGBA
SDL_BLENDMODE_BLEND	alpha blending
	dstRGB = (srcRGB * srcA) + (dstRGB * (1-srcA))
	dstA = srcA + (dstA * (1-srcA))
SDL_BLENDMODE_ADD	additive blending
	dstRGB = (srcRGB * srcA) + dstRGB
	dstA = dstA
SDL_BLENDMODE_MOD	color modulate
	dstRGB = srcRGB * dstRGB
	dstA = dstA

Textures

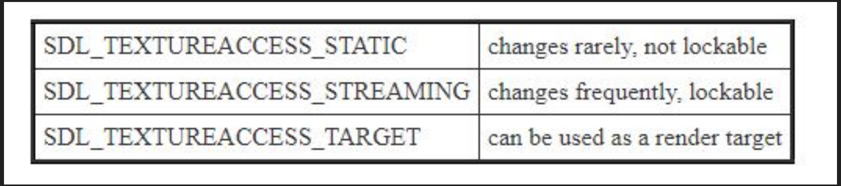
- They are blocks of memory stored in the GPU dedicated memory (if available)
- They have a width a height and a pixel format
- Remember they are not directly accessible by the system, so if you need to update them you need to ask the GPU to copy data (pixels) from the system memory to the GPU memory of the texture
- This is generally accomplished using a ‘staging buffer’, a block of system memory accessible both from your user process as well as the GPU. You fill it from the CPU and the GPU will copy it in the texture memory.
- Even the opposite procedure is available
- When using system memory you need to take caching into account (generally you prefer to turn it off)
- While these will be covered in deep while looking at lower-level GPU apis, we can now safely more-or-less ignore them and rely on the SDL abstractions

Creating a Texture

```
SDL_Texture *texture = SDL_CreateTexture (renderer, SDL_PIXELFORMAT_RGBA32 , SDL_TEXTUREACCESS_STATIC , 128, 128);
```

The PixelFormat defines how color channels are serialized RGBA32 means 8 bits per channel in RGBA order (ABGR is another popular one but check https://wiki.libsdl.org/SDL_PixelFormatEnum)

The TextureAccess can be STATIC, STREAMING or TARGET



SDL_TEXTUREACCESS_STATIC	changes rarely, not lockable
SDL_TEXTUREACCESS_STREAMING	changes frequently, lockable
SDL_TEXTUREACCESS_TARGET	can be used as a render target

Uploading data (pixels) to a Texture

```
SDL_Texture *texture = SDL_CreateTexture (renderer, SDL_PIXELFORMAT_RGBA32 , SDL_TEXTUREACCESS_STATIC , 128, 128);
unsigned int pixels[] = {0xAA00FF00, 0xAAFF0000};
SDL_Rect area_to_update = {0, 0, 2, 1};
SDL_UpdateTexture (texture, &area_to_update, pixels, 128 * 4);
```

```
SDL_Texture *texture2 = SDL_CreateTexture (renderer, SDL_PIXELFORMAT_RGBA32 , SDL_TEXTUREACCESS_STREAMING , 128,
128);
unsigned int* pixels_ptr;
int pitch;
SDL_LockTexture (texture2, NULL, (void**)&pixels_ptr, &pitch);
pixels_ptr[0] = 0xAA00FF00;
pixels_ptr[1] = 0xAAFF0000;
SDL_UnlockTexture (texture2);
```

Streaming Textures

- At texture creation time a chunk of (malloc’ed) memory with the size of the texture is created
- When you lock you get the pointer to that area
- When you unlock the malloc’ed area is transferred to the GPU (generally involving a staging buffer [it depends on the GPU library/driver/hardware])
- If you do not have really specific reasons (like video playing) avoid streaming textures! (they could put the texture’s data in system memory, making the GPU processing slower)

Blitting Textures

```
SDL_Texture *texture = SDL_CreateTexture(renderer, SDL_PIXELFORMAT_RGBA32, SDL_TEXTUREACCESS_STATIC, 128, 128);
unsigned int pixels[] = {0xAA00FF00, 0xAAFF0000};
SDL_Rect area_to_update = {0, 0, 2, 1};
SDL_UpdateTexture(texture, &area_to_update, pixels, 128 * 4);
SDL_SetTextureAlphaMod(texture, 255);
SDL_SetTextureBlendMode(texture, SDL_BLENDMODE_BLEND);

while (running)
{
    SDL_Event event;
    while (SDL_PollEvent(&event))
    {
        if (event.type == SDL_QUIT)
        {
            running = 0;
        }
    }

    SDL_SetRenderDrawColor(renderer, 255, 0, 0, 255);
    SDL_RenderClear(renderer);

    SDL_Rect texture_rect = {0, 0, 2, 2};
    SDL_Rect target_rect = {100, 100, 256, 256};
    SDL_RenderCopy(renderer, texture, &texture_rect, &target_rect);

    SDL_RenderPresent(renderer);
}
```

Managing time

Operating systems (generally) exposes 4 types of time measures:

- Wall clock: this is time as seen by humans (known as the “real time” clock on some platform). Think about it: can increment or decrement. You can overwrite it (assuming you have the right privileges)
- Monotonic: a counter that increments (always!, it never decrements) at a specific rate.
- Performance Counter: A high resolution counter (generally in nanosecond resolution). It always increments. Generally the best choice for gaming.
- Process time: the time the cpu has executed a specific process

Time in SDL

void SDL_Delay(unsigned int ms);

unsigned long long SDL_GetPerformanceCounter();

unsigned long long SDL_GetPerformanceFrequency();

unsigned int SDL_GetTicks(); // from the SDL_Init call. wraps every 49.7 days

unsigned long long SDL_GetTicks64(); // from the SDL_Init call

Heap and Filesystem

void* SDL_malloc(size_t);

void SDL_free(void*);

void* SDL_calloc(size_t, size_t);

```
SDL_RWops *rw = SDL_RWFromFile("test.bin", "r");
if (rw != NULL)
{
    Uint8 buf[256];
    SDL_RWread(rw, buf, sizeof(buf), 1);
    SDL_RWclose(rw);
}
```

Gamepads (and Joysticks)

```c
SDL_GameController *controller = NULL;
for (int i = 0; i < SDL_NumJoysticks(); ++i)
{
    if (SDL_IsGameController(i))
    {
        controller = SDL_GameControllerOpen(i);
        if (controller)
        {
            SDL_Log("Found Controller %s", SDL_GameControllerName(controller));
            break;
        }
        else
        {
            SDL_Log("Could not open gamecontroller %d: %s\n", i, SDL_GetError());
        }
    }
}
```

Remember to add `SDL_INIT_GAMECONTROLLER` in SDL_Init()!

Audio

TODO

EOF