POO

Gavrilut Dragos

Summary

- Administrative
- Glossary
- Compilers
- OS architecture
- C++ history and revisions
- C++ compilers
- C++ grammar

Administrative

- Site: https://sites.google.com/site/fiicoursepoo/
- Partea de laborator in POO consta in 3 examene in timpul laboratorului, distribuite astfel:
  - 3-4 laboratoare se fac diverse probleme bazate pe ce se preda la curs
  - Sunt urmate apoi de un test practic (~40 minute) in care trebuie sa implementeze o problema bazata pe ce s-a predate pana in acel moment
  - Programare acestor examene se mai poate schimba pe parcurs. Urmariti tot timpul pagina cursului pentru detalii legate de acest lucru.
- Numarul maxim de absente nemotivate cu care se poate promova laboratorul este de 2.
- La finalul cursului o sa mai fie un test teoretic din notiunile predate.

Administrative

- Examenele de laborator au loc in acelasi timp pentru mai multe grupe si in acelasi timp pentru tot anul.
- Primul si al treilea examen de laborator se vor da in ziua de sambata din saptamana in care trebuia sa se dea examenul. In acea saptamana nu se mai face laboratorul.
- Al doilea examen de laborator are loc in saptamana a 8-a (care este oricum saptamana de examen) si se va da intr-o zi din acea saptamana cand gasim cat mai multe laboratoare libere.
- Datele exacte si orele la care fiecare grupa trebuie sa fie prezenta pentru examen se vor anunta inainte de examen. Examenele de laborator se pot da inainte sau dupa data stabilita doar in cazuri exceptionale si trebuie aduse la cunostinta conducatorilor cursului inainte de examen. Absenta la un examen de laborator se puncteaza cu 0 puncte. In acest caz examenul nu se mai poate da inca o data

Administrative

- Formatul examenului de laborator e in felul urmator:
  - fiecare grupa se imparte in doua semigrupe (aprox. 15 studenti pe semigrupa) - in ordine alfabetica
  - Daca o grupa are examenul de la 08:00 la 10:00, atunci semigrupa 1 va veni la ora 08:00 iar semigrupa 2 la 09:00
  - Examenul se va da in fata calculatorului si consta intr-o problema care trebuie rezolvata in 40 de minute.
  - Nota de laborator se pune pe loc, in restul de 20 de minute din ora respectiva
  - Pentru fiecare problema va exista un barem de corectare pe care o sa il prezentam ulterior testului.

Administrative

- Nota finala se calculeaza in felul urmator:

Prezenta minima la laborator	**10 pct**
Test 1 laborator	**15 pct**
Test 2 laborator	**25 pct**
Test 3 laborator	**15 pct**
Test final din materia de curs	**35 pct**

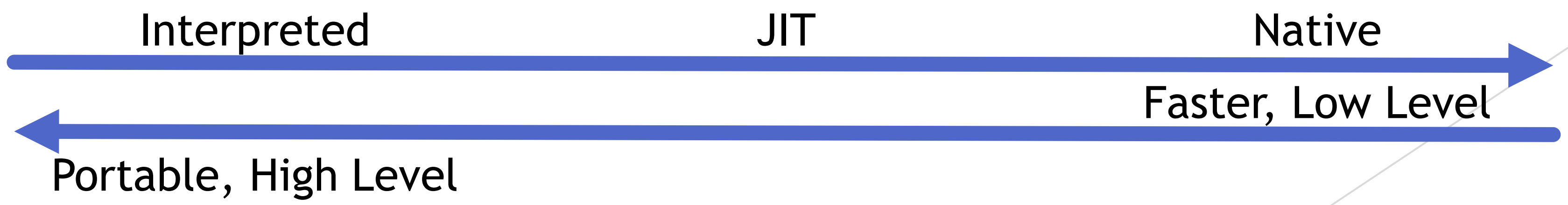
- Mai multe detalii pe pagina cursului la sectiunea administrative.

Glossary

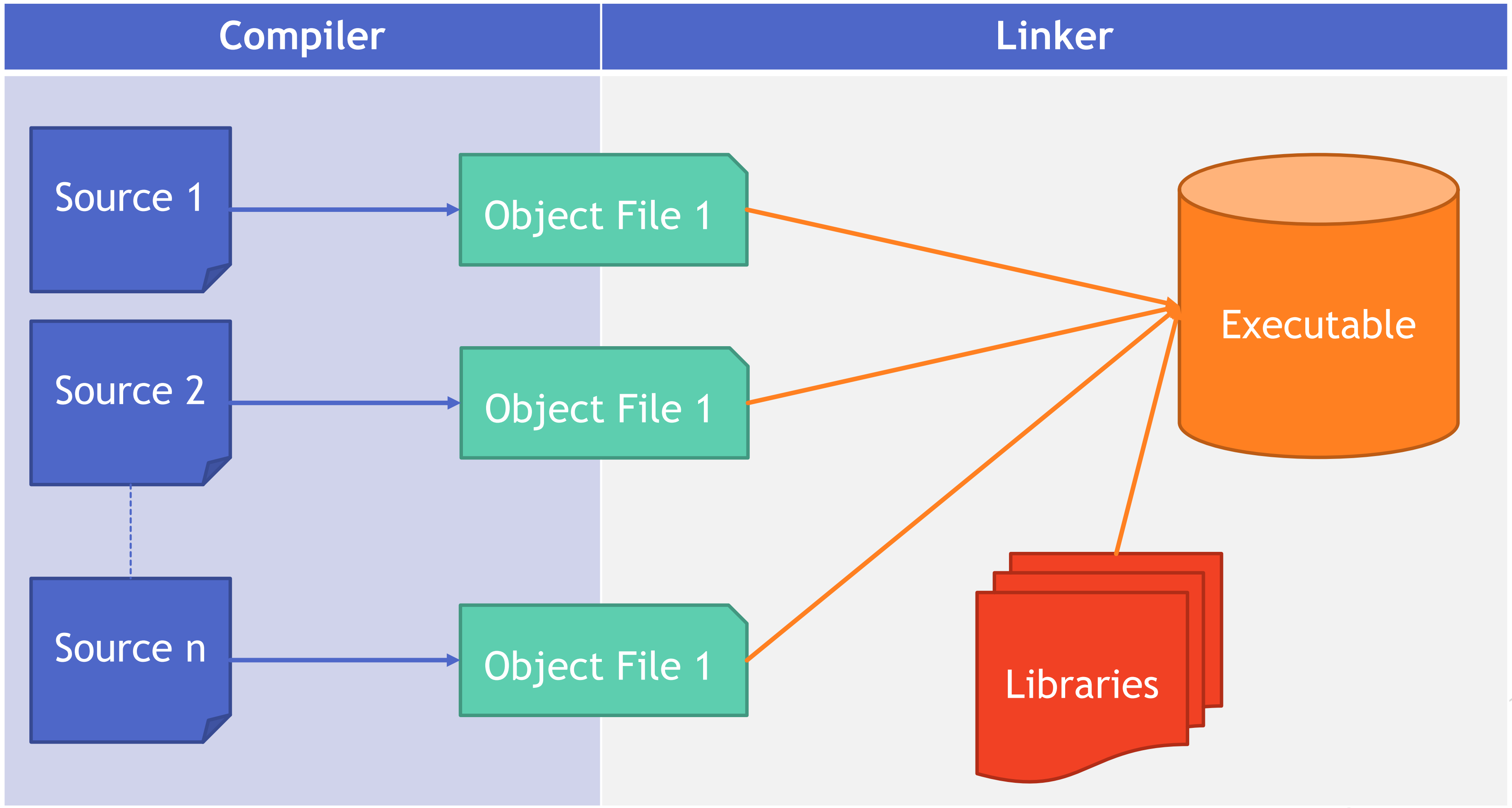
- API → <u>A</u>pplication <u>P</u>rogram <u>I</u>nterface
- Library – a set o functions that can be use by multiple programs at the same time (for example math functions like cos, sin, tan, etc)
- GUI → <u>G</u>raphic <u>U</u>ser <u>I</u>nterface

Glossary

- Compiler – a program that translates from a source code (a readable code) into a machine code (binary code that is understand by a specific architecture – x86, x64, ARM, etc)
- A compiler can be:
  - Native – the result is a native code application for the specific architecture
  - Interpreted – the result is a code (usually called byte-code) that requires an interpreter to be executed. It's portability depends on the portability of its interpreter
  - JIT (Just In Time Compiler) – the result is a byte-code, but during the execution parts of this code are converted to native code for performance

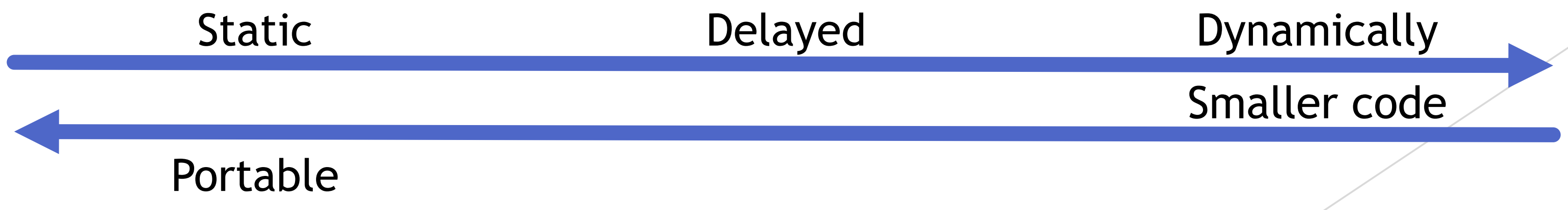


Glossary


Compiler
Linker
Source 1
Source 2
Source n
Object File 1
Object File 1
Object File 1
Executable
Libraries

Glossary

- Linker – a program that merges the object files obtained from the compiler phase into a single executable
- It also merges various libraries to the executable that is being create.
- Libraries can be linked in the following ways:
  - Dynamically: When application is executed, the operating system links it with the necessary libraries (if available). If not an execution error may appear.
  - Static: The resulted executable code contains the code from the libraries that it uses as well
  - Delayed: Similar with the Dynamic load, but the libraries are only loaded when the application needs one function (and not before that moment).



OS Architecture

- What happens when the OS executes a native application that is obtain from a compiler such as C++ ?
- Let’s consider the following C/C++ file that is compile into an executable application:

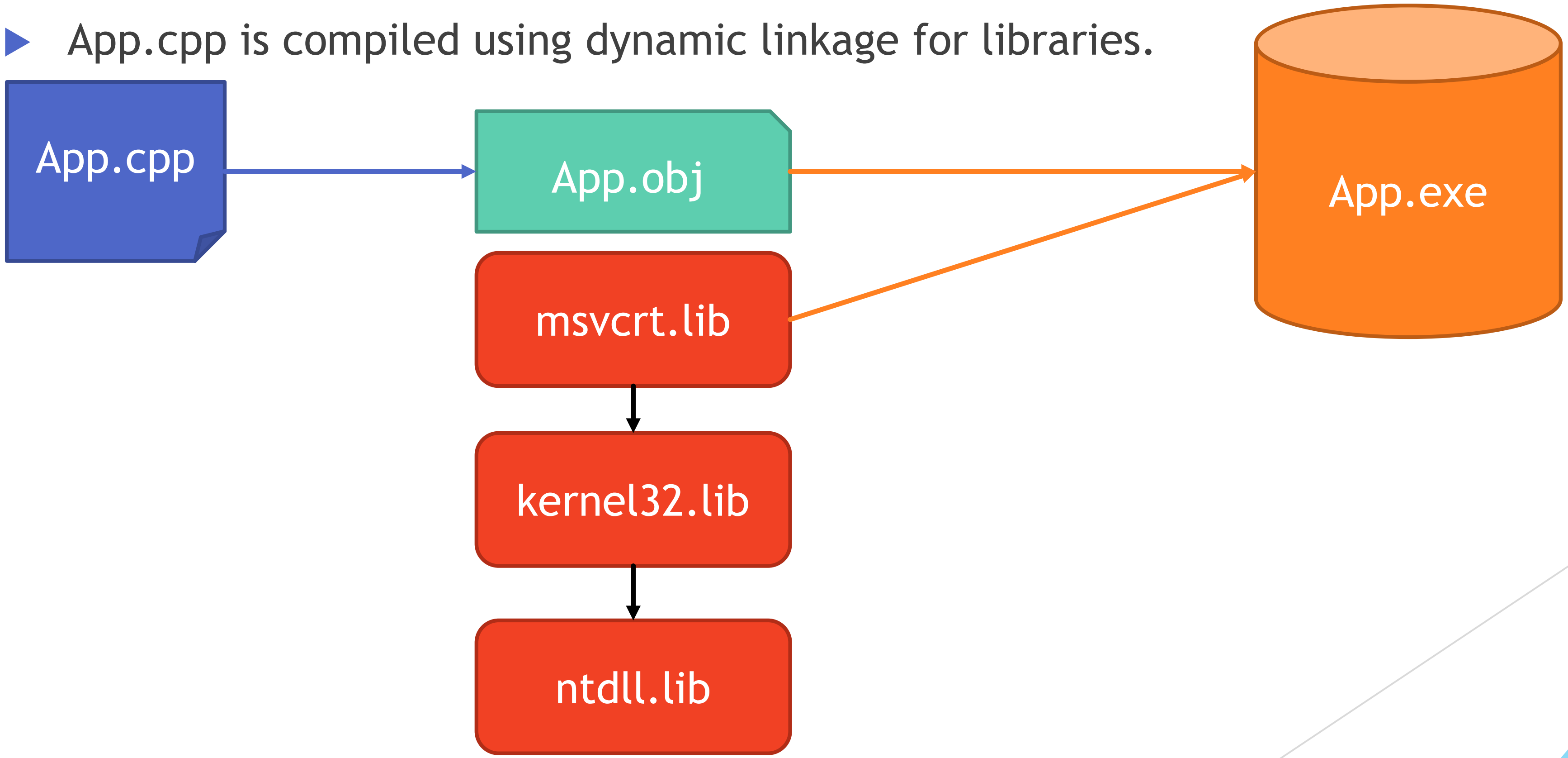
App.cpp

```
#include <stdio.h>
int vector[100];

bool IsNumberOdd(int n) {
    return ((n % 2)==0);
}
void main(void) {
      int poz,i;
      for (poz=0,i=1;poz<100;i++) {
            if (IsNumberOdd(i)) {
                  vector[poz++] = i;
            }
      }
      printf("Found 100 odd numbers !");
}
```

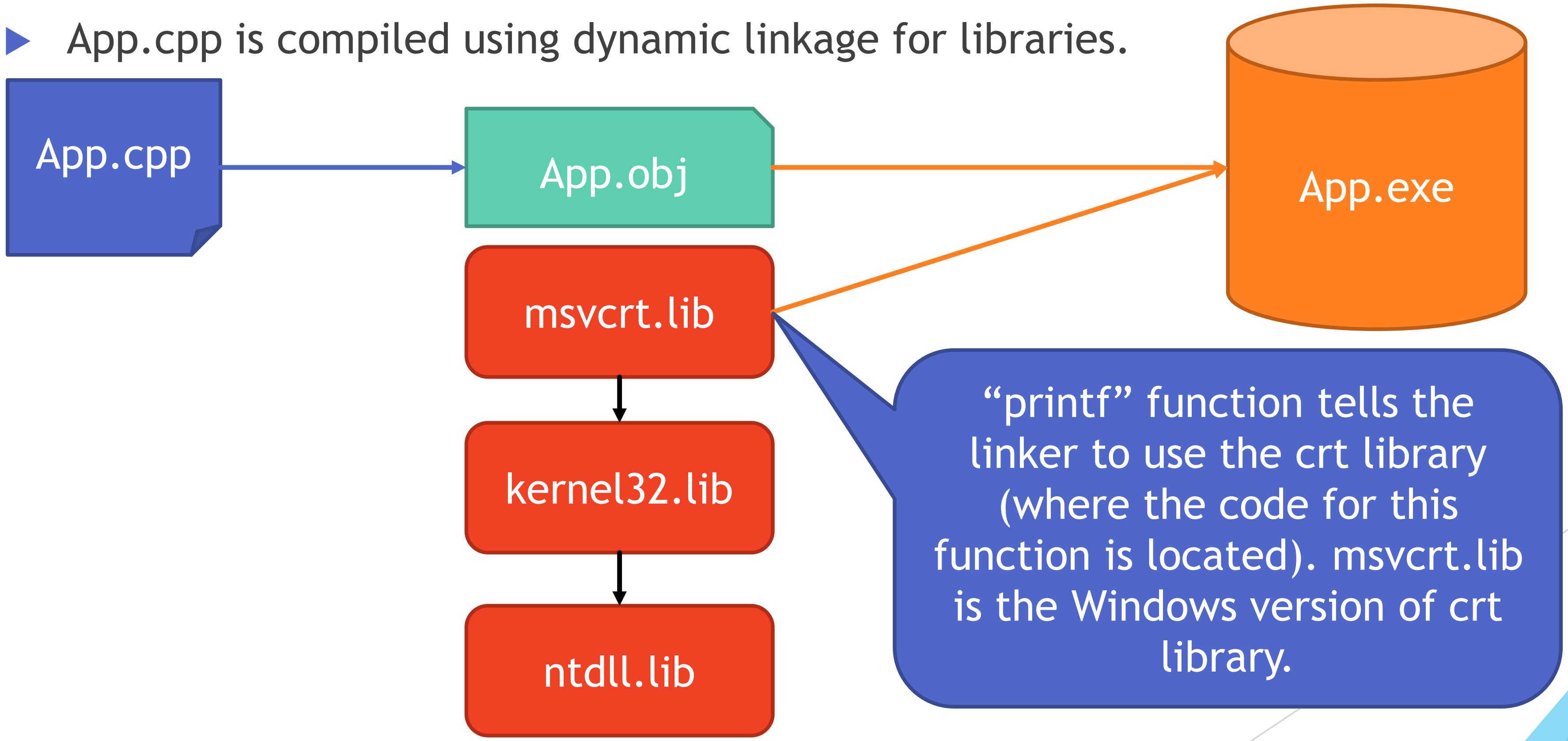
OS Architecture

- Let’s assume that we compile “App.cpp” on a Windows system using Microsoft C++ compiler (cl.exe).
- App.cpp is compiled using dynamic linkage for libraries.



OS Architecture

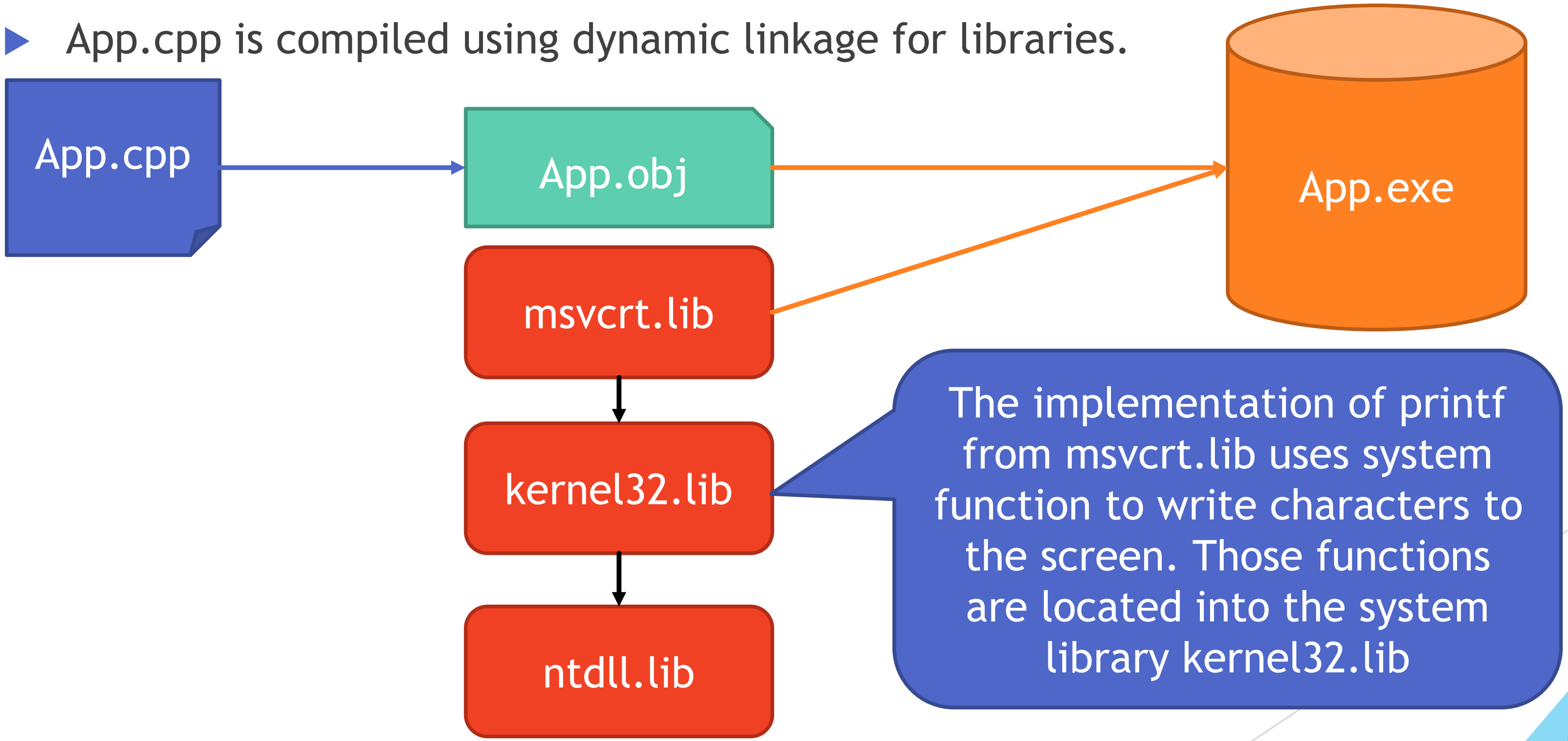
- Let’s assume that we compile “App.cpp” on a Windows system using Microsoft C++ compiler (cl.exe).
- App.cpp is compiled using dynamic linkage for libraries.



“printf” function tells the linker to use the crt library (where the code for this function is located). msvcrt.lib is the Windows version of crt library.

OS Architecture

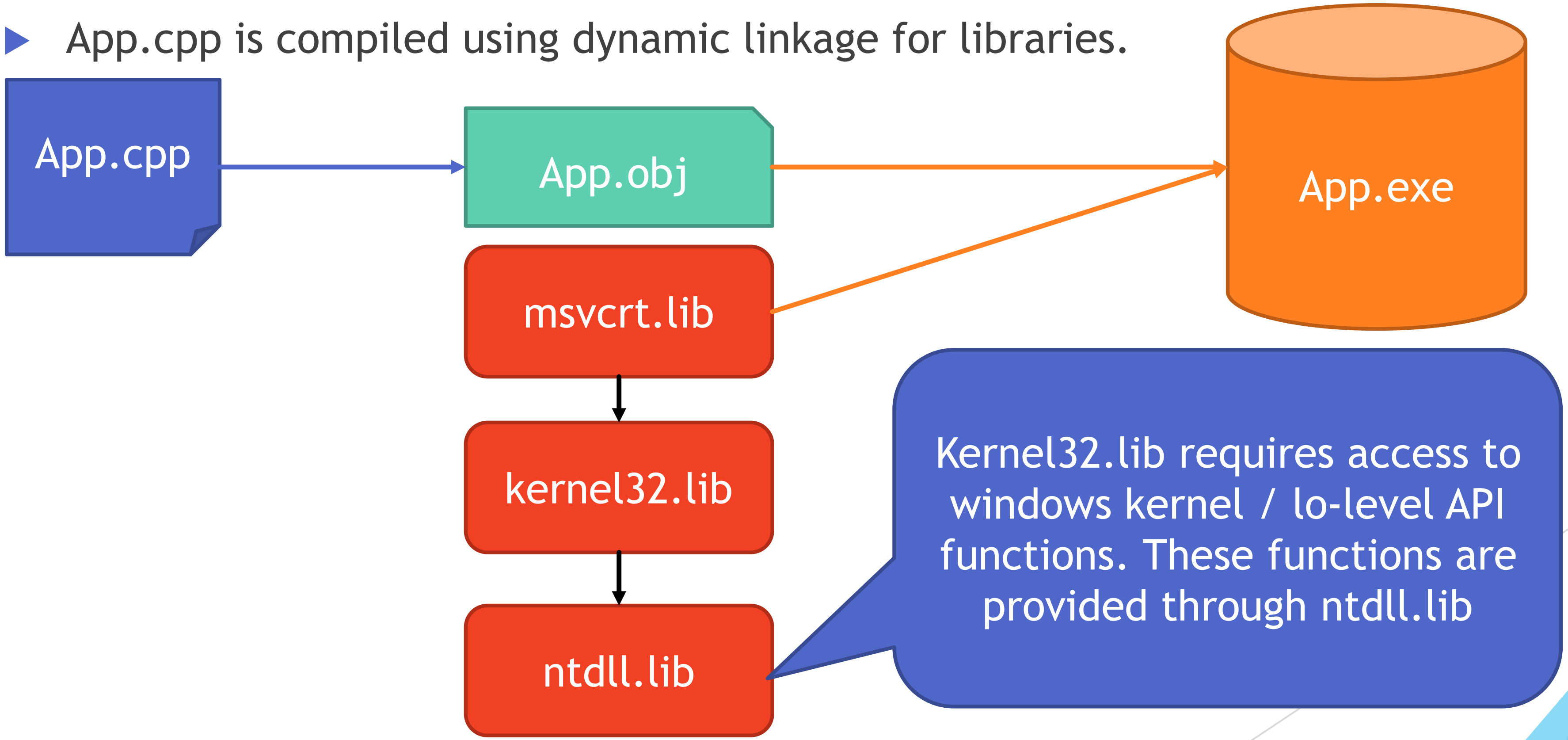
- Let’s assume that we compile “App.cpp” on a Windows system using Microsoft C++ compiler (cl.exe).
- App.cpp is compiled using dynamic linkage for libraries.



The implementation of printf from msvcrt.lib uses system function to write characters to the screen. Those functions are located into the system library kernel32.lib

OS Architecture

- Let’s assume that we compile “App.cpp” on a Windows system using Microsoft C++ compiler (cl.exe).
- App.cpp is compiled using dynamic linkage for libraries.

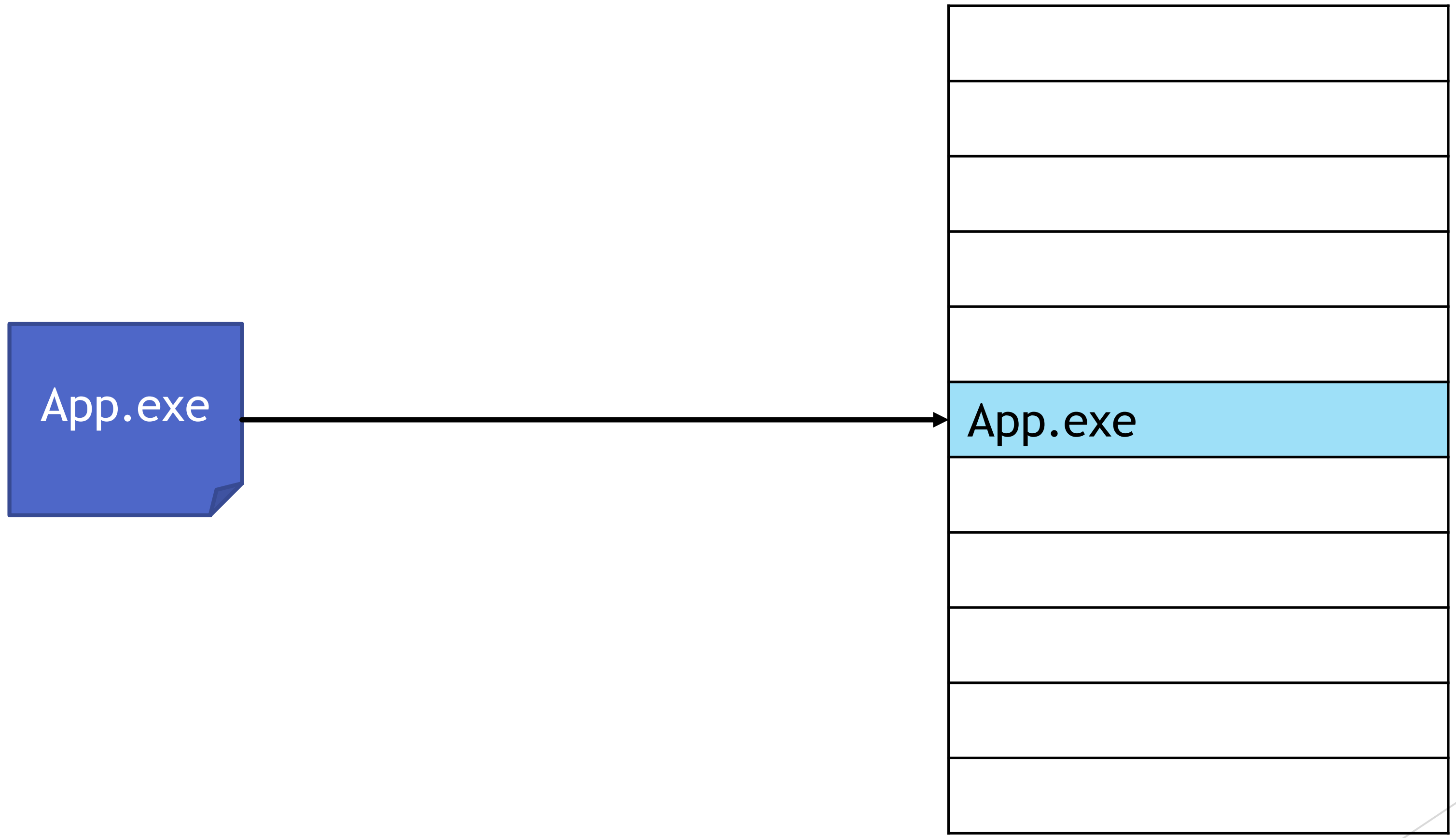

App.cpp
App.obj
App.exe
msvcrt.lib
kernel32.lib
ntdll.lib
Kernel32.lib requires access to windows kernel / lo-level API functions. These functions are provided through ntdll.lib

OS Architecture

- What happens when a.exe is executed:

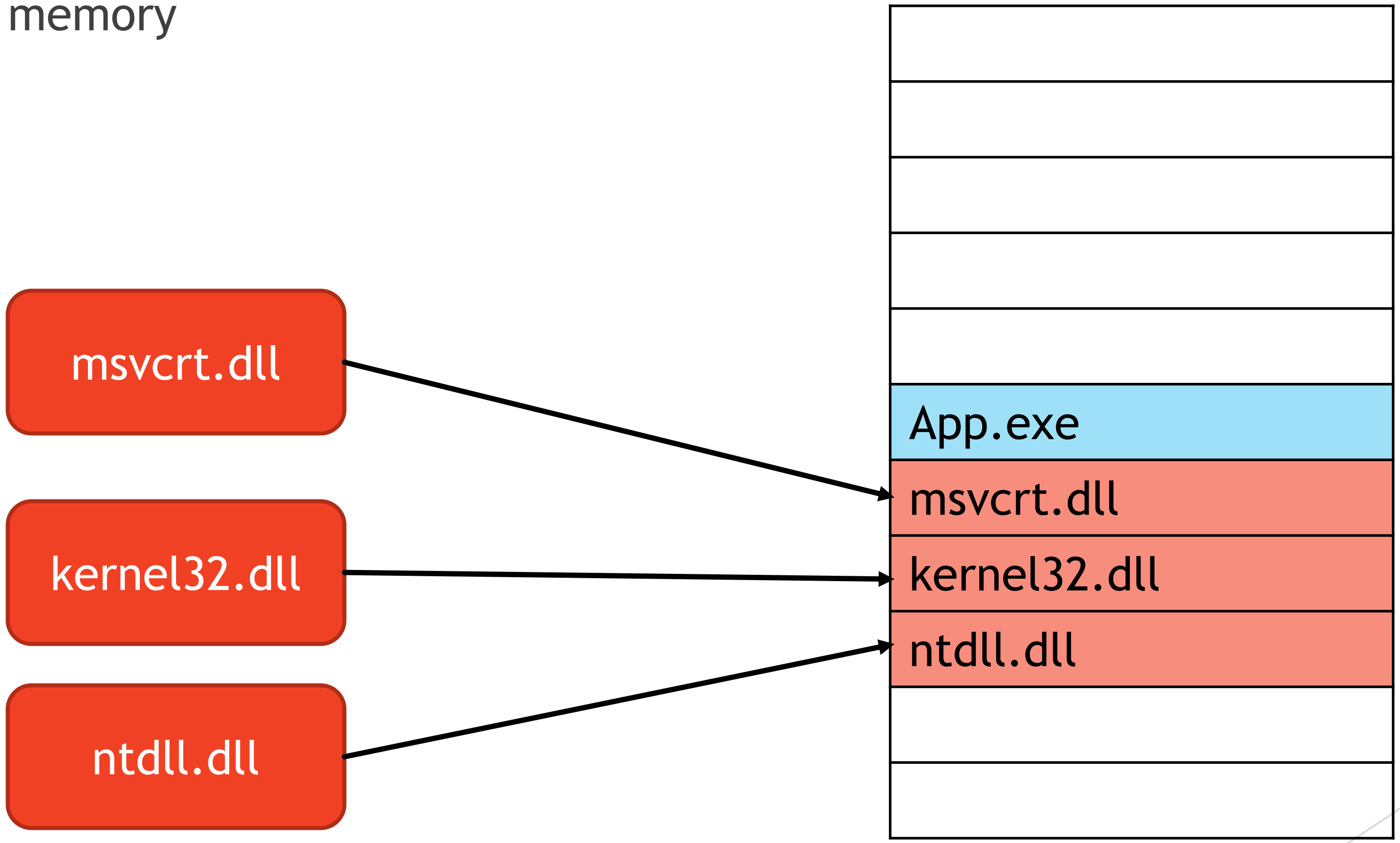
OS Architecture

- Content of “app.exe” is copied in the process memory



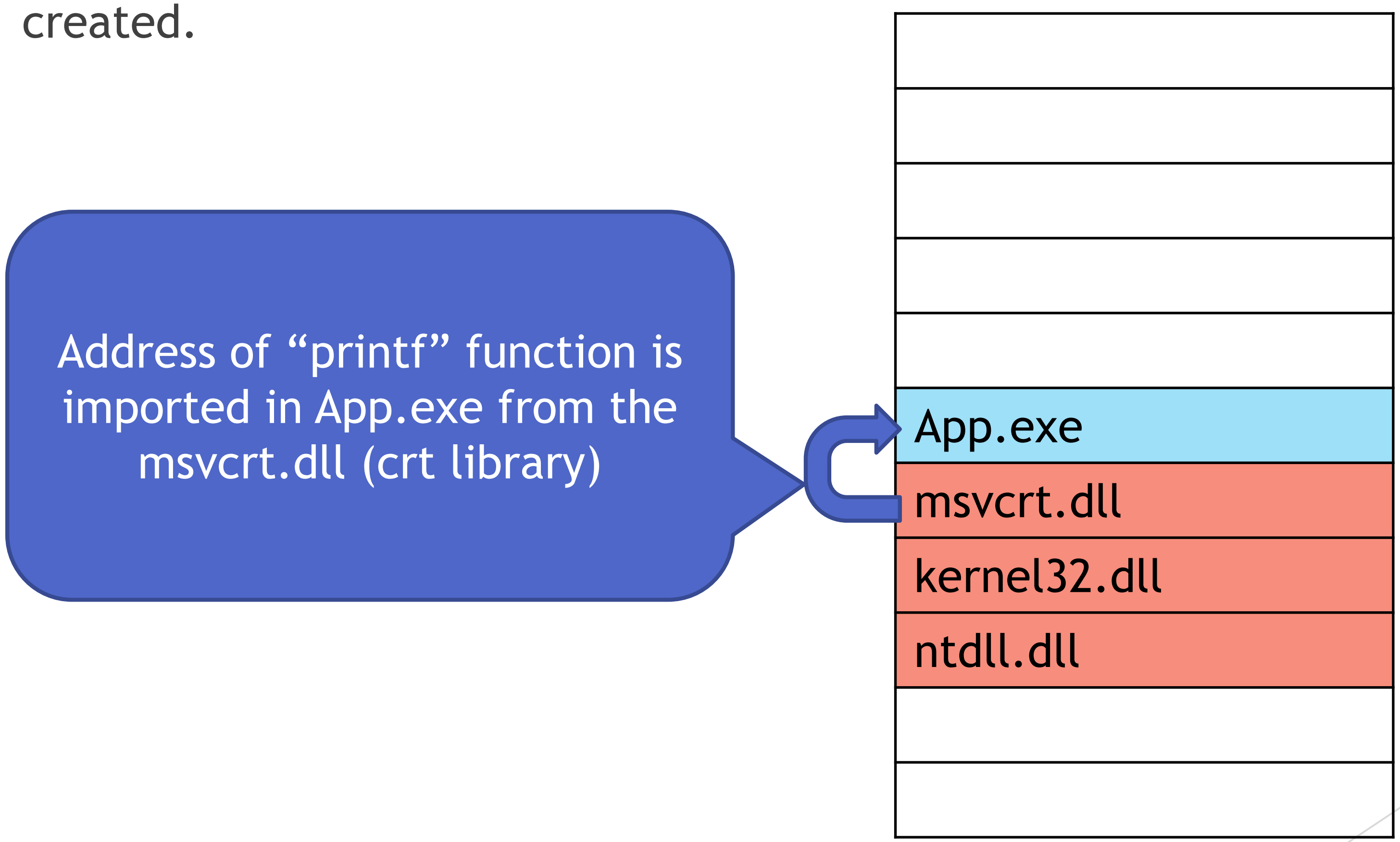
OS Architecture

- Content of the libraries that are needed by “a.exe” is copied in the process memory



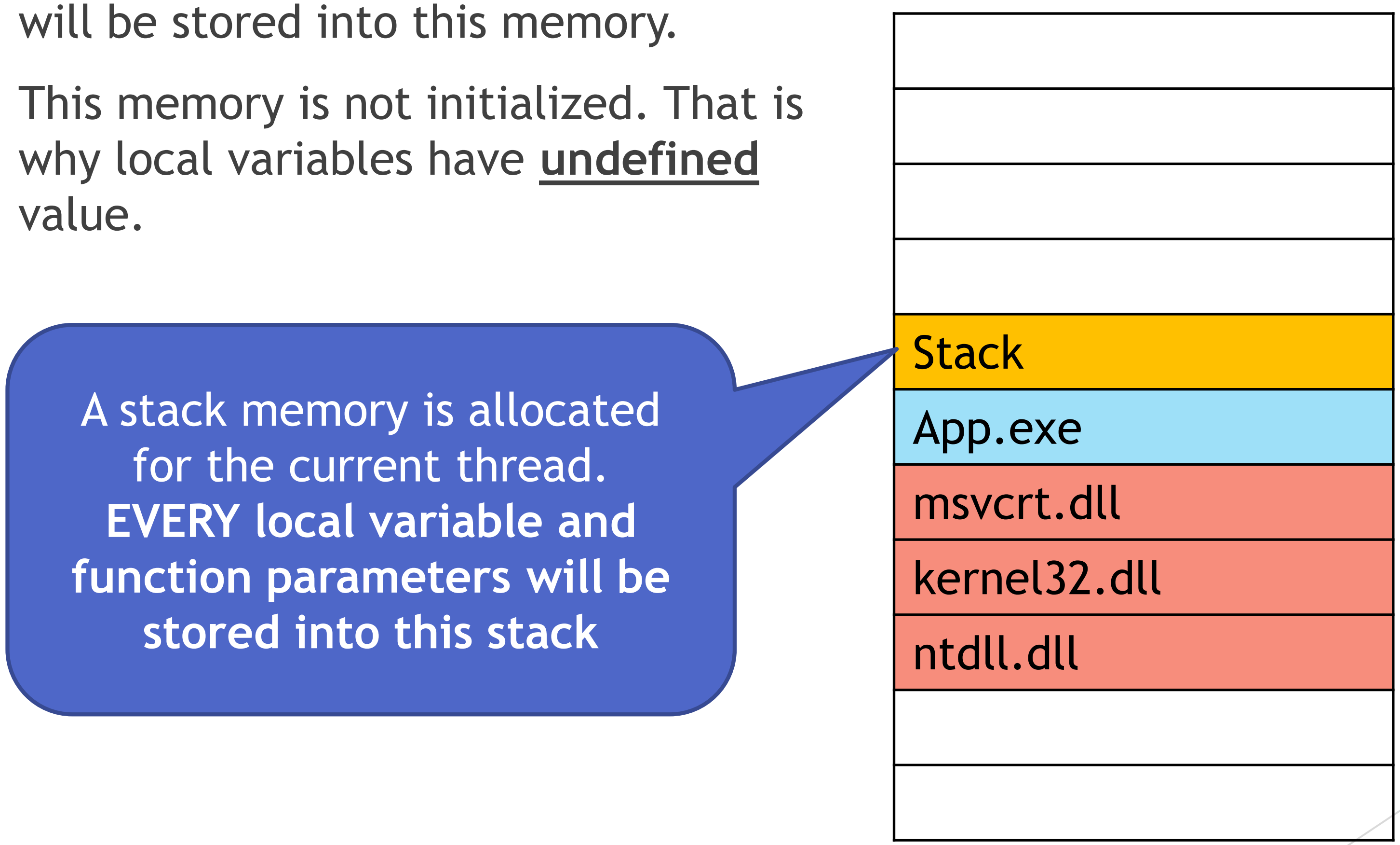
OS Architecture

- References to different functions that are needed by the main module are created.


Address of “printf” function is imported in App.exe from the msvcrt.dll (crt library)
App.exe
msvcrt.dll
kernel32.dll
ntdll.dll

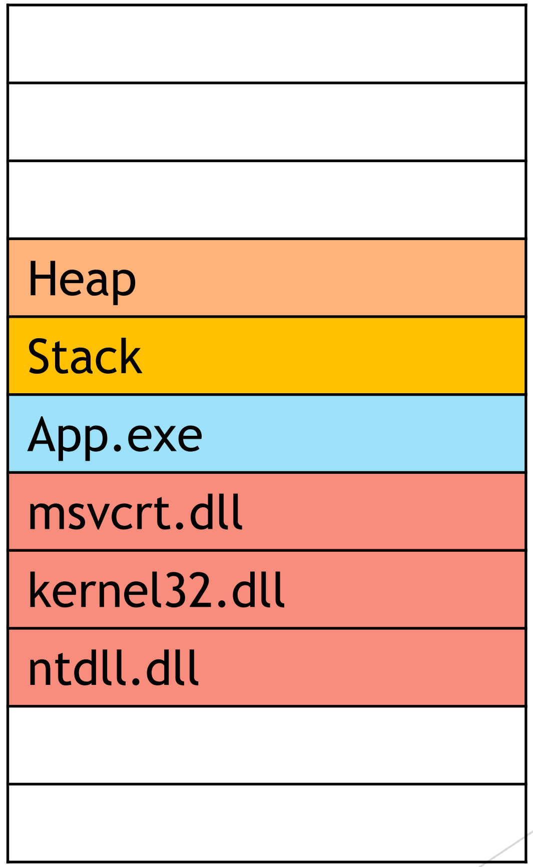
OS Architecture

- Stack memory is created. In our example, variable **poz**, **i,** and parameter **n** will be stored into this memory.
- This memory is not initialized. That is why local variables have **undefined** value.


A stack memory is allocated for the current thread.
EVERY local variable and function parameters will be stored into this stack
Stack
App.exe
msvcrt.dll
kernel32.dll
ntdll.dll

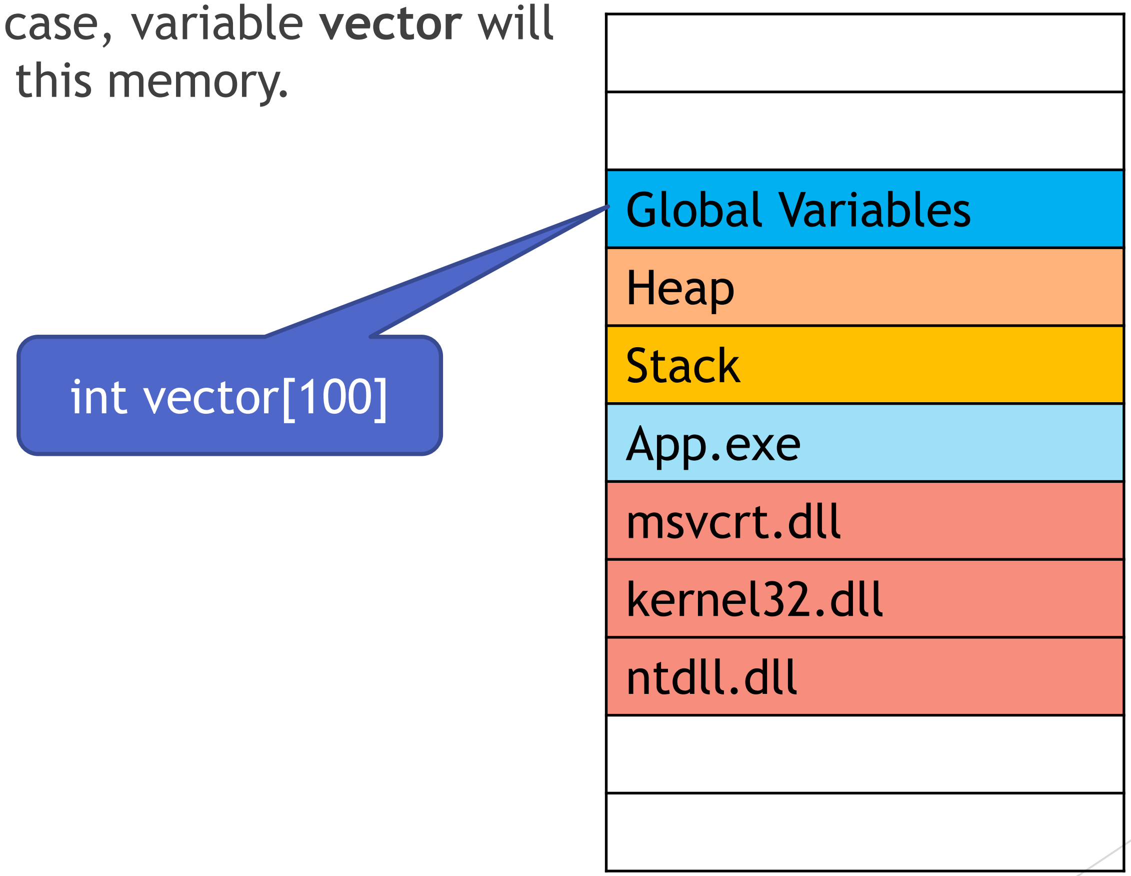
OS Architecture

- Heap memory is allocated. Heap memory is large memory from where smaller buffers are allocated. Heap is used by the following functions:
  - Operator **new**
  - **malloc, calloc, etc**
- **Heap memory is not initialized.**



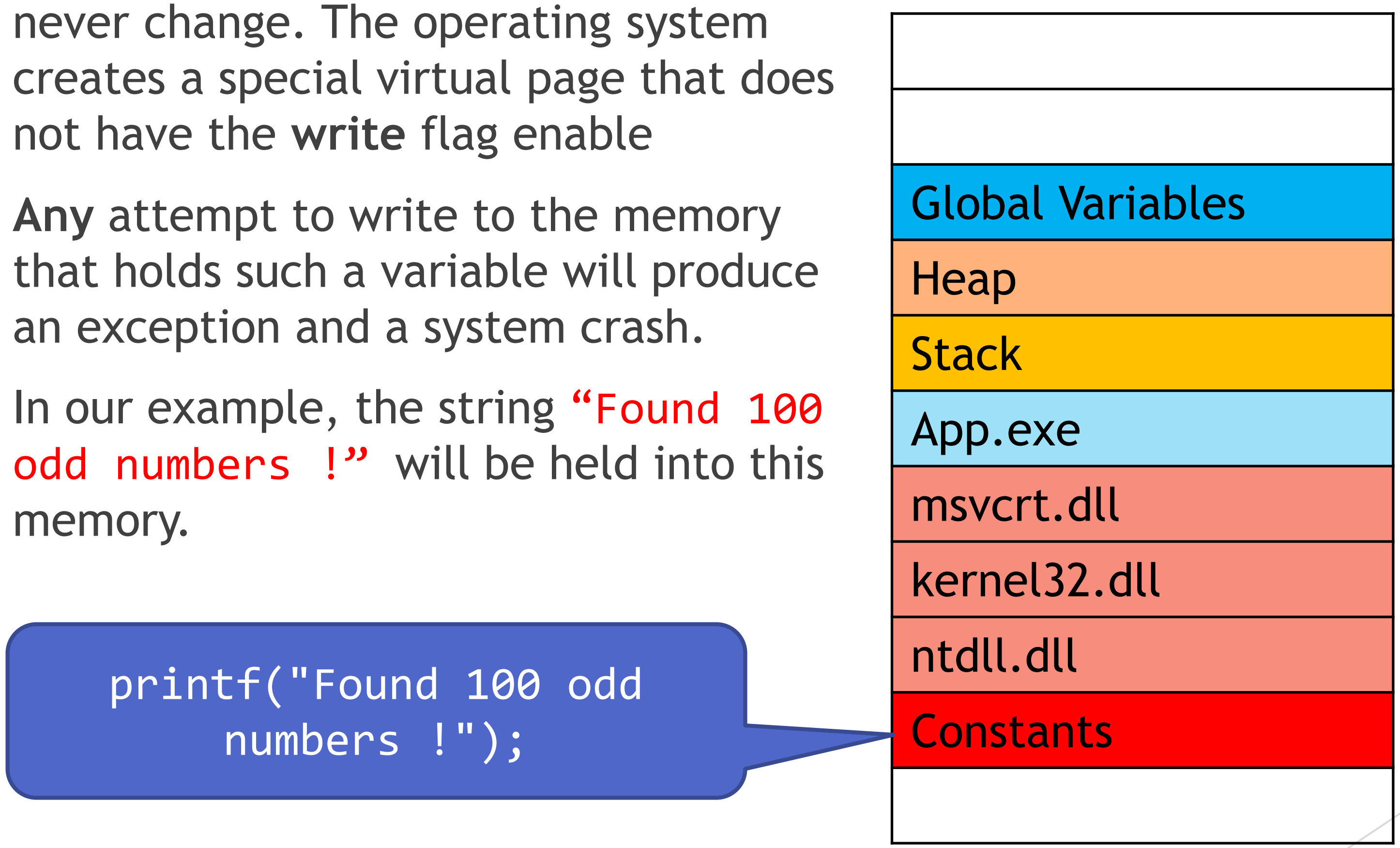
OS Architecture

- A memory for global variable is allocated. This memory is initialized with 0 values. In our case, variable **vector** will be stored into this memory.



OS Architecture

- A memory for constant data is created. This memory holds data that will never change. The operating system creates a special virtual page that does not have the **write** flag enable
- **Any** attempt to write to the memory that holds such a variable will produce an exception and a system crash.
- In our example, the string “`Found 100 odd numbers !`” will be held into this memory.



```
printf("Found 100 odd
numbers !");
```

OS Architecture

- Let’s consider the following example:

App.cpp

```
void main (void)
{
      char s1,s2,s3;
      char *p;
      s1 = 'a';
      s2 = 'b';
      s3 = 'c';
      p = &s1;
      *p = '0';
      p[1] = '1';
      *(p+2) = '2';
}
```

OS Architecture

- The program has 4 variable (3 of type char –'a','b' and 'c' and a pointer 'p').
- Let's consider that the stack start at the physical address 100

App.cpp

```
void main (void)
{
	char s1,s2,s3;
	char *p;
	s1 = 'a';
	s2 = 'b';
	s3 = 'c';
	p = &s1;
	*p = '0';
	p[1] = '1';
	*(p+2) = '2';
}
```

Stack Address	Var
99	(s1)
98	(S2)
97	(s3)
93	(p)

OS Architecture

- Let’s also consider the following pseudo code that mimic the behavior of the original code

App.cpp

```
void main (void)
{
        char s1,s2,s3;
        char *p;
        s1 = 'a';
        s2 = 'b';
        s3 = 'c';
        p = &s1;
        *p = '0';
        p[1] = '1';
        *(p+2) = '2';
}
```

Pseudo - code

Stack Address	Var
99	(s1)
98	(S2)
97	(s3)
93	(p)

OS Architecture

- Upon execution – the following will happen:

App.cpp

```
void main (void)
{
        char s1,s2,s3;
        char *p;
        s1 = 'a';
        s2 = 'b';
        s3 = 'c';
        p = &s1;
        *p = '0';
        p[1] = '1';
        *(p+2) = '2';
}
```

Pseudo - code
`Stack[99] = 'a'`

Stack Address	Value
99	‘a’
98	?
97	?
93	?

OS Architecture

- Upon execution – the following will happen:

App.cpp

```
void main (void)
{
        char s1,s2,s3;
        char *p;
        s1 = 'a';
        s2 = 'b';
        s3 = 'c';
        p = &s1;
        *p = '0';
        p[1] = '1';
        *(p+2) = '2';
}
```

Pseudo - code
`Stack[99] = 'a'`
`Stack[98] = 'b'`

Stack Address	Value
99	‘a’
98	‘b’
97	?
93	?

OS Architecture

- Upon execution – the following will happen:

App.cpp

```
void main (void)
{
        char s1,s2,s3;
        char *p;
        s1 = 'a';
        s2 = 'b';
        s3 = 'c';
        p = &s1;
        *p = '0';
        p[1] = '1';
        *(p+2) = '2';
}
```

Pseudo - code
`Stack[99] = 'a'`
`Stack[98] = 'b'`
`Stack[97] = 'c'`

Stack Address	Value
99	‘a’
98	‘b’
97	‘c’
93	?

OS Architecture

- Upon execution – the following will happen:

App.cpp

```
void main (void)
{
        char s1,s2,s3;
        char *p;
        s1 = 'a';
        s2 = 'b';
        s3 = 'c';
        p = &s1;
        *p = '0';
        p[1] = '1';
        *(p+2) = '2';
}
```

Pseudo - code
`Stack[99] = 'a'`
`Stack[98] = 'b'`
`Stack[97] = 'c'`
`Stack[93] = 99`

Stack Address	Value
99	‘a’
98	‘b’
97	‘c’
93	99

OS Architecture

- Upon execution – the following will happen:
  Stack[93] = 99, Stack[99] = ‘0’

App.cpp

```
void main (void)
{
        char s1,s2,s3;
        char *p;
        s1 = 'a';
        s2 = 'b';
        s3 = 'c';
        p = &s1;
        *p = '0';
        p[1] = '1';
        *(p+2) = '2';
}
```

Pseudo - code

```
Stack[99] = 'a'
Stack[98] = 'b'
Stack[97] = 'c'
Stack[93] = 99
Stack[Stack[93]] = '0'
```

Stack Address	Value
99	‘0’
98	‘b’
97	‘c’
93	99

OS Architecture

- Upon execution – the following will happen:
  Stack[93] = 99, Stack[99-1] = ‘1’

App.cpp

```
void main (void)
{
        char s1,s2,s3;
        char *p;
        s1 = 'a';
        s2 = 'b';
        s3 = 'c';
        p = &s1;
        *p = '0';
        p[1] = '1';
        *(p+2) = '2';
}
```

Pseudo - code
`Stack[99] = 'a'`
`Stack[98] = 'b'`
`Stack[97] = 'c'`
`Stack[93] = 99`
`Stack[Stack[93]] = '0'`
`Stack[Stack[93]-1] = '1'`

Stack Address	Value
99	‘0’
98	‘1’
97	‘c’
93	99

OS Architecture

- Upon execution – the following will happen:
  Stack[93] = 99, Stack[99-1] = ‘1’

App.cpp

```
void main (void)
{
	char s1,s2,s3;
	char *p;
	s1 = 'a';
	s2 = 'b';
	s3 = 'c';
	p = &s1;
	*p = '0';
	p[1] = '1';
	*(p+2) = '2';
}
```

Pseudo - code
`Stack[99] = 'a'`
`Stack[98] = 'b'`
`Stack[97] = 'c'`
`Stack[93] = 99`
`Stack[Stack[93]] = '0'`
`Stack[Stack[93]-1] = '1'`
`Stack[Stack[93]-2] = '2'`

Stack Address	Value
99	‘0’
98	‘1’
97	‘2’
93	99

OS Architecture (memory alignament)

```
struct Test
{
        int x;
        int y;
        int z;
};
```

```
sizeof(Test) = 12
```

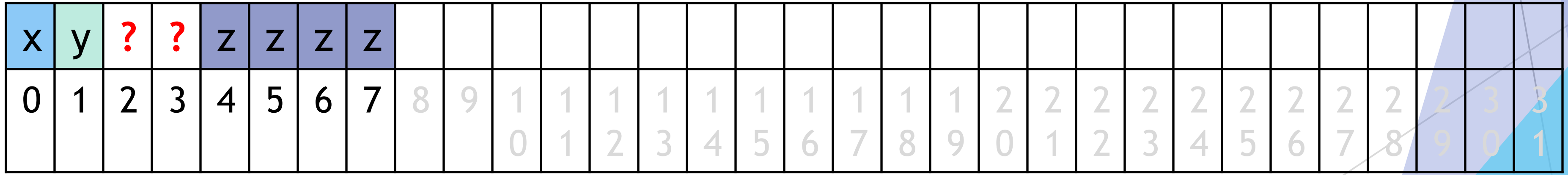
x	x	x	x	y	y	y	y	z	z	z	z																				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

OS Architecture (memory alignament)

```
struct Test
{

        char x;
        char y;
        int  z;
};
```

```
sizeof(Test) = 8
```



OS Architecture (memory alignament)

```
struct Test
{

        char x;
        char y;
        char z;
        int  t;
};
```

```
sizeof(Test) = 8
```

x	y	z	?	t	t	t	t																								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

OS Architecture (memory alignament)

```
struct Test
{

        char x;
        char y;
        char z;
        short s;
        int  t;
};
```

```
sizeof(Test) = 12
```

x	y	z	?	s	s	?	?	t	t	t	t																				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

OS Architecture (memory alignament)

```
struct Test
{

        char x;
        short y;
        char z;
        short s;
        int  t;
};
```

```
sizeof(Test) = 12
```

x	?	y	y	z	?	s	s	t	t	t	t																				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

OS Architecture (memory alignament)

```
struct Test
{

        char x;
        short y;
        double z;
        char s;
        short t;
        int u;
};
```

```
sizeof(Test) = 24
```

x	?	y	y	?	?	?	?	z	z	z	z	z	z	z	z	s	?	t	t	u	u	u	u								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

OS Architecture (memory alignament)

```
struct Test
{

        char x;
        double y;
        int z;
};
```

```
sizeof(Test) = 24
```

x	?	?	?	?	?	?	?	y	y	y	y	y	y	y	y	z	z	z	z	?	?	?	?								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

OS Architecture (memory alignament)

```
struct Test
{

        char x;
        short y;
        int z;
        char t;
};
```

```
sizeof(Test) = 12
```

x	?	y	y	z	z	z	z	t	?	?	?																				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

OS Architecture (memory alignament)

```
#pragma pack(1)
struct Test
{

        char x;
        short y;
        int z;
        char t;
};
```

```
sizeof(Test) = 8
```

x	y	y	z	z	z	z	t																								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

OS Architecture (memory alignament)

```
#pragma pack(2)
struct Test
{

        char x;
        short y;
        int z;
        char t;
};
```

```
sizeof(Test) = 10
```

x	?	y	y	z	z	z	z	t	?																						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

OS Architecture (memory alignament)

```
#pragma pack(1)
_declspec(align(16)) struct Test
{

        char x;
        short y;
        int z;
        char t;
};
```

```
sizeof(Test) = 16
```

x	y	y	z	z	z	z	t	?	?	?	?	?	?	?	?																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

OS Architecture (memory alignament)

```
struct Test
{
        char x;
        short y;
        Test2 z;
        int t;
        char u;
};
```

```
sizeof(Test) = 20
```

```
struct Test2
{
        char x;
        short y;
        int z;
};
```

x	?	y	y	z	z	z	z	z	z	z	z	t	t	t	t	u	?	?	?												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

OS Architecture (memory alignament)

- Reguli pentru cl.exe (setarile default)
  - Fiecare tip este aliniat la o adresa care este divizibila cu dimensinea lui (char din 1 in 1 octeti, short din 2 in 2 octeti, int din 4 in 4 octeti, s.a.m.d).
  - Regula se aplica la tipuri de baza !
  - Se foloseste tot timpul adresa imediat superioara adresei sfarsitului elementului precedent din structura.

```
ALIGN(pozitie,tip) ← (((pozitie - 1)/sizeof(tip))+1)*sizeof(tip)
```

  - Dimensiunea structurii este aliniata si ea la dimensiunea tipului de baza cel mai mare.
  - Directivele pragma **pack** si **_declspec(align)** sunt specifice compilatorului VS (Windows).

C++ history and revisions

Year	
1979	Bjarne Stroustrup starts to work at a super class of the **C** language. The initial name was C with Classes
1983	The name is changed to C++
1990	Borland Turbo C++ is released
1998	First C++ standards (ISO/IEC 14882:1998) → C++98
2003	Second review → C++03
2005	Third review → C++0x
2011	Fourth review → C++11
2014	Fifth review → C++14
2017	The sixth review is expected → C++17

C++98

Keywords	asm do if return typedef auto double inline short typeid bool dynamic_cast int signed typename break else long sizeof union case enum mutable static unsigned catch explicit namespace static_cast using char export new struct virtual class extern operator switch void const false private template volatile const_cast float protected this wchar_t continue for public throw while default friend register true delete goto reinterpret_cast try
Operators	{ } [ ] # ## ( ) <: :> <% %> %: %:%: ; : ... new delete ? :: . .* + * / % ^ & | ~ ! = < > += = *= /= %= ^= &= |= << >> >>= <<= == != <= >= && || ++ , >* >

C++ compilers

- There are many compilers that exists today for C++ language. However, the most popular one are the following:

Compiler	Producer	Latest Version	Compatibility
Visual C++	Microsoft	2013	C++11 (partial)
GCC/G++	GNU Compiler	4.9	C++14 (partial)
clang		3.7	C++11