I. call code = involves saving volatile registers (EAX, ECX, EDX), passing parameters and saving the return address.

Ex:

```
push EAX
push ECX      |-> volatile resources are saved by pushing them onto the stack

push EAX
push format   |=> parameters are passed by pushing them onto the stack from right to left.

call [printf] |-> the return address is automatically saved onto the stack by the "call" instruction
add esp, 2
```

entry code = involves setting up a new stack frame, allocating space for variables and saving non-volatile resources.

Ex:

```
push EBP
mov EBP, ESP  |=> a new stackframe is built in order to maintain a reference point for passing function parameters
sub esp, 4    |-> allocation of space on the stack for data to be stored.
push ebx      |-> saving non-volatile resources to be retrieved later.
```

exit code = involves reversing all the actions performed during entry code: restoring non-volatile resources, freeing space allocated for data, deallocating the stack and returning from the function and releasing arguments.

```
pop ebx
add esp, 4    - proper restauration of the stack and registers.
pop ebp       - helps maintaining stack integrity and preventing unintended side effects.
ret 4.
```

CDECL

- stands for "C declaration" and represents a standard calling convention in the C programming language. Any parameters, but extended at least to dword can be pushed onto the stack from right to left, and the caller is responsible for clearing up the stack. Return values are usually stored in EAX, EDX:EAX and ST0 (FPU). Additionaly, certain registers like EAX, ECX, EDX are considered volatile:
- simplicity + compatibility + any parameters.
- used in, ~~win~~ Standard C libraries, Cross-Platform Development.

STDCALL

- stands for "Standard Call" and represents a standard calling convention in the C programming language. (Winapi on Windows Operating system)
- resembles the "CDECL" calling convention, with the only difference being that there is a fixed number of parametes and the callee does the cleaning of the stack.
- standardized and efficient way for Windows system libraries to communicate.

The assembly programer is responsable for writing the entry and exit code ~~in Assembly~~ for each assembly function and the C programmer is responsible for writing the call code in C.

The compiler plays a crucial role in translating the high level 'C' code in machine instructions, ensuring that the calling conventions are followed during the function calls.

When linking a C module with an assembly module, the linker combines the compiled code from both languages into a single executable, aligning the entry, exit and call code correctly based on the specific calling convention.

ASM + ASM:

- global variables from one asm module can be accessed by other ASM modules:

ex: module1.asm
```
segment data:
    a dd 0
```
module2.asm
```
extern a
```

- asm modules can communicate using jumps and labels: (jumps instructions in one module and labels in the other).

ex: module1.asm
```
my_label:
    ; code.
```
module2.asm
```
jmp my_label.
```

ASM + C:

- functions defined in ASM modules can be called from C modules and vice-versa
- parameters can be passed between C and asm modules using specific ~~instructions~~ registers or the stack.
- functions can return values to their callers. The return value is typically stored in a register or as specific memory location