OOP

Gavrilut Dragos

Course 1

Summary

- Administrative
- Glossary
- Compilers
- OS architecture
- C++ history and revisions
- From C to C++
- Classes
- Classes – Data Members
- Classes - Methods

- Administrative

Administrative

- Site: https://sites.google.com/view/fii-poo/
- Final grade for the OOP exam:
  - First lab examination (week 8) → 30 points
  - Second lab examination (week 14 or 15) → 30 points
  - Course examination → 30 points
  - Lab activity → up to 1 point for each laboratory and no more than 10 points in total
- Minimum requirements to pass OOP exam:

1. capability to write C++ programs based on specifications
2. capability to correctly apply OOP principles (inheritance, polymorphism, etc)
3. ability to understand OO principles/programming-techniques written in C++
4. ability to detect simple errors in a C++ program and understand them

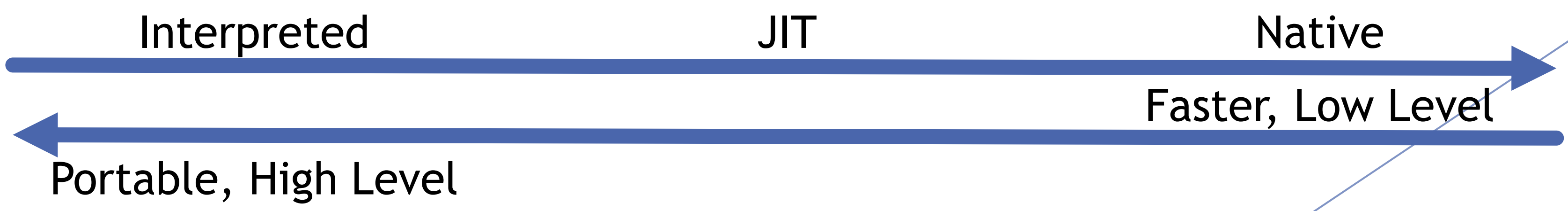
- Glossary

Glossary

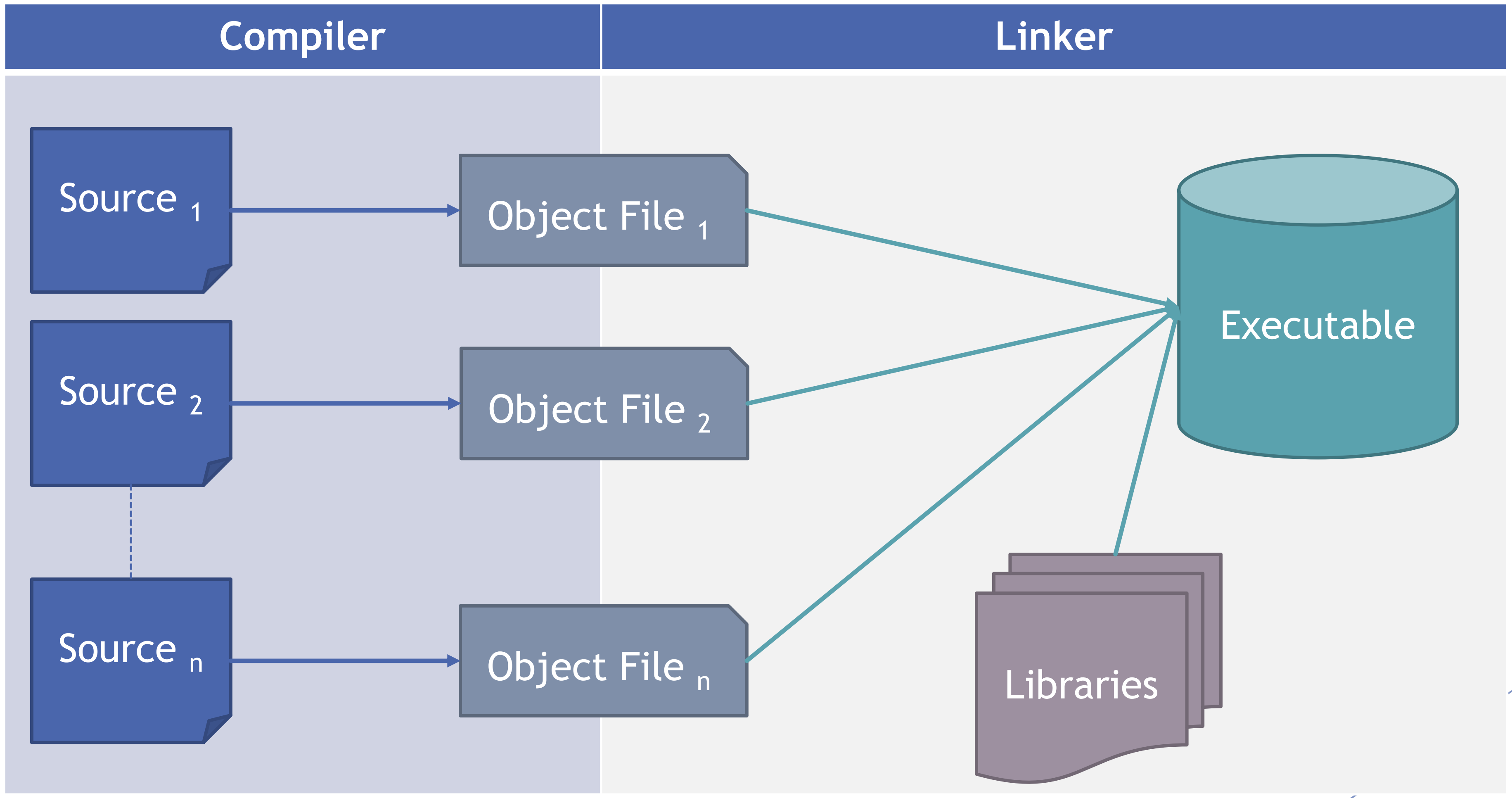
- API → **A**pplication **P**rogram **I**nterface
- Library – a set o functions that can be use by multiple programs at the same time (for example math functions like cos, sin, tan, etc)
- GUI → **G**raphic **U**ser **I**nterface

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. Its 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

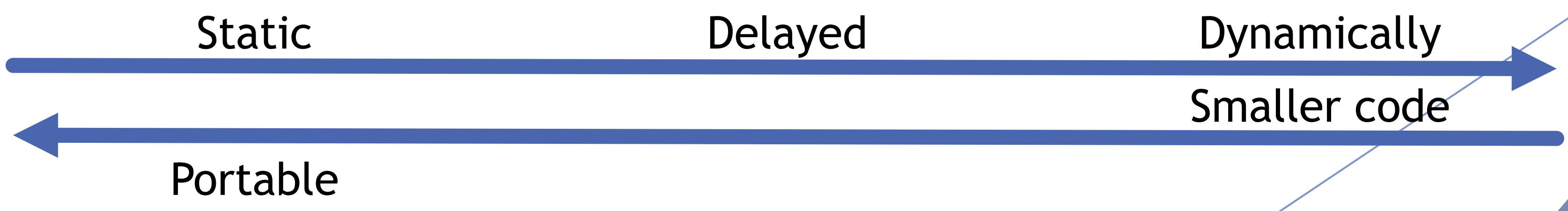


Glossary


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).



- OS Architecture

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:

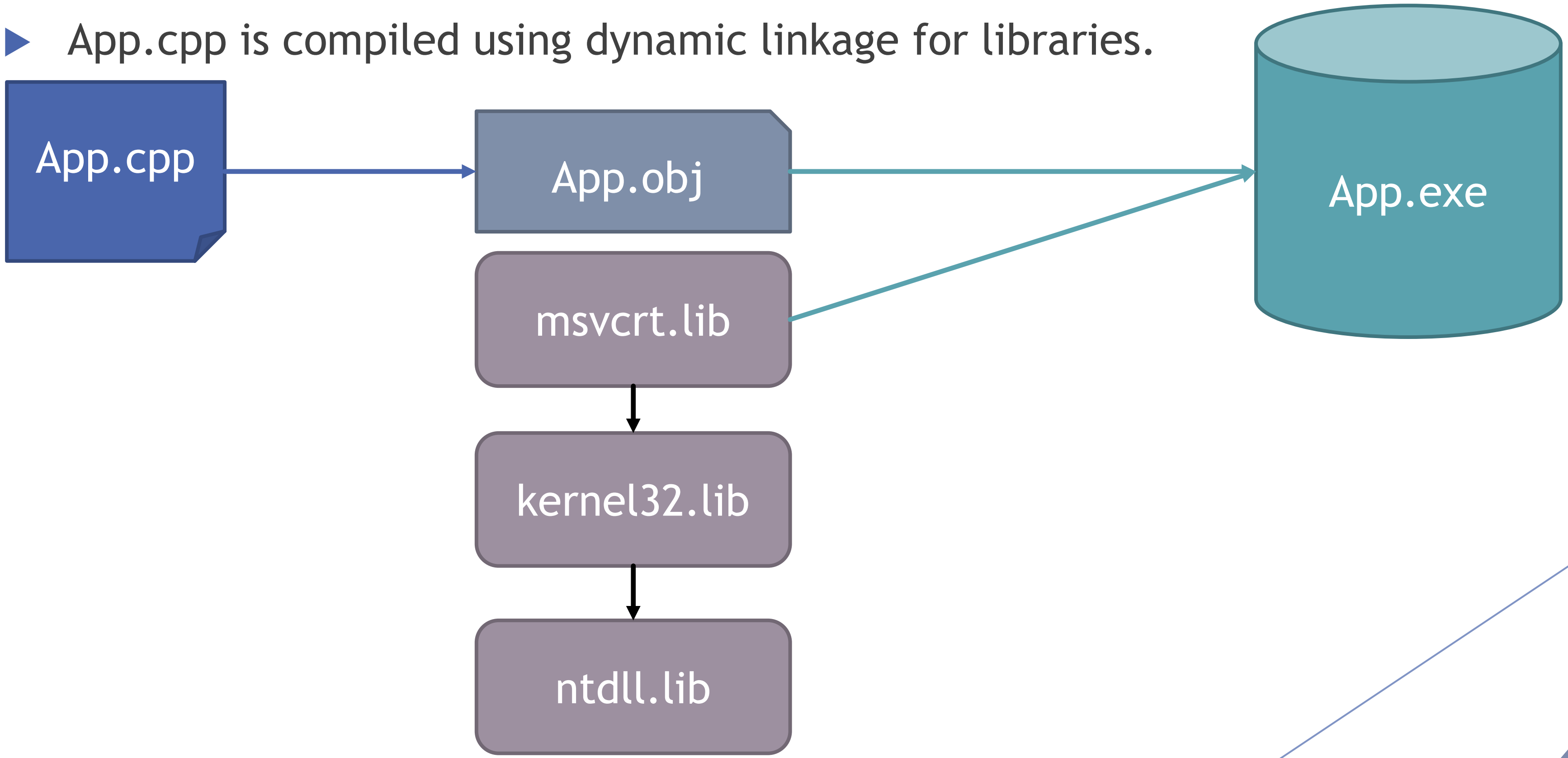
App.cpp

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

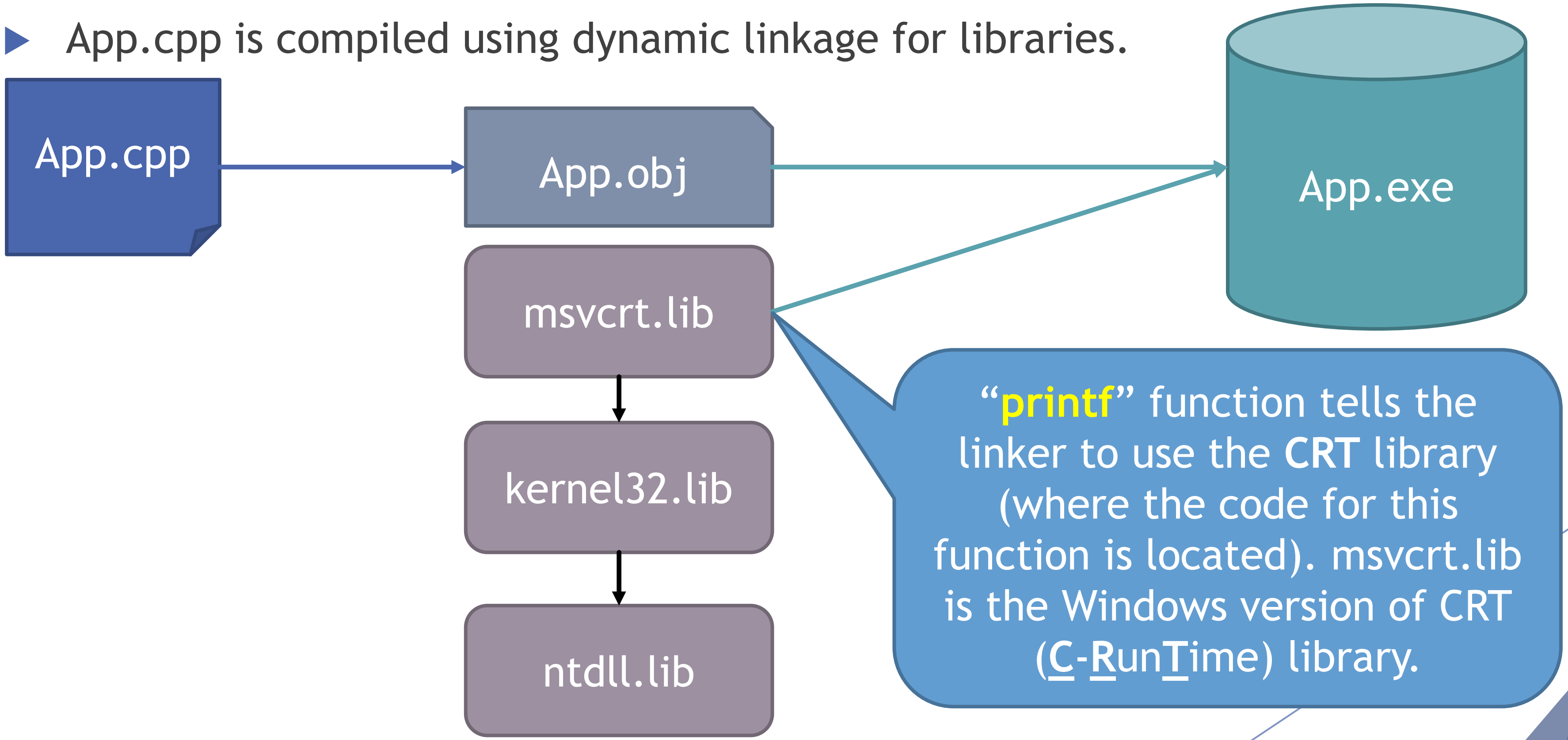
OS Architecture

- 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

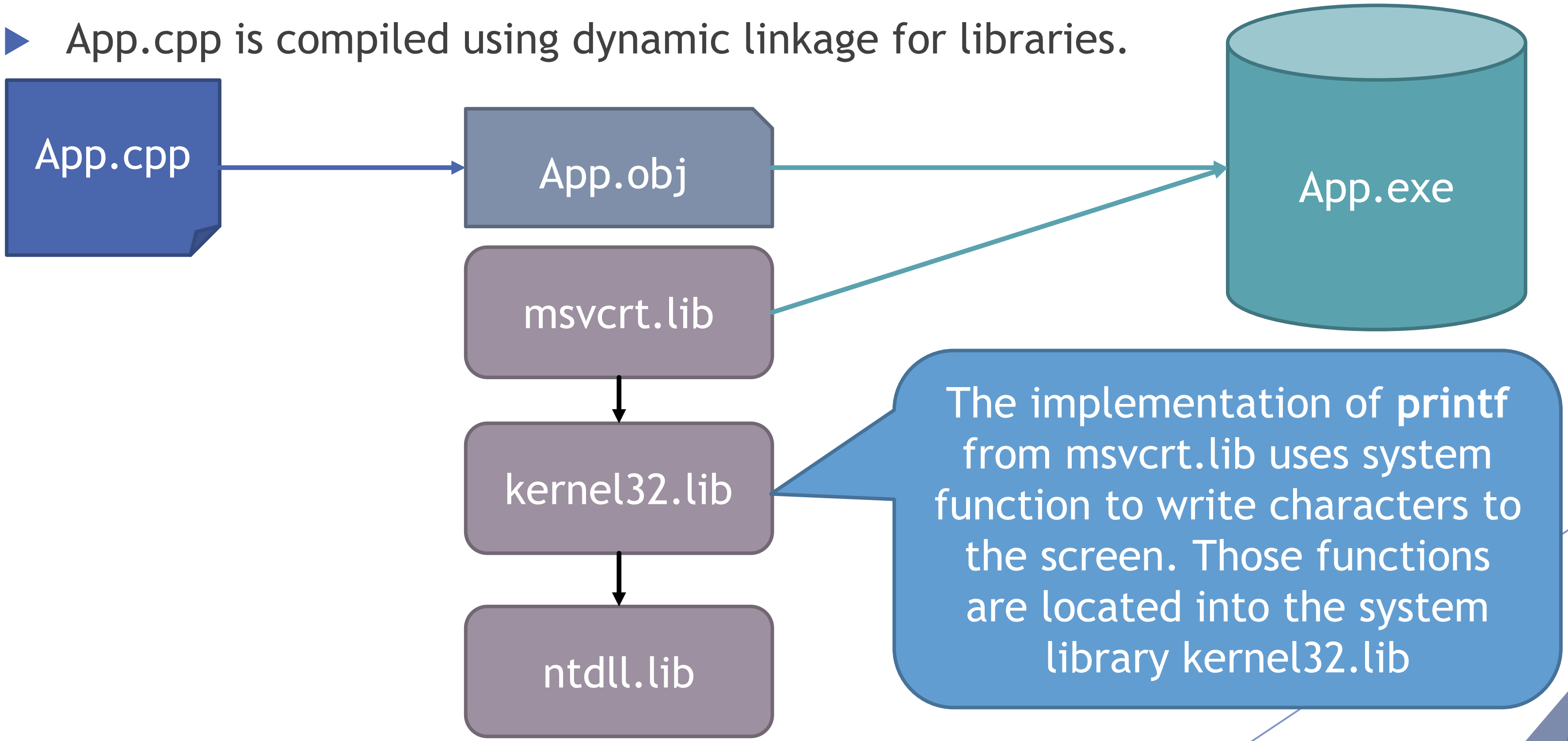
- Let’s assume that we compile “App.cpp” on a Windows system using Microsoft C++ compiler (cl.exe).
- App.cpp is compiled using dynamic linkage for libraries.



“**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 (C-RunTime) library.

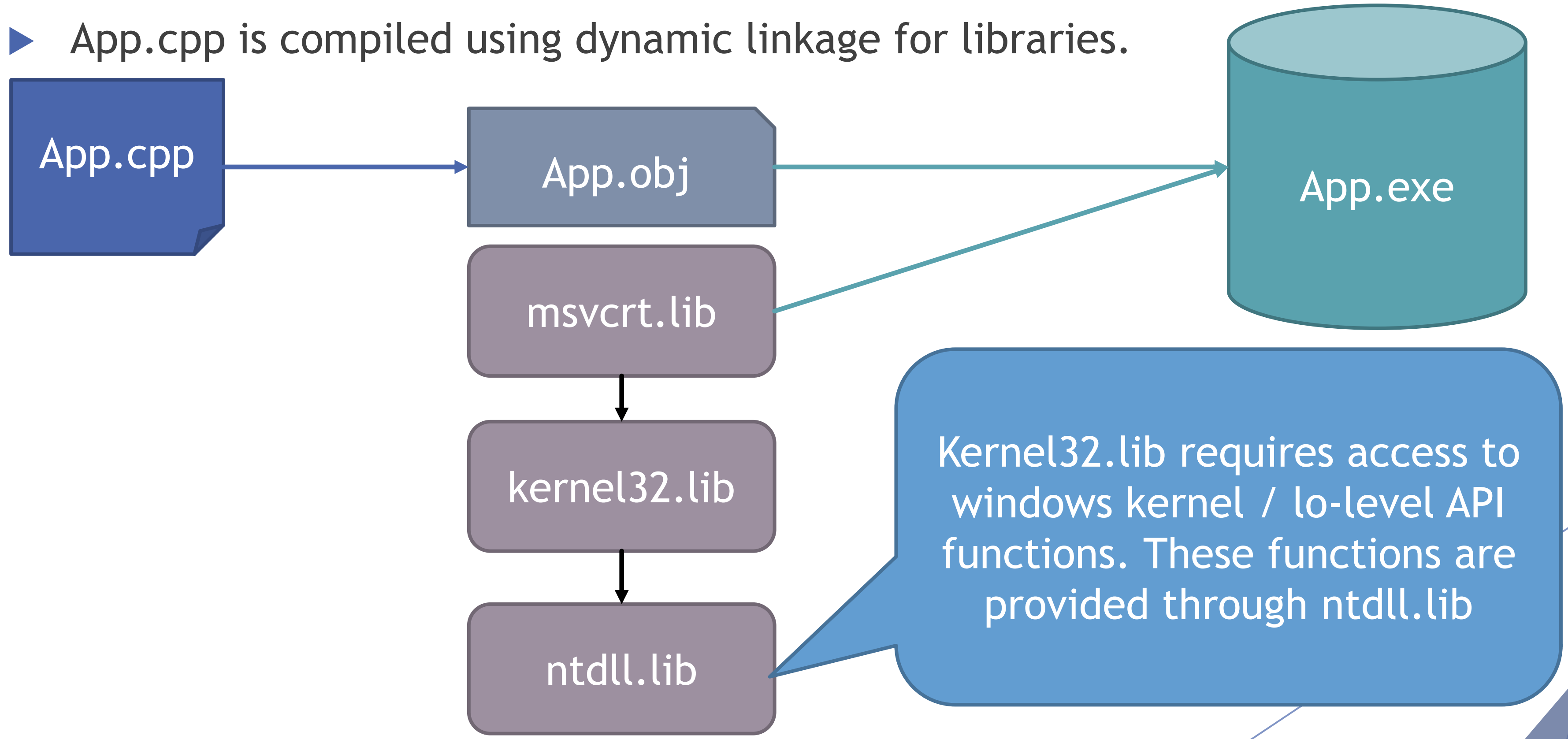
OS Architecture

- 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.


App.cpp
App.obj
App.exe
msvcrt.lib
kernel32.lib
ntdll.lib
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.

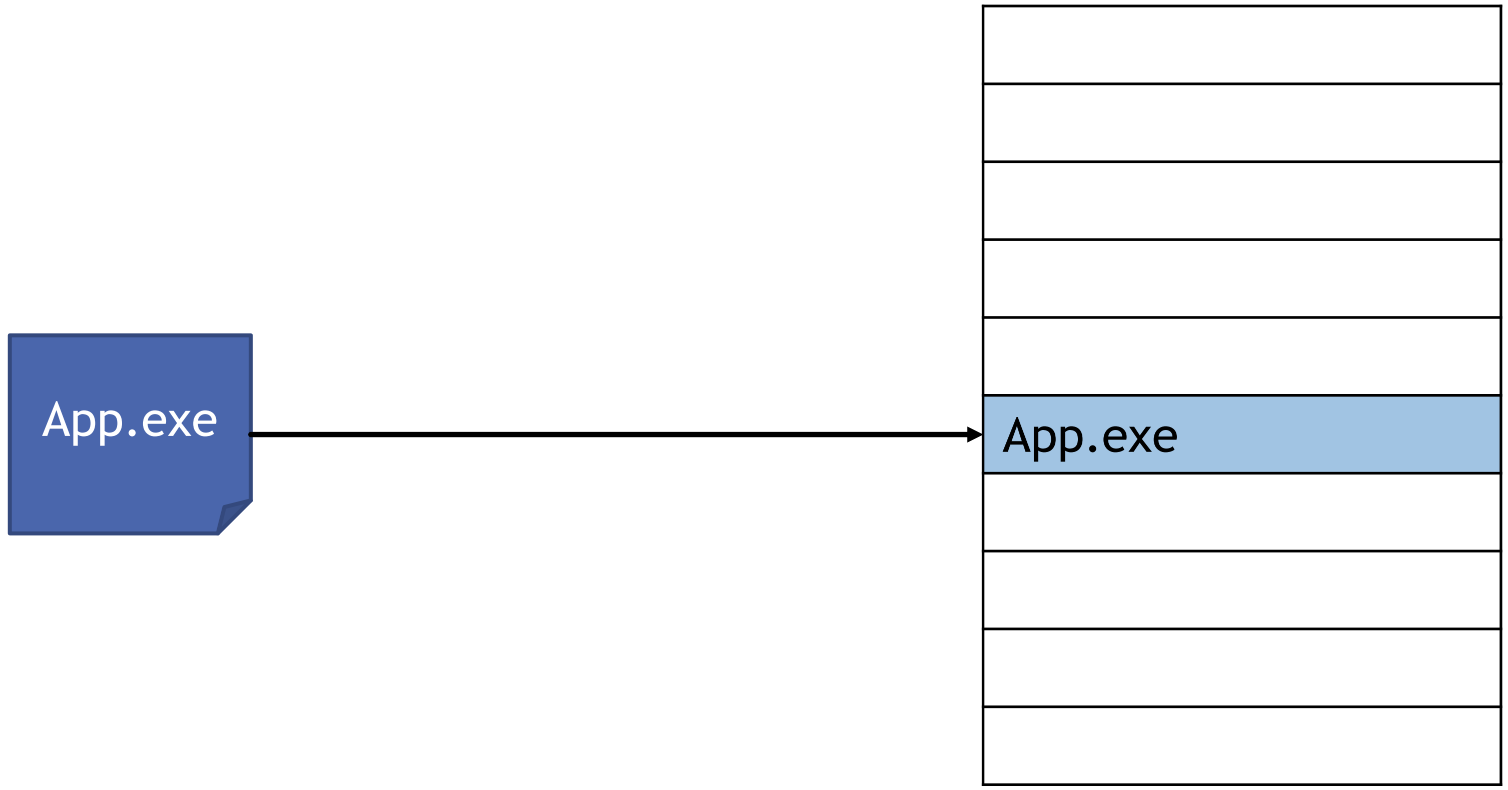

App.cpp
App.obj
App.exe
msvcrt.lib
kernel32.lib
ntdll.lib
Kernel32.lib requires access to windows kernel / lo-level API functions. These functions are provided through ntdll.lib

OS Architecture

- What happens when a.exe is executed:

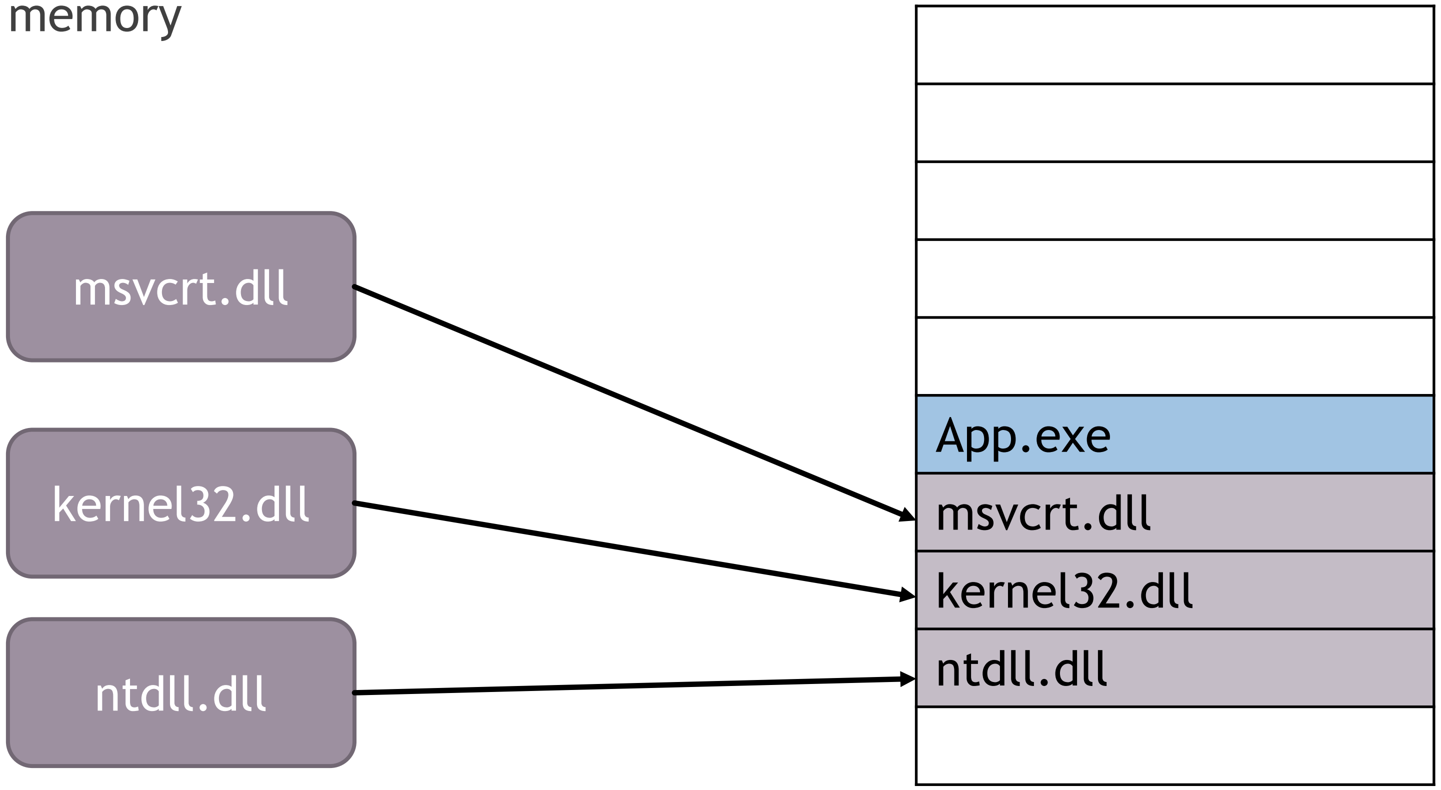
OS Architecture

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



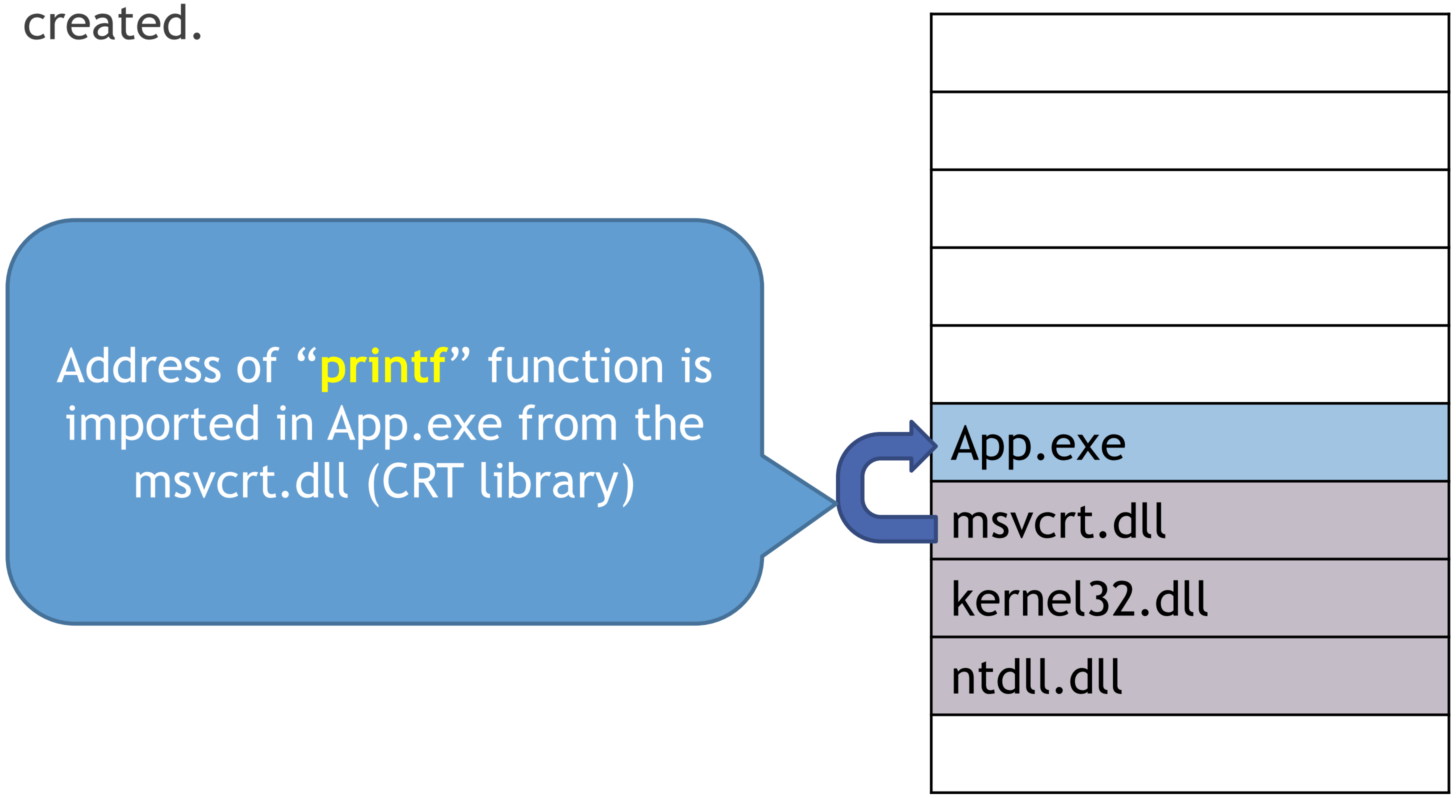
OS Architecture

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



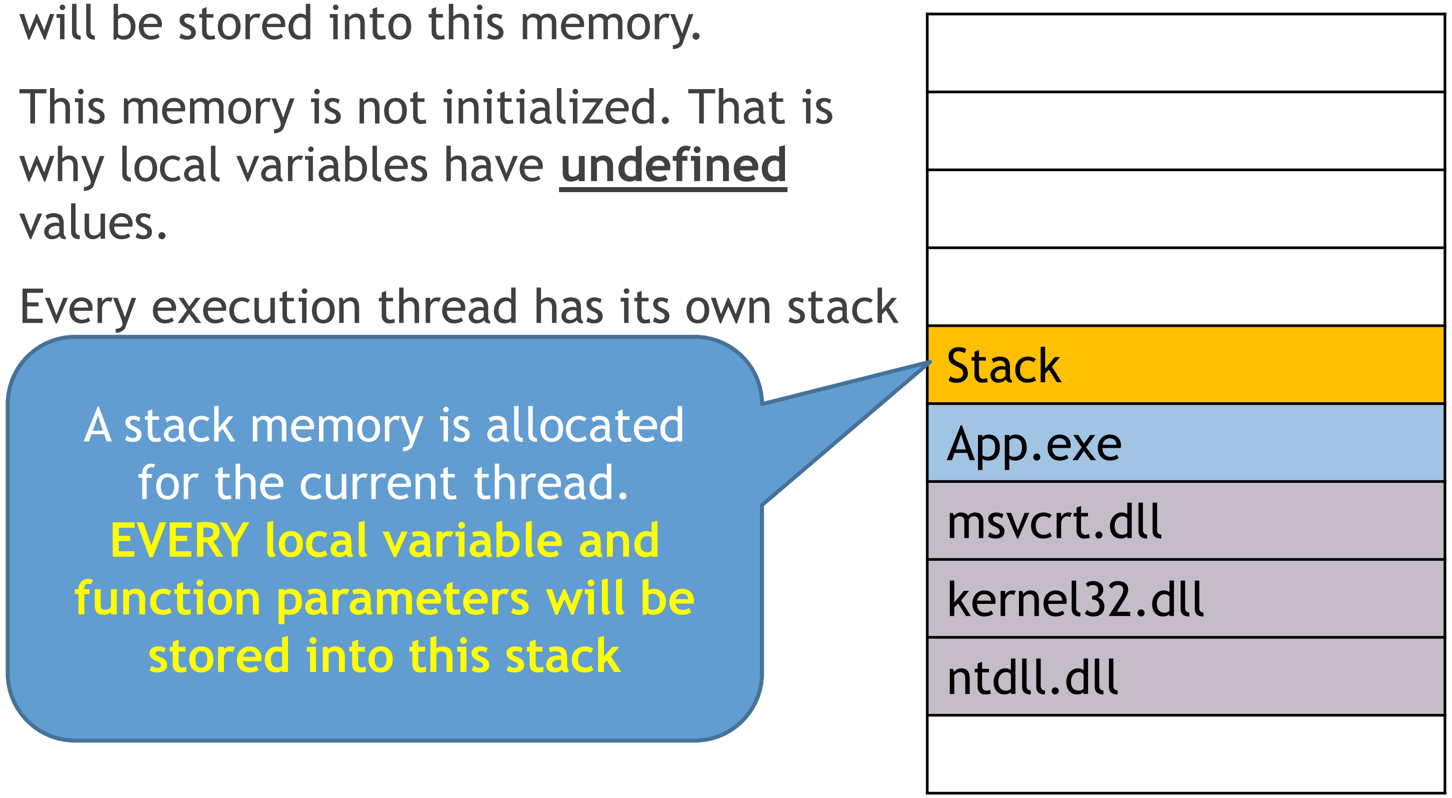
OS Architecture

- 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

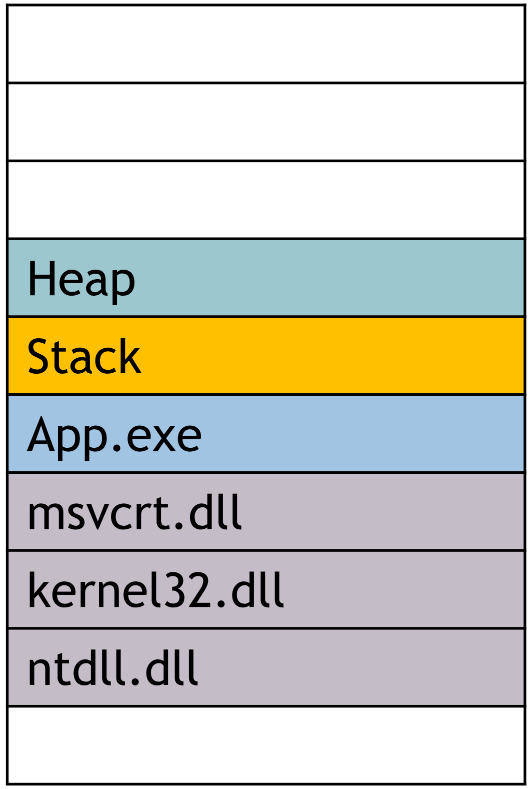
OS Architecture

- 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** values.
- Every execution thread has its own stack


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

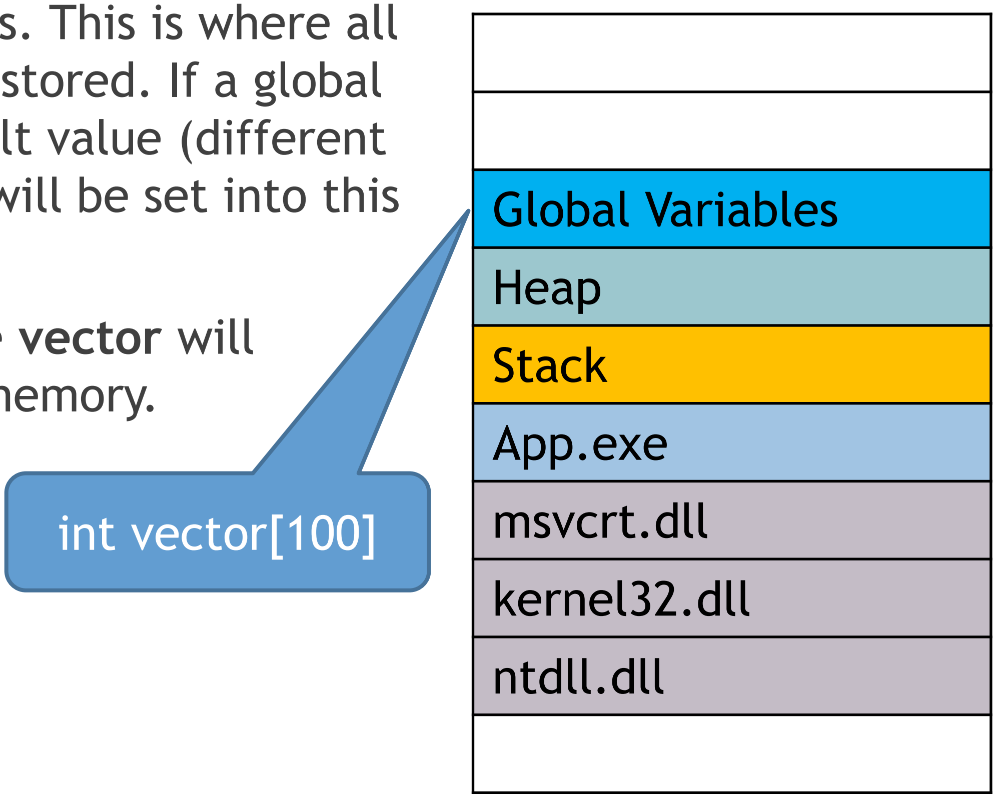
OS Architecture

- Heap memory is allocated. Heap memory is a 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.**
- **The same heap can be used by multiple threats**



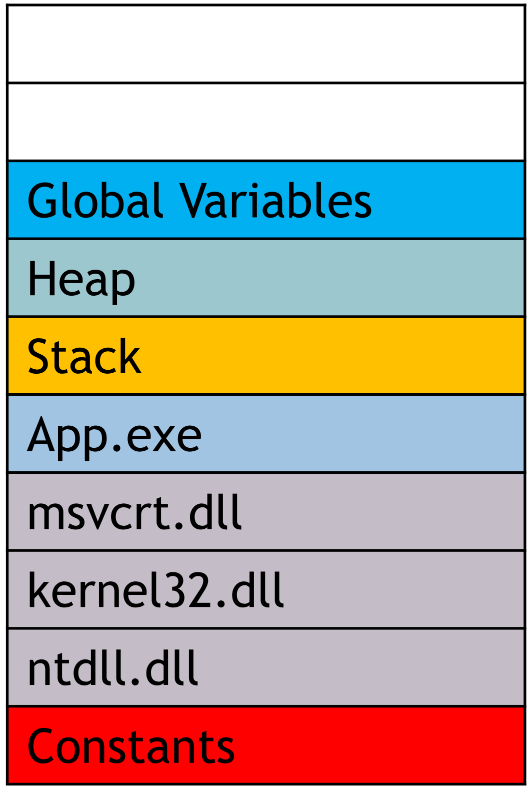
OS Architecture

- A memory for global variable is allocated. This memory is initialized by default with 0 values. This is where all global variables are stored. If a global variable has a default value (different than 0), that value will be set into this memory space.
- In our case, variable **vector** will be stored into this memory.



OS Architecture

- A memory space for constant data is created. This memory stores 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 stores such a variable will produce an exception and a system crash.
- In our example, the string “`Found 100 odd numbers !`” will be stored 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;
};
```

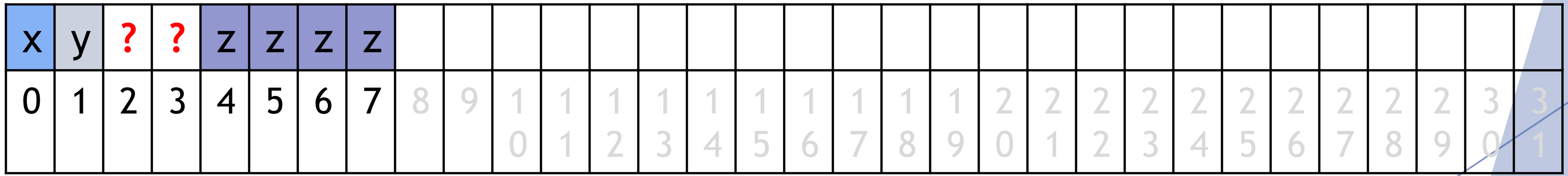
```
sizeof(Test) = 12
```

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
```



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
```

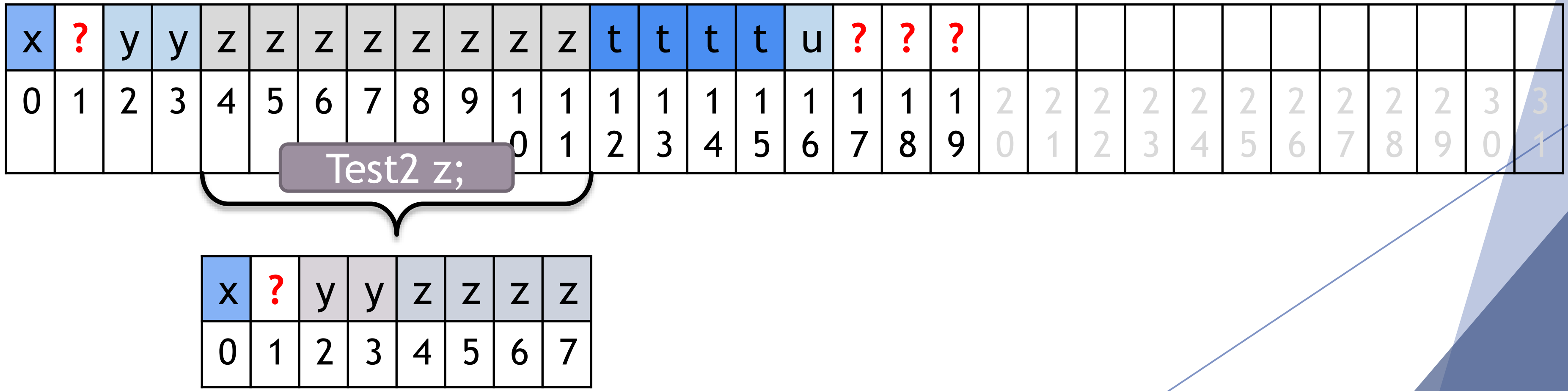
x	y	y	z	z	z	z	t	?	?	?	?	?	?	?	?																
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

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

x	?	y	y	z	z	z	z
0	1	2	3	4	5	6	7

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, the following formula can be used:

```
ALIGN(position,type) ← (((position - 1)/sizeof(type))+1)*sizeof(type)
```

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

C++ history and revisions

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
2020	Seventh review → C++20
2023	Eight review → C++23 or C++2b

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	2021 (14.29.30133)	C++20 (C++23 partial)
GCC/G++	GNU Compiler	11.2	C++20 (C++23 partial)
Clang (LLVM)		13.0.1	C++20 (C++23 partial)

- From C to C++

C to C++

- Let’s look at the following C code.

App.cpp

```
struct Person
{
      int Age;
      int Height;
}
void main()
{
      Person p;
      printf(“Age = %d”,p.Age);
      p.Age = -5;
      p.Height = 100000;
}
```

- What can we observe that does not have any sense ?

C to C++

- Let’s look at the following C code.

App.cpp

```
struct Person
{
	int Age;
	int Height;
}
void main()
{
	Person p;
	printf(“Age = %d”,p.Age);
	p.Age = -5;
	p.Height = 100000;
}
```

- The program is correct, however having these values for the field Age and Height does not make any sense.
- There is no form of initialization for the variable p. This means that the value for the Age field that printf function will show is undefined.

C to C++

- The solution is to create some functions to initialize and validate structure Person.

App.c

```
struct Person
{
	int Age;
	int Height;
}
void main()
{
	Person p;
	printf("Age = %d",p.Age);
	p.Age = -5;
	p.Height = 100000;
}
```

App.c

```
struct Person
{
	int Age;
	int Height;
}
void Init(Person *p)
{
	p->Age= 10;
	p->Height= 100;
}
void SetAge(Person *p,int value)
{
	if ((value>0) && (value<200))
		p->Age= value;
}
void SetHeight(Person *p,int value)
{
	if ((value>50) && (value<300))
		p->Height= value;
}
void main()
{
	Person p;
	Init(&p);
	SetAge(&p, -5);
	SetHeight(&p, 100000);
}
```

C to C++

- This approach while it provides certain advantages also comes with some drawbacks:

App.c

```
struct Person
{
      int Age;
      int Height;
}
void Init(Person *p)
{
      p->Age= 10;
      p->Height= 100;
}
void SetAge(Person *p,int value)
{
      if ((value>0) && (value<200))
            p->Age= value;
}
void SetHeight(Person *p,int value)
{
      if ((value>50) && (value<300))
            p->Height= value;
}
void main()
{
      Person p;
      Init(&p);
      SetAge(&p, -5);
      SetHeight(&p, 100000);
}
```

C to C++

- This approach while it provides certain advantages also comes with some drawbacks:

App.c

```
struct Person
{
	int Age;
	int Height;
}
void Init(Person *p) { … }

void SetAge(Person *p,int value)
{
	if ((value>0) && (value<200) && (p!=NULL))
		p->Age= value;
}
void SetHeight(Person *p,int value) { … }

void main()
{
	Person p;
	Init(&p);
	SetAge(&p, -5);
	SetHeight(&p, 100000);
}
```

a) Pointer “p” from functions SetAge and SetHeight must be validated

C to C++

- This approach while it provides certain advantages also comes with some drawbacks:

App.c

```
struct Person
{
      int Age;
      int Height;
}
void Init(Person *p) { … }

void SetAge(Person *p,int value)
{
      if ((value>0) && (value<200) && (p!=NULL)))
            p->Age= value;
}
void SetHeight(Person *p,int value) { … }

void main()
{
      Person p;
      Init(&p);
      SetAge(&p, -5);
      SetHeight(&p, 100000);
      p.Age = -1;
      p.Height = -2;
}
```

a) Pointer "p" from functions SetAge and SetHeight must be validated

b) We can still change the values for fields Age and Heights and the program will compile and execute.

C to C++

- This approach while it provides certain advantages also comes with some drawbacks:

App.c

```
struct Person
{
      int Age;
      int Height;
}
void Init(Person *p) { … }

void SetAge(Person *p,int value) { … }

void SetHeight(Person *p,int value) { … }

void main()
{
      Person p;
      Init(&p);
      printf("Age = %d",p.Age);
}
```

a) Pointer "p" from functions SetAge and SetHeight must be validated

b) We can still change the values for fields Age and Heights and the program will compile and execute.

c) Variable p is not initialized by default (we have to call a special function to do this).

C to C++

- This approach while it provides certain advantages also comes with some drawbacks:

App.c

```
struct Person
{
      int Age;
      int Height;
}
void Init(Person *p) {...}

void SetAge(Person *p,int value) {...}

void SetHeight(Person *p,int value) {...}

void AddYear(Person *p,int value) {...}

void AddHeight(Person *p, int value) {...}

int  GetAge(Person *p)  {...}

int  GetHeight(Person *p) {...}
```

a) Pointer “p” from functions SetAge and SetHeight must be validated

b) We can still change the values for fields Age and Heights and the program will compile and execute.

c) Variable p is not initialized by default (we have to call a special function to do this).

d) Having a lot of functions that work with a structure means that each time one of those functions is called we need to be sure that the right pointer is pass to that function.

C to C++

Basically , we need a language that can do the following:

- Restrict access to certain structure fields
- There should be an at least one initialization function that is called whenever an instance of that structure is created.
- We should find a way to not send a pointer to the structure every time we need to call a function that modifies different fields of that structure
- We should not need to validate that pointer (the validation should be done during the compiler phase).

C to C++ (Conversions)

App.c

```
struct Person
{
      int Age;
}
void SetAge(Person *p,int value)
{
      if (p==NULL)
            return;
      if ((value>0) && (value<200))
            p->Age = value;
}
void Init(Person *p)
{
      if (p==NULL)
            return;
      p->Age = 10;
}
void main()
{
      Person p;
      Init(&p);
      SetAge(&p,10);
}
```

App.cpp

C to C++ (Conversions)

App.c

```
struct Person
{
	int Age;
}
void SetAge(Person *p,int value)
{
	if (p==NULL)
		return;
	if ((value>0) && (value<200))
		p->Age = value;
}
void Init(Person *p)
{
	if (p==NULL)
		return;
	p->Age = 10;
}
void main()
{
	Person p;
	Init(&p);
	SetAge(&p,10);
}
```

App.cpp

```
class Person
{
```

C to C++ (Conversions)

App.c

```
struct Person
{
	int Age;
}
void SetAge(Person *p,int value)
{
	if (p==NULL)
		return;
	if ((value>0) && (value<200))
		p->Age = value;
}
void Init(Person *p)
{
	if (p==NULL)
		return;
	p->Age = 10;
}
void main()
{
	Person p;
	Init(&p);
	SetAge(&p,10);
}
```

App.cpp

```
class Person
{
	private:
```

Access modifier (specifies who can access the fields that are declare after it)

C to C++ (Conversions)

App.c

```
struct Person
{
      int Age;
}
void SetAge(Person *p,int value)
{
      if (p==NULL)
            return;
      if ((value>0) && (value<200))
            p->Age = value;
}
void Init(Person *p)
{
      if (p==NULL)
            return;
      p->Age = 10;
}
void main()
{
      Person p;
      Init(&p);
      SetAge(&p,10);
}
```

App.cpp

```
class Person
{
      private:
            int Age;
```

C to C++ (Conversions)

App.c

```
struct Person
{
	int Age;
}
void SetAge(Person *p,int value)
{
	if (p==NULL)
		return;
	if ((value>0) && (value<200))
		p->Age = value;
}
void Init(Person *p)
{
	if (p==NULL)
		return;
	p->Age = 10;
}
void main()
{
	Person p;
	Init(&p);
	SetAge(&p,10);
}
```

App.cpp

```
class Person
{
	private:
		int Age;
	public:
		void SetAge(int value);
```

C to C++ (Conversions)

App.c

```
struct Person
{
	int Age;
}
void SetAge(Person *p,int value)
{
	if (p==NULL)
		return;
	if ((value>0) && (value<200))
		p->Age = value;
}
void Init(Person *p)
{
	if (p==NULL)
		return;
	p->Age = 10;
}
void main()
{
	Person p;
	Init(&p);
	SetAge(&p,10);
}
```

App.cpp

```
class Person
{
	private:
		int Age;
	public:
		void SetAge(int value);
		Person();
}
```

Constructor

C to C++ (Conversions)

App.c

```
struct Person
{
      int Age;
}
void SetAge(Person *p,int value)
{
      if (p==NULL)
            return;
      if ((value>0) && (value<200))
            p->Age = value;
}
void Init(Person *p)
{
      if (p==NULL)
            return;
      p->Age = 10;
}
void main()
{
      Person p;
      Init(&p);
      SetAge(&p,10);
}
```

App.cpp

```
class Person
{
      private:
            int Age;
      public:
            void SetAge(int value);
            Person();
}
void Person::SetAge(int value)
{
      if ((value>0) && (value<200))
            this->Age = value;
}
```

C to C++ (Conversions)

App.c

```
struct Person
{
	int Age;
}
void SetAge(Person *p,int value)
{
	if (p==NULL)
		return;
	if ((value>0) && (value<200))
		p->Age = value;
}
void Init(Person *p)
{
	if (p==NULL)
		return;
	p->Age = 10;
}
void main()
{
	Person p;
	Init(&p);
	SetAge(&p,10);
}
```

App.cpp

```
class Person
{
	private:
		int Age;
	public:
		void SetAge(int value);
		Person();
}
void Person::SetAge(int value)
{
	if ((value>0) && (value<200))
		this->Age = value;
}
```

C to C++ (Conversions)

App.c

```
struct Person
{
	int Age;
}
void SetAge(Person *p,int value)
{
	if (p==NULL)
		return;
	if ((value>0) && (value<200))
		p->Age = value;
}
void Init(Person *p)
{
	if (p==NULL)
		return;
	p->Age = 10;
}
void main()
{
	Person p;
	Init(&p);
	SetAge(&p,10);
}
```

App.cpp

```
class Person
{
	private:
		int Age;
	public:
		void SetAge(int value);
		Person();
}
void Person::SetAge(int value)
{
	if ((value>0) && (value<200))
		this->Age = value;
}
Person::Person()
{
	this->Age = 10;
}
```

C to C++ (Conversions)

App.c

```
struct Person
{
	int Age;
}
void SetAge(Person *p,int value)
{
	if (p==NULL)
		return;
	if ((value>0) && (value<200))
		p->Age = value;
}
void Init(Person *p)
{
	if (p==NULL)
		return;
	p->Age = 10;
}
void main()
{
	Person p;
	Init(&p);
	SetAge(&p,10);
}
```

App.cpp

```
class Person
{
	private:
		int Age;
	public:
		void SetAge(int value);
		Person();
}
void Person::SetAge(int value)
{
	if ((value>0) && (value<200))
		this->Age = value;
}
Person::Person()
{
	this->Age = 10;
}
void main()
{
	Person p;
```

The constructor is called by default whenever an object of type Person is created.

C to C++ (Conversions)

App.c

```
struct Person
{
	int Age;
}
void SetAge(Person *p,int value)
{
	if (p==NULL)
		return;
	if ((value>0) && (value<200))
		p->Age = value;
}
void Init(Person *p)
{
	if (p==NULL)
		return;
	p->Age = 10;
}
void main()
{
	Person p;
	Init(&p);
	SetAge(&p,10);
}
```

App.cpp

```
class Person
{
	private:
		int Age;
	public:
		void SetAge(int value);
		Person();
}
void Person::SetAge(int value)
{
	if ((value>0) && (value<200))
		this->Age = value;
}
Person::Person()
{
	this->Age = 10;
}
void main()
{
	Person p;
	p.SetAge(10);
}
```

C to C++ (Conversions)

App.c

```
struct Person
{
      int Age;
}
void SetAge(Person *p,int value)
{
      if (p==NULL)
            return;
      if ((value>0) && (value<200))
            p->Age = value;
}
void Init(Person *p)
{
      if (p==NULL)
            return;
      p->Age = 10;
}
void main()
{
      Person p;
      Init(&p);
      SetAge(&p,10);
      p.Age = -1;
}
```

The code compiles and modifies the value of data member **Age**

App.cpp

```
class Person
{
      private:
            int Age;
      public:
            void SetAge(int value);
            Person();
}
void Person::SetAge(int value)
{
      if ((value>0) && (value<200))
            this->Age = value;
}
Person::Person()
{
      this->Age = 10;
}
void main()
{
      Person p;
      p.SetAge(10);
      p.Age = -1;
```

Compiler error – field Age is declared as **private**

- Classes

Classes (format)

- Member variables
  - Variable defined as member of the class
  - Each data member can have its own access modifier
  - Data member can also be static
  - A class may have no data members
- Member functions (methods)
  - Functions define within the class
  - Each method can have its own access modifier
  - A method can access any data member defined or other method defined in the class regardless of its access modifier
  - A class may have no methods
- Constructors
  - Methods without a return type that are called whenever an instance of a class is created
  - A class does not have to have constructors
  - A constructor may have different access modifiers
- Destructor
  - A function without a return type that is called whenever an instance of a class is destroyed
  - A class does not have to have a destructor
- Operators

Classes

- (Access modifiers)

Classes (access modifiers)

- The are 3 access modifiers defined in C++ language:
  - public (allow access to that member for everyone)
  - private (access to that member is only allowed from functions that were defined in that class). This is the default access modifier.
  - protected

Classes (access modifiers)

App.cpp

```
class Person
{
      public:
            int Age;
}
void main()
{
      Person p;
      p.Age = 10;
}
```

- The code compiles and runs correctly
- Member "Age" from class Person is declared public and may be access from outside the class.

Classes (access modifiers)

App.cpp

```
class Person
{
        private:
                int Age;
}
void main()
{
        Person p;
        p.Age = 10;
}
```

- This code won’t compile (Age is defined as private and can not be access outside its class/scope).

Classes (access modifiers)

App.cpp

```
class Person
{
      private:
           int Age;
      public:
           void SetAge(int val);
}
void Person::SetAge(int val)
{
      this->Age = val;
}
void main()
{
      Person p;
      p.SetAge(10);
}
```

- The code compiles and runs correctly
- From outside the class scope only the SetAge method is call (SetAge is defined as public)
- As SetAge is a method in class Person it can access any other method or data member regardless of their access modifiers.

Classes (access modifiers)

App.cpp

```
class Person
{
      int Age;
}
void main()
{
      Person p;
      p.Age = 10;
}
```

- This code won’t compile (member Age from class Person is private and can not be access outside its scope).
- If no access modifier is specified, the default access modifier will be private.

Classes (access modifiers)

App.cpp

```
class Person {
    int Age;
};
void main()
{
    Person p;
    p.Age = 10;
}
```

- This code won’t compile (member Age from class Person is **private** and can not be access outside its scope).

App.cpp

```
struct Person {
    int Age;
};
void main()
{
    Person p;
    p.Age = 10;
}
```

- This code will compile. C++ structures support access modifiers as well. However, the default access modifier for a structure is **public**.

Classes

- (Data members)

Classes (member data)

App.cpp

```
class Person
{
      private:
           int Age,Height;
      public:
           char *Name;
}
void main()
{
      Person p;

}
```

- Member data are variables defined within the class
- In this example – Age and Height are private, and Name is public

Classes (member data)

App.cpp

```
class Person
{
      private:
           int Age,Height;
      public:
           char *Name;
}
void main()
{
      Person p;
      p.Age = 10;
}
```

- This code does not compile because Age is a private data member

Classes (member data)

App.cpp

```
class Person
{
      private:
           int Age,Height;
      public:
           char *Name;
}
void main()
{
      Person p;
      p.Name = “Popescu”;
}
```

- This code compiles because Name is declared as public.

Classes (member data)

App.cpp

```
class Person
{
      private:
           int Age,Height;
           static int X;
      public:
           char *Name;
           static int Y;
}
```

- Member data can also be static and have access modifiers at the same time.

Classes (member data)

App.cpp

```
class Person
{
      private:
            int Age,Height;
            static int X;
      public:
            char *Name;
            static int Y;
}
int Person::X;
int Person::Y = 10;
```

- Member data can also be static and have access modifiers at the same time
- Any static data member that is defined within a class has to be defined outside its class as well (similar to a global variable).
- One can also initialize this static variables. If you do not initialize these variables, the result is identical to the use of global variables (the default value will be 0)

Classes (member data)

App.cpp

```
class Person
{
      private:
           int Age,Height;
           static int X;
      public:
           char *Name;
           static int Y;
}
int Person::X;
int Person::Y = 10;

void main()
{
      Person p;
      p.Y = 5;
      Person::Y++;
}
```

- Static data members can be access as the scope of the class or as a member from of any instance of that class.
- In this case, after the code is executed, Y will be 6.

Classes (member data)

App.cpp

```
class Person
{
      private:
           int Age,Height;
           static int X;
      public:
           char *Name;
           static int Y;
}
int Person::X;
int Person::Y = 10;

void main()
{
      Person p;
      p.X = 6;
}
```

- The code does not compile because X is private.
- We need to create a method to be able to access this value.

Classes (member data)

App.cpp

```
class Person
{
      private:
           int Age,Height;
           static int X;
      public:
           char *Name;
           static int Y;
           void SetX(int value);
}
int Person::X;
int Person::Y = 10;

void Person::SetX(int value)
{
      X = value;
}

void main()
{
      Person p;
      p.SetX(6);
}
```

- Now the code compiles and X value is set to 6

Classes (member data)

App.cpp

```
class C1
{
      int X,Y;
};
class C2
{
      int X,Y;
      static int Z;
};
class C3
{
      static int T;
};
class C4
{
};
int C2::Z;
int C3::T;
void main()
{
  printf("sizeof(C1)=%d",sizeof(C1));
  printf("sizeof(C2)=%d",sizeof(C2));
  printf("sizeof(C3)=%d",sizeof(C3));
  printf("sizeof(C4)=%d",sizeof(C4));
}
```

- The code compiles and runs correctly
- Static data members belong to the class and not to the instance – that's why they don't count when we compute the size of an instance
- A class may be defined without any data member. In this case it's size will be 1.
- Upon execution the program will print:

```
sizeof(C1) = 8
sizeof(C2) = 8
sizeof(C3) = 1
sizeof(C4) = 1
```

Classes (member data)

App.cpp

```
class Date
{
   public:
      int X,Y;
      static int Z;
};
int Date::Z;
void main()
{
      Date d1,d2,d3;
}
```

Address	Name	Value
100000	Date::Z	0
300000	d1.X	?
300004	d1.Y	?
300008	d2.X	?
300012	d2.Y	?
300016	d3.X	?
300020	d3.Y	?

Classes (member data)

App.cpp

```
class Date
{
   public:
      int X,Y;
      static int Z;
};
int Date::Z;
void main()
{
      Date d1,d2,d3;
      d1.Z = 5;
}
```

Address	Name	Value
100000	Date::Z	5
300000	d1.X	?
300004	d1.Y	?
300008	d2.X	?
300012	d2.Y	?
300016	d3.X	?
300020	d3.Y	?

Classes (member data)

App.cpp

```
class Date
{
   public:
      int X,Y;
      static int Z;
};
int Date::Z;
void main()
{
      Date d1,d2,d3;
      d1.Z = 5;
      d1.X = 7;
}
```

Address	Name	Value
100000	Date::Z	5
300000	d1.X	7
300004	d1.Y	?
300008	d2.X	?
300012	d2.Y	?
300016	d3.X	?
300020	d3.Y	?

Classes (member data)

App.cpp

```
class Date
{
   public:
      int X,Y;
      static int Z;
};
int Date::Z;
void main()
{
      Date d1,d2,d3;
      d1.Z = 5;
      d1.X = 7;
      d2.Y = d3.Z + 1;
}
```

Address	Name	Value
100000	Date::Z	5
300000	d1.X	7
300004	d1.Y	?
300008	d2.X	?
300012	d2.Y	6
300016	d3.X	?
300020	d3.Y	?

Classes (member data)

App.cpp

```
class Date
{
   public:
      int X,Y;
      static int Z;
};
int Date::Z;
void main()
{
      Date d1,d2,d3;
      d1.Z = 5;
      d1.X = 7;
      d2.Y = d3.Z + 1;
      Date::Z = d2.Z + 1;
}
```

Address	Name	Value
100000	Date::Z	6
300000	d1.X	7
300004	d1.Y	?
300008	d2.X	?
300012	d2.Y	6
300016	d3.X	?
300020	d3.Y	?

Classes (member data)

App.cpp

```
class Date
{
   public:
      int X,Y;
      static int Z;
};
int Date::Z;
void main()
{
      Date d1,d2,d3;
      d1.Z = 5;
      d1.X = 7;
      d2.Y = d3.Z + 1;
      Date::Z = d2.Z + 1;
      d3.X = d2.Z+d1.Z-1;
}
```

Address	Name	Value
100000	Date::Z	6
300000	d1.X	7
300004	d1.Y	?
300008	d2.X	?
300012	d2.Y	6
300016	d3.X	11
300020	d3.Y	?

Classes (Methods)

Classes (methods)

App.cpp

```
class Person
{
	private:
		int Age;
		bool CheckValid(int val);
	public:
		void SetAge(int val);
};
bool Person::CheckValid(int val)
{
	return ((val>0) && (val<200));
}
void Person::SetAge(int val)
{
	if (CheckValid(val))
		this->Age = val;
}
void main()
{
	Person p;
	p.SetAge(40);
}
```

- Methods are functions define within the class. Their main role is to operate and change data members from the class (especially private ones)
- Just like data member, a method can have an access modifier.
- A method can access any other method declared in the same scope (that belongs to the same class) regardless of that methods access modifier.

Classes (methods)

App.cpp

```
class Person
{
private:
      int Age;
public:
      static bool Check(int val);
      void SetAge(int val);
};
bool Person::Check(int val)
{
      return ((val>0) && (val<200));
}
void Person::SetAge(int val)
{
      if (Check(val))
            this->Age = val;
}
void main()
{
      Person p;
      if (Person::Check(40))
      {
      printf("40 is a valid age");
      }
}
```

- A method can be static and have an access modifier at the same time.

Classes (methods)

App.cpp

```
class Person
{
private:
      int Age;
      static bool Check(int val);
public:
      void SetAge(int val);
};
bool Person::Check(int val)
{
      return ((val>0) && (val<200));
}
void Person::SetAge(int val)
{
      if (Check(val))
            this->Age = val;
}
void main()
{
      Person p;
      if (Person::Check(40))
      {
      printf("40 is a valid age");
      }
}
```

- In this case the code does not compile because Check method is declared private,

Classes (methods)

App.cpp

```
class Person
{
private:
      int Age;
      static bool Check(int val);
public:
      void SetAge(int val);
};
bool Person::Check(int val)
{
      return ((val>0) && (val<200));
}
void Person::SetAge(int val)
{
      if (Check(val))
            this->Age = val;
}
void main()
{
      Person p;
      p.SetAge(40);
}
```

- The code compiles - SetAge method is public and can be called
- Any method (even if it is declared private like method Check) can be accessed by another method declared in that class (in this case SetAge)

Classes (methods)

App.cpp

```
class Date
{
private:
      int X;
      static int Y;
public:
      static void Increment();
};
int Date::Y = 0;
void Date::Increment()
{
      Y++;
}
void main()
{
      Date::Increment();
}
```

- A static method can access any static member declared in the same scope as the method regardless of that member access modifier
- In this example, Increment function will add 1 to the static and private data member Y.

Classes (methods)

App.cpp

```cpp
class Date
{
private:
	int X;
	static int Y;
public:
	static void Increment();
};
int Date::Y = 0;
void Date::Increment()
{
	X++;
}
void main()
{
	Date::Increment();
}
```

- This code does not compile
- A static function can not access a non-static member
- A static function can not access the pointer **this**

Classes (methods)

App.cpp

```
class Person
{
private:
    int Age;
public:
    void SetAge( Person * p, int value)
    {
        p->Age = value;
    }
};

int main()
{
    Person p1, p2;
    p1.SetAge(&p2, 10);
    return 0;
}
```

- A method within a class can access private members / methods from instances of the same class !
- In this case, ***p1*** can access data member ***Age*** from ***p2***. The cod compiles and runs correctly.

Classes (methods)

App.cpp

```
class Person
{
private:
    int Age;
public:
    static void SetAge( Person * p, int value)
    {
        p->Age = value;
    }
};

int main()
{
    Person p1, p2;
    p1.SetAge(&p2, 10);
    Person::SetAge(&p1, 20);
    return 0;
}
```

- The same rule applies for static methods as well.
- In this example, the code compiles and runs correctly. After the execution, ***p1.Age*** is 20 and ***p2.Age*** is 10.

Q & A