OOP

Gavrilut Dragos

Course 1

Summary

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

Administrative

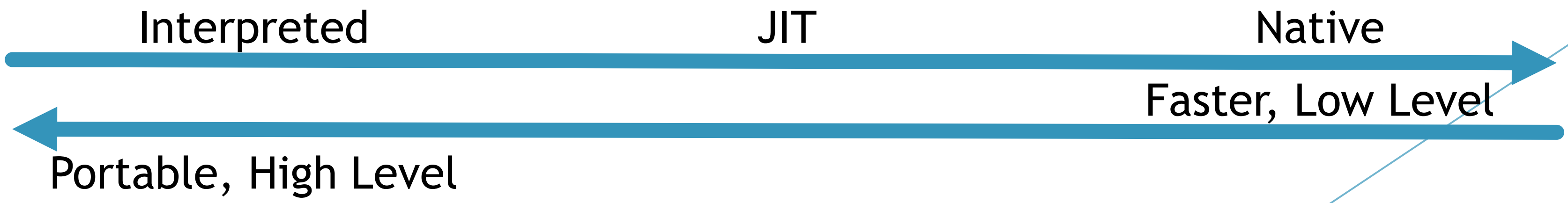
- Site: https://sites.google.com/site/fiicoursepoo/
- Final grade for the OOP exam:
  - First lab examination (week 8) → 35 points
  - Second lab examination (week 14 or 25) → 25 points
  - Course examination → 28 points
  - Lab attendance → 1 point for each lab (12 points maximum)
- Minimum requirements to pass OOP exam:
  - 20 points from first and second lab examination
  - 8 points from the course examination
  - 10 points from lab attendance

Glossary

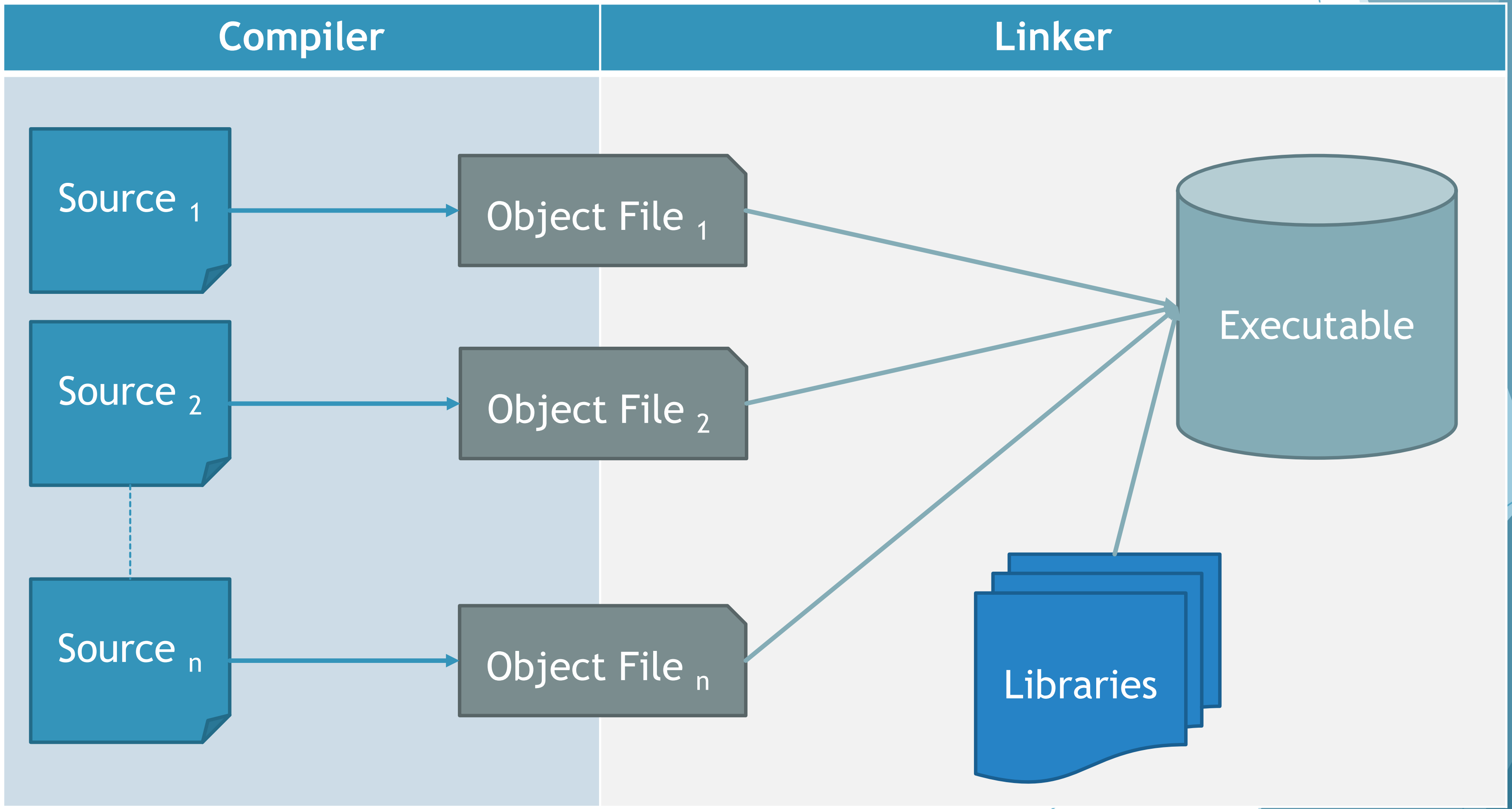
- API → Application Program Interface
- Library – a set o functions that can be use by multiple programs at the same time (for example math functions like cos, sin, tan, etc)
- GUI → Graphic User Interface

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

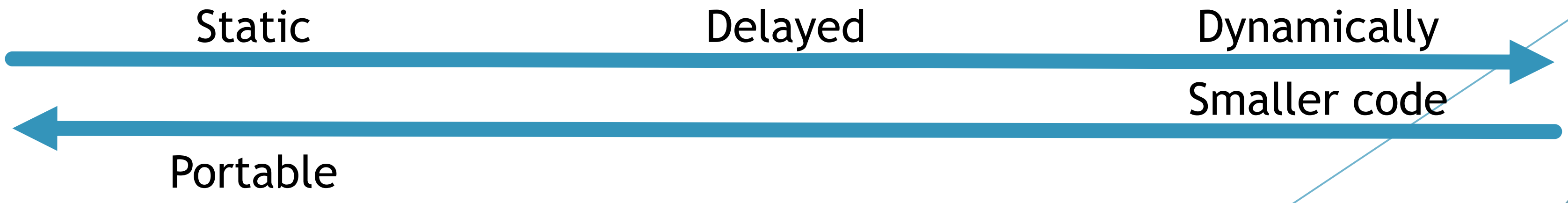


Glossary


Compiler
Linker
Source 1
Source 2
Source n
Object File 1
Object File 2
Object File n
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).



Static Delayed Dynamically

Smaller code

Portable

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:

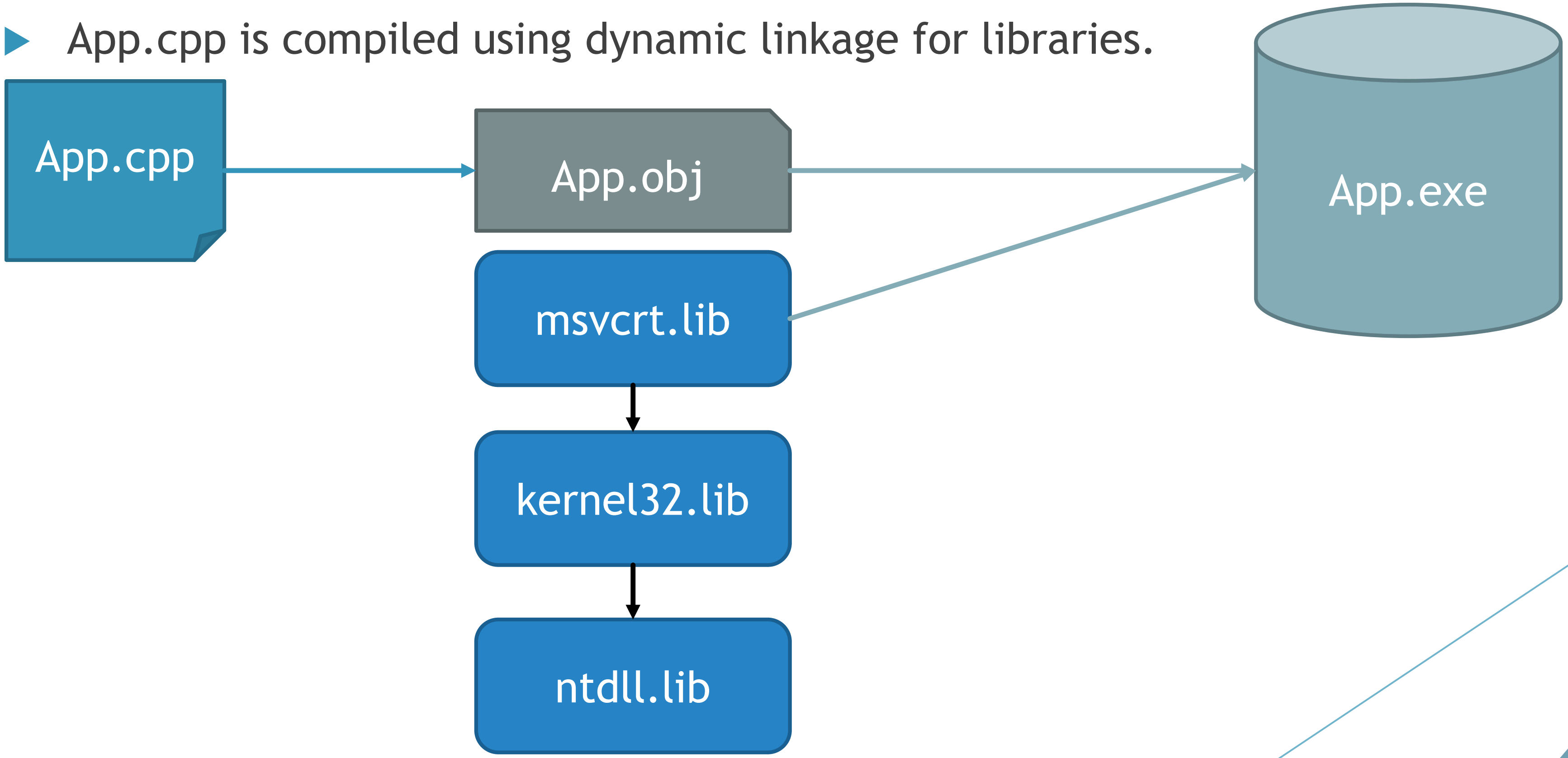
App.cpp

```
#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 !");
}
```

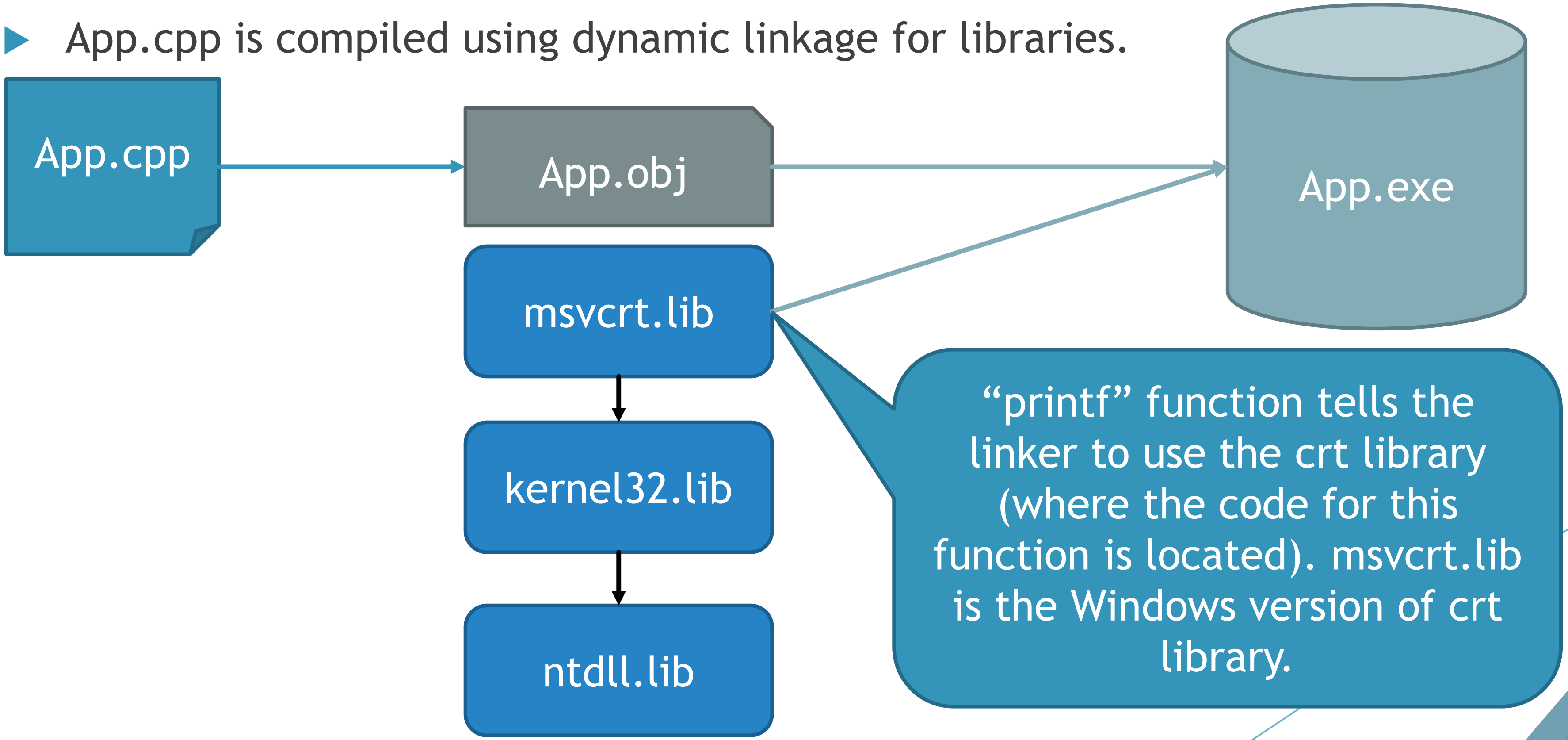
OS Architecture

- 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

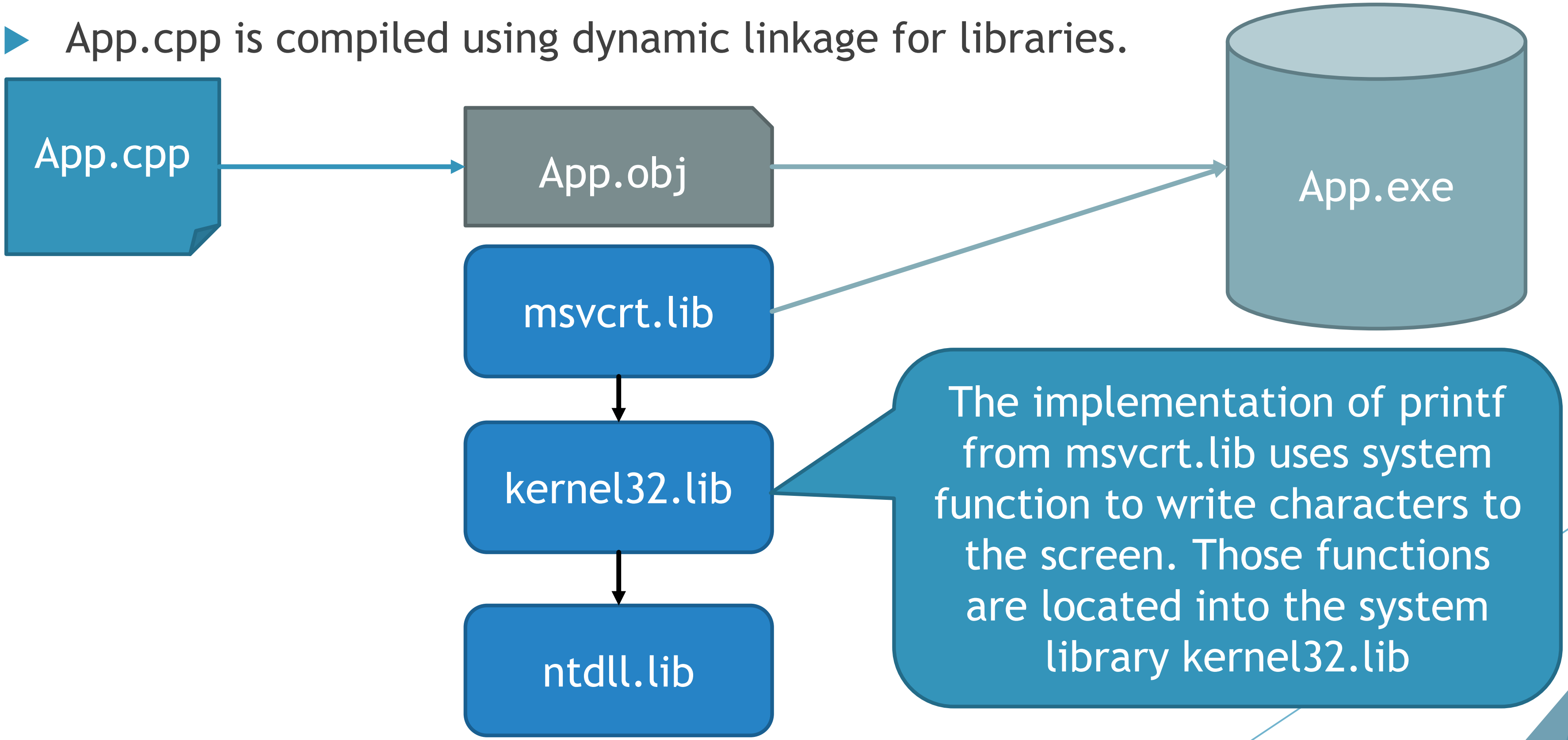
- Let’s assume that we compile “App.cpp” on a Windows system using Microsoft C++ compiler (cl.exe).
- App.cpp is compiled using dynamic linkage for libraries.



“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

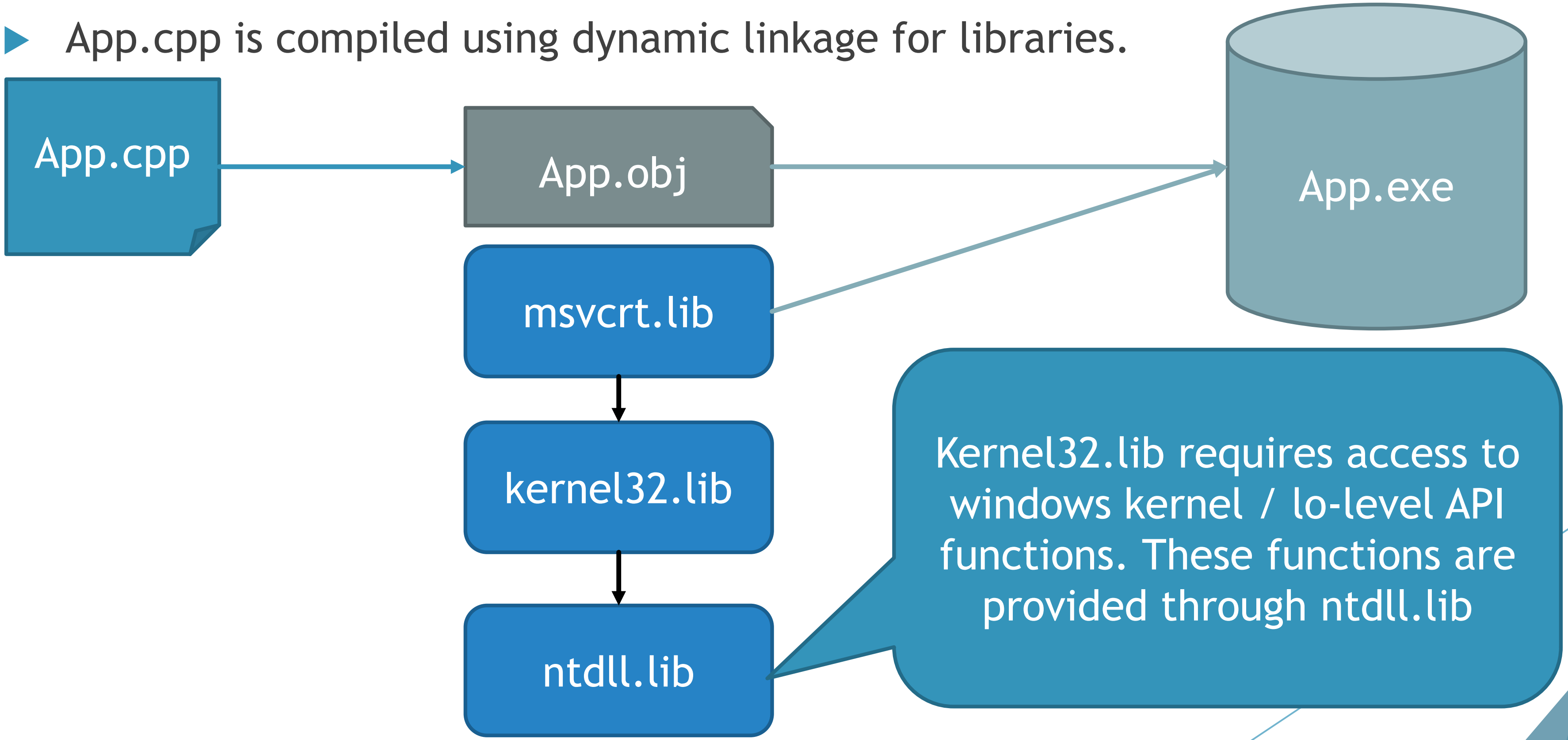
- Let's assume that we compile "App.cpp" on a Windows system using Microsoft C++ compiler (cl.exe).
- App.cpp is compiled using dynamic linkage for libraries.



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.



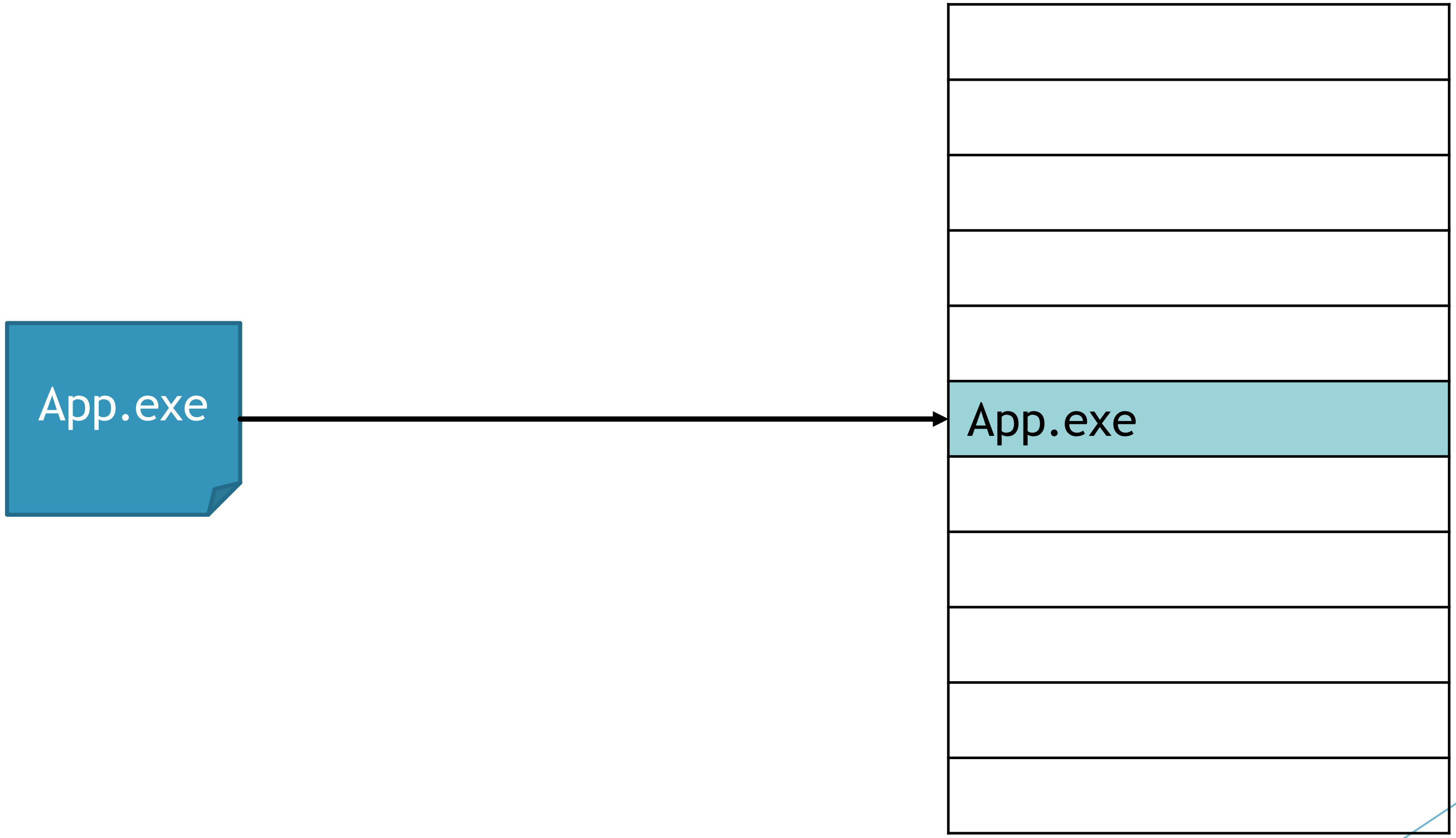
Kernel32.lib requires access to windows kernel / lo-level API functions. These functions are provided through ntdll.lib

OS Architecture

- What happens when a.exe is executed:

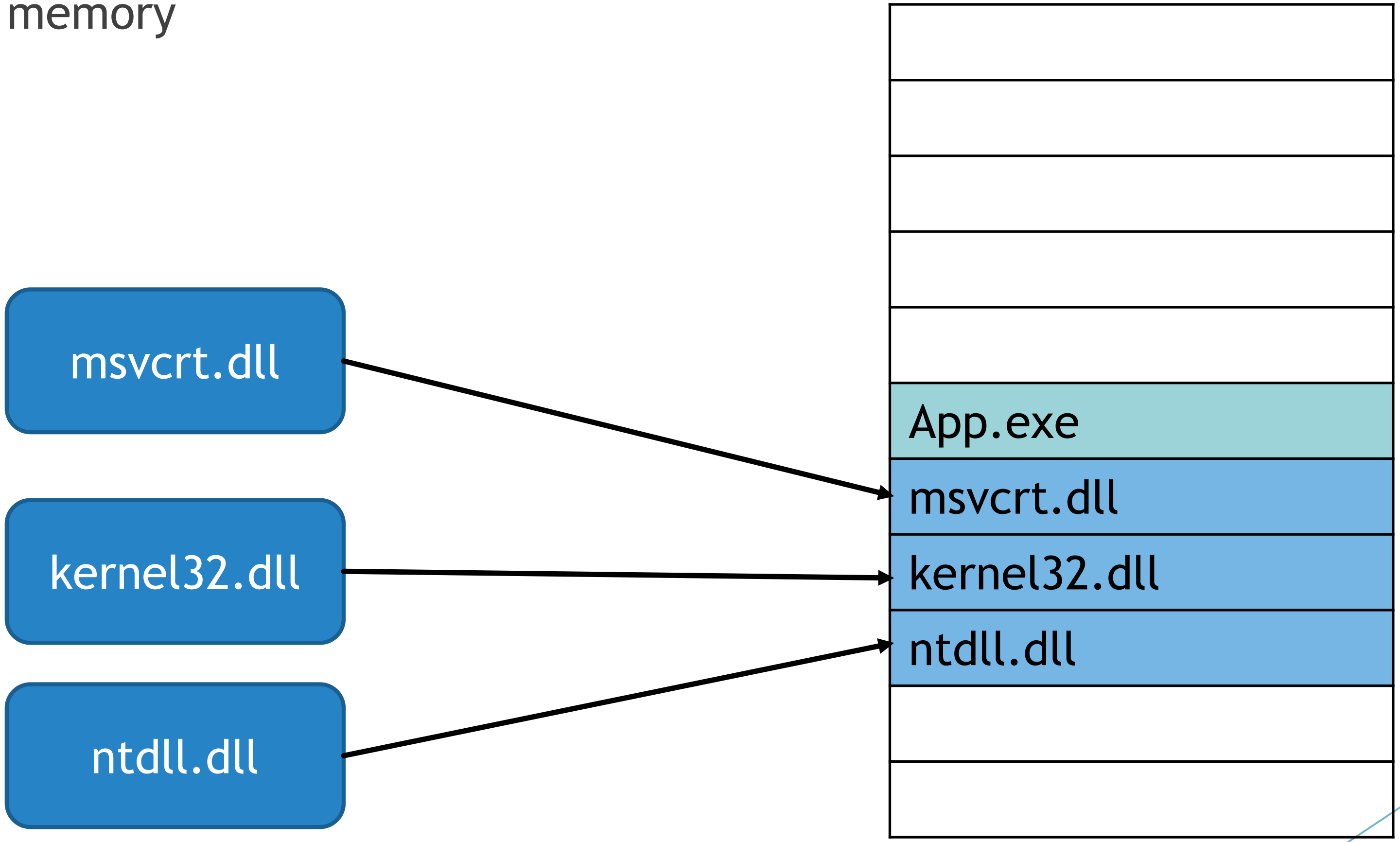
OS Architecture

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



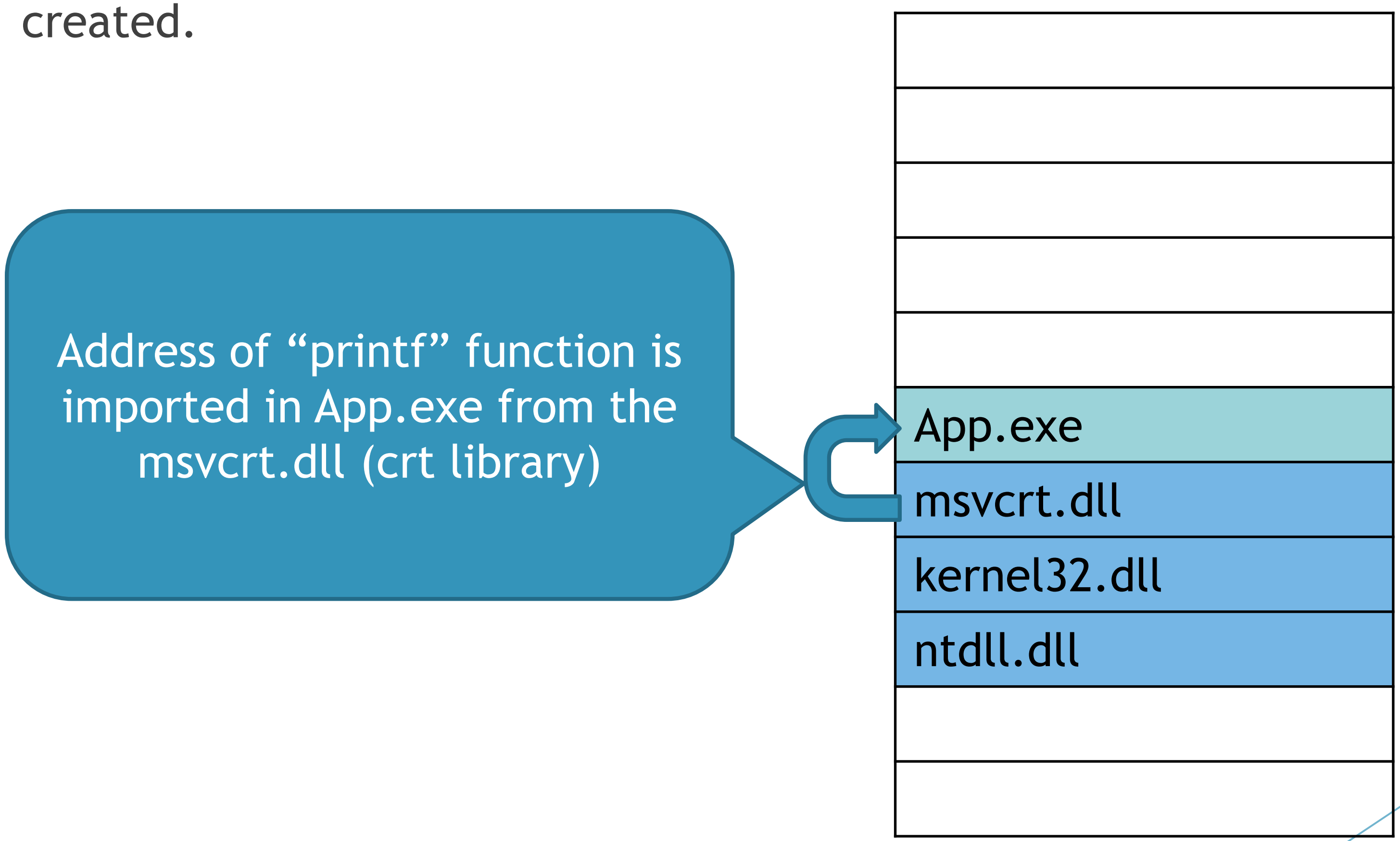
OS Architecture

- 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.

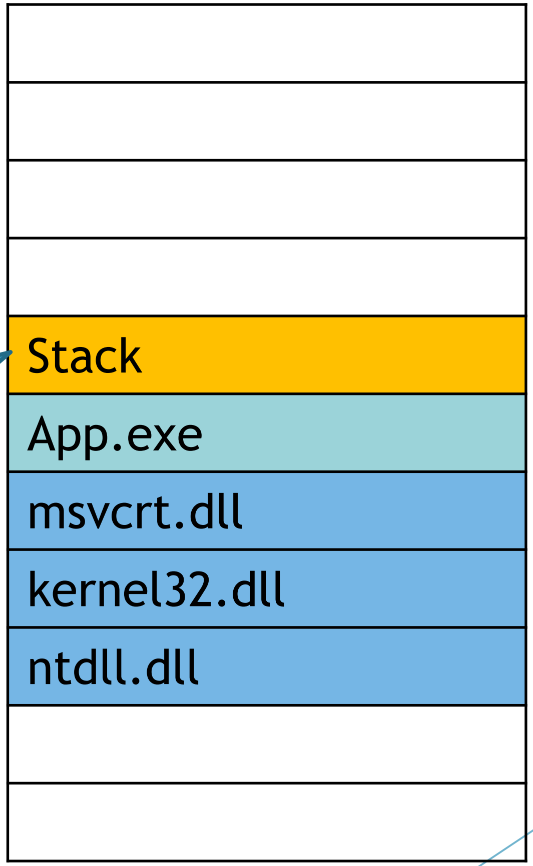


Address of “printf” function is imported in App.exe from the msvcrt.dll (crt library)
App.exe
msvcrt.dll
kernel32.dll
ntdll.dll

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 **<u>undefined</u>** 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

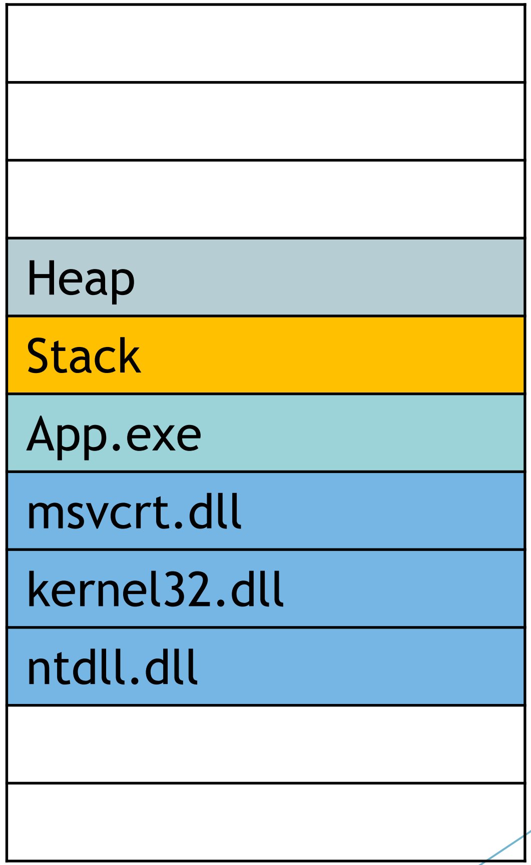
msvcrt.dll

kernel32.dll

ntdll.dll

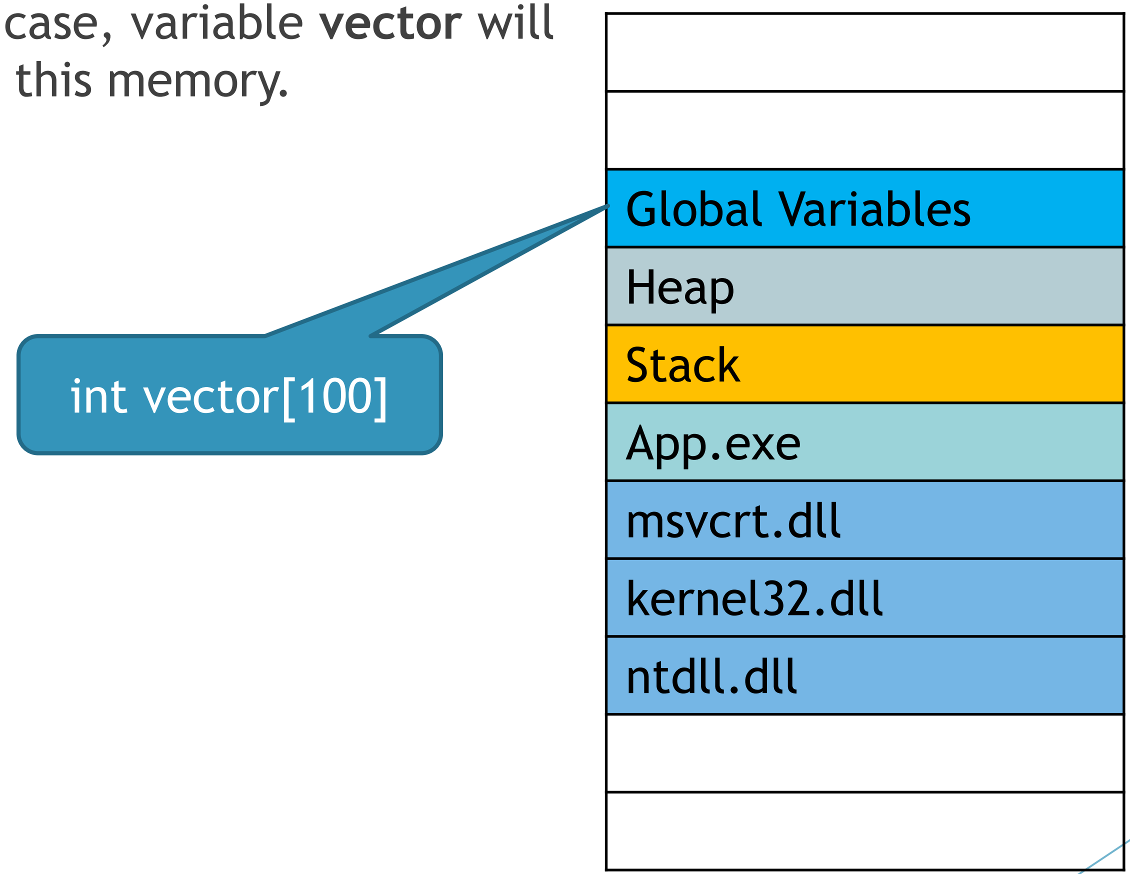
OS Architecture

- 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.**



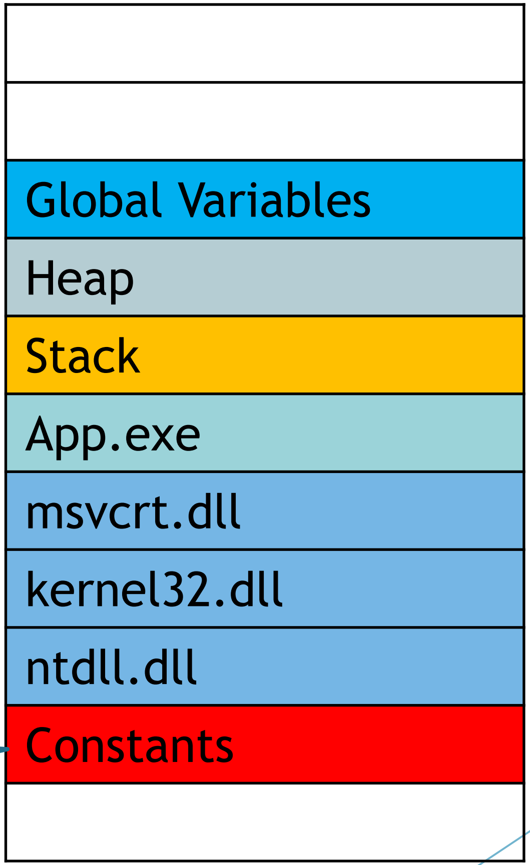
OS Architecture

- 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

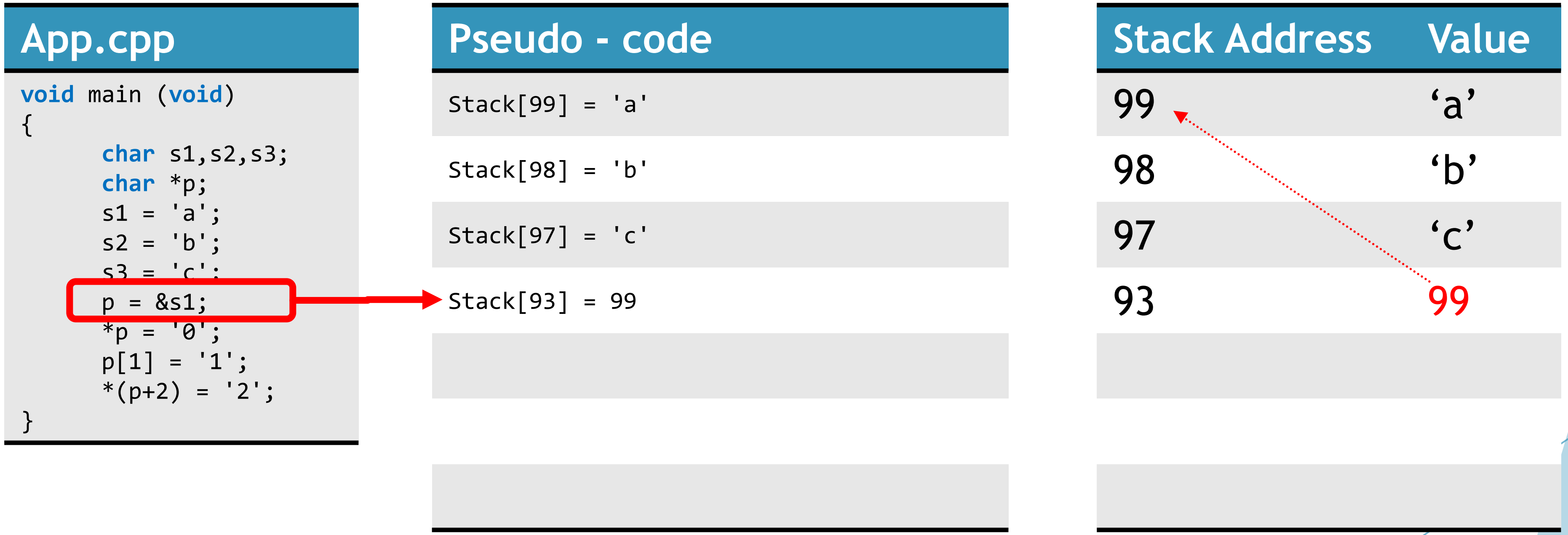
```
void main (void)
{
        char s1,s2,s3;
        char *p;
        s1 = 'a';
        s2 = 'b';
        s3 = 'c';
        p = &s1;
        *p = '0';
        p[1] = '1';
        *(p+2) = '2';
}
```

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 alignment)

```
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 alignment)

```
struct Test
{

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

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

x	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 alignment)

```
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 alignment)

```
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 alignment)

```
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 alignment)

```
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 alignment)

```
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 alignment)

```
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 alignment)

```
#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 alignment)

```
#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 alignment)

```
#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 alignment)

```
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 alignment)

- Alignment rules for cl.exe (default settings)
  - Every type is aligned at the first offset that is a multiple of its size.
  - Rule only applies for basic types
  - To compute an offset for a type one can use the following formula:

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

  - The size of thee structure is a multiple of the biggest basic type size
  - Directive: pragma **pack** and _declspec(align) are specific to Windows C++ compiler (cl.exe)

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	2017	C++17
GCC/G++	GNU Compiler	7.3	C++17
Clang (LLVM)		5.0.1	C++2a (experimental)