

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

The background features a large, abstract graphic on the left side composed of various shades of blue and dark blue triangles. These triangles overlap and extend from the top-left corner towards the center, creating a sense of depth and movement.

► 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 → 1 point for labs 1 to 7, labs 10 to 12 → 10 points
- ▶ Minimum requirements to pass OOP exam:
 - capability to model and build POO programs in C ++ that solve relatively simple problems
 - capability to correctly apply OOP principles (encapsulation, inheritance, polymorphism)
 - capability to write C ++ programs based on specifications
 - ability to understand OO programs written in C ++
 - ability to detect simple errors in a program and correct them

A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented diagonally, creating a sense of depth and movement.

► Glossary

Glossary

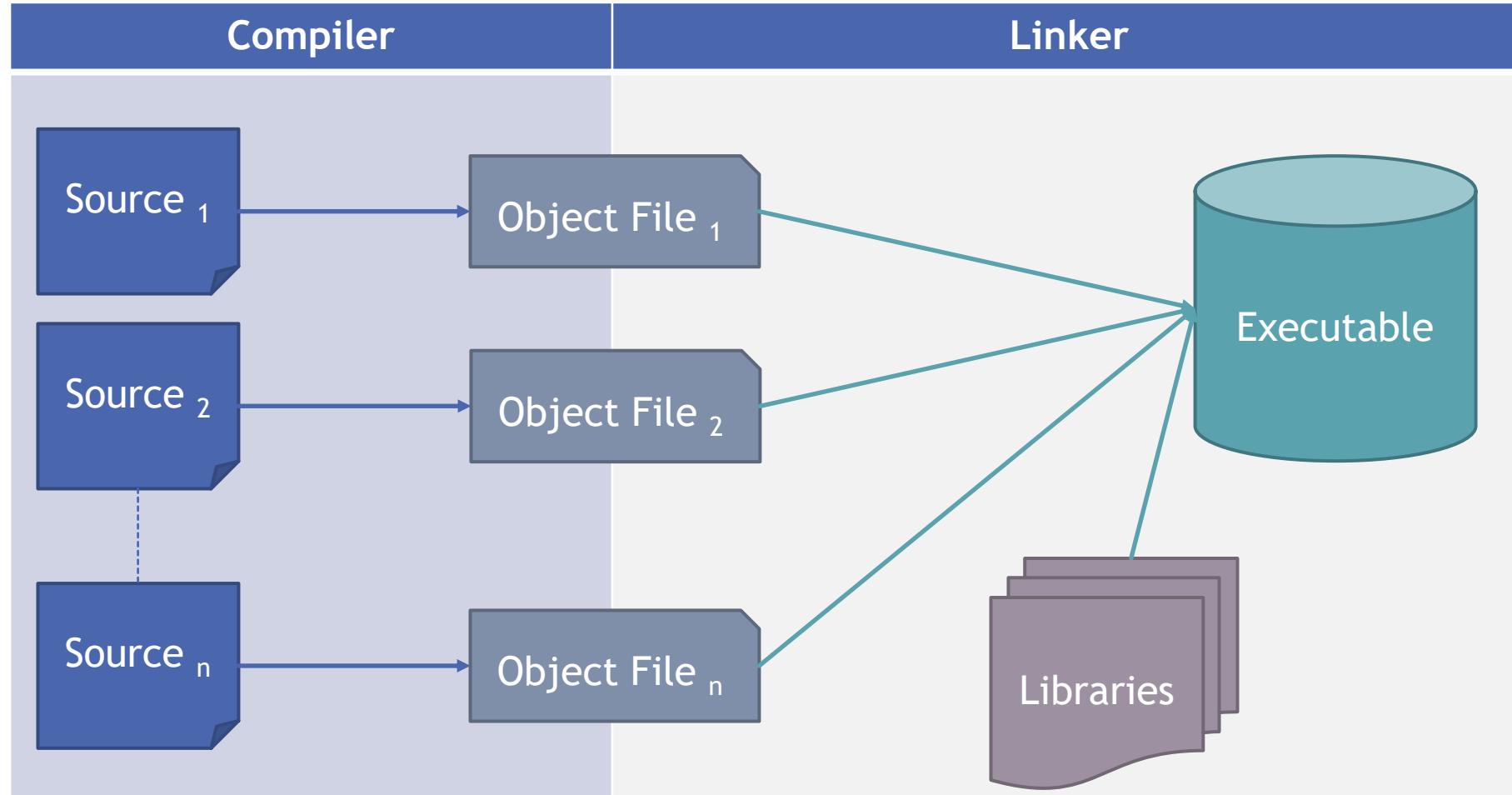
- ▶ API → Application Program Interface
- ▶ Library - a set of functions that can be used by multiple programs at the same time (for example math functions like cos, sin, tan, etc)
- ▶ GUI → Graphic User Interface

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. 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

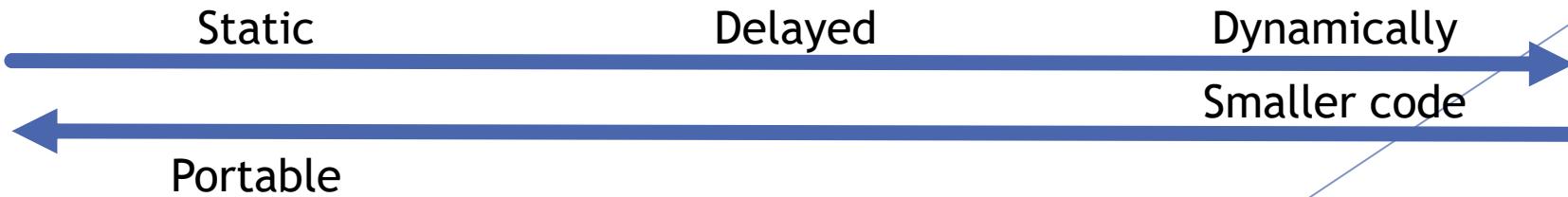


Glossary



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 created.
- ▶ 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).



A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented at various angles, creating a sense of depth and movement. They are set against a solid dark blue background.

- ▶ 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

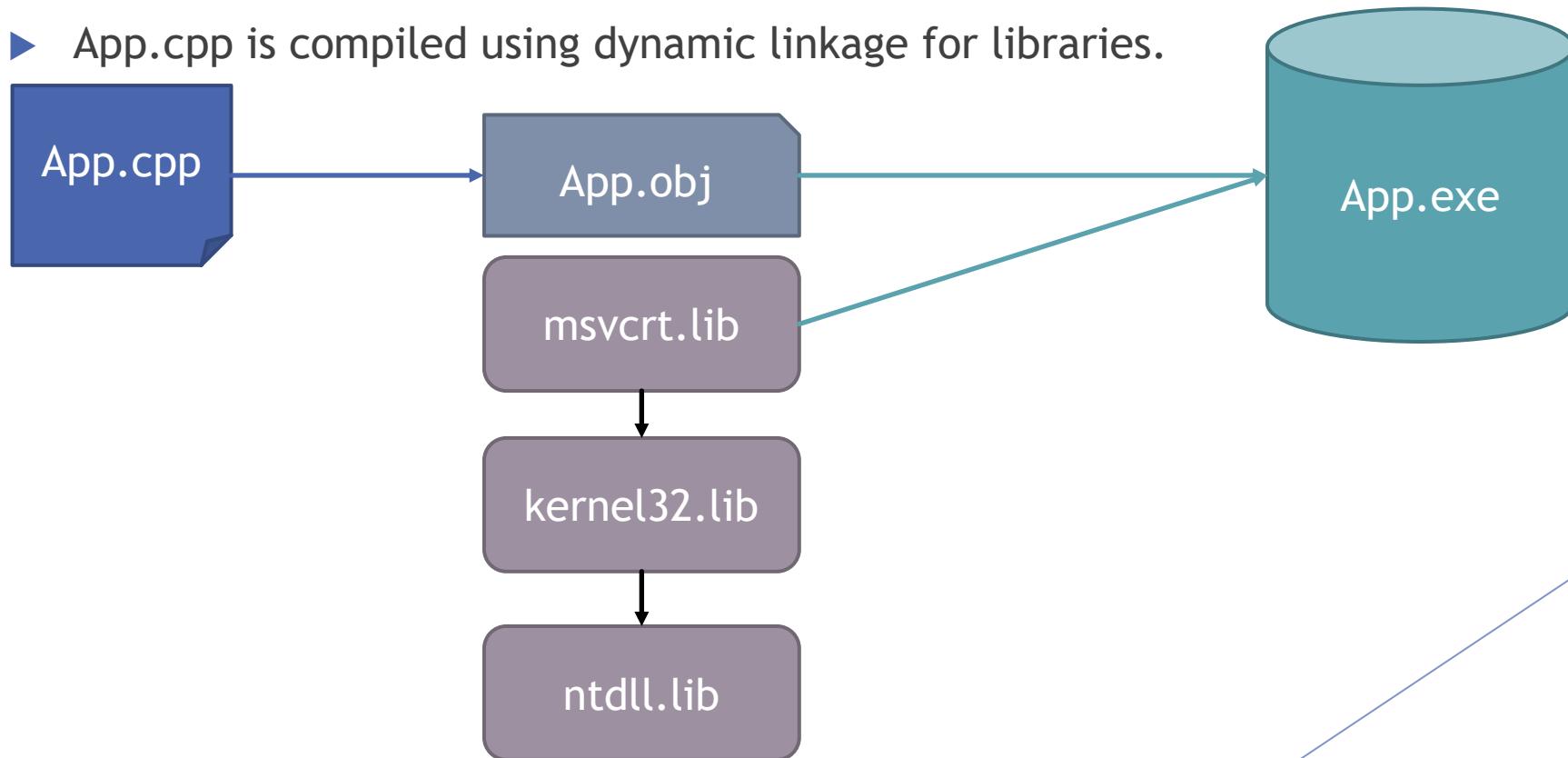
```
#include <stdio.h>
int vector[100];

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 !");
}
```

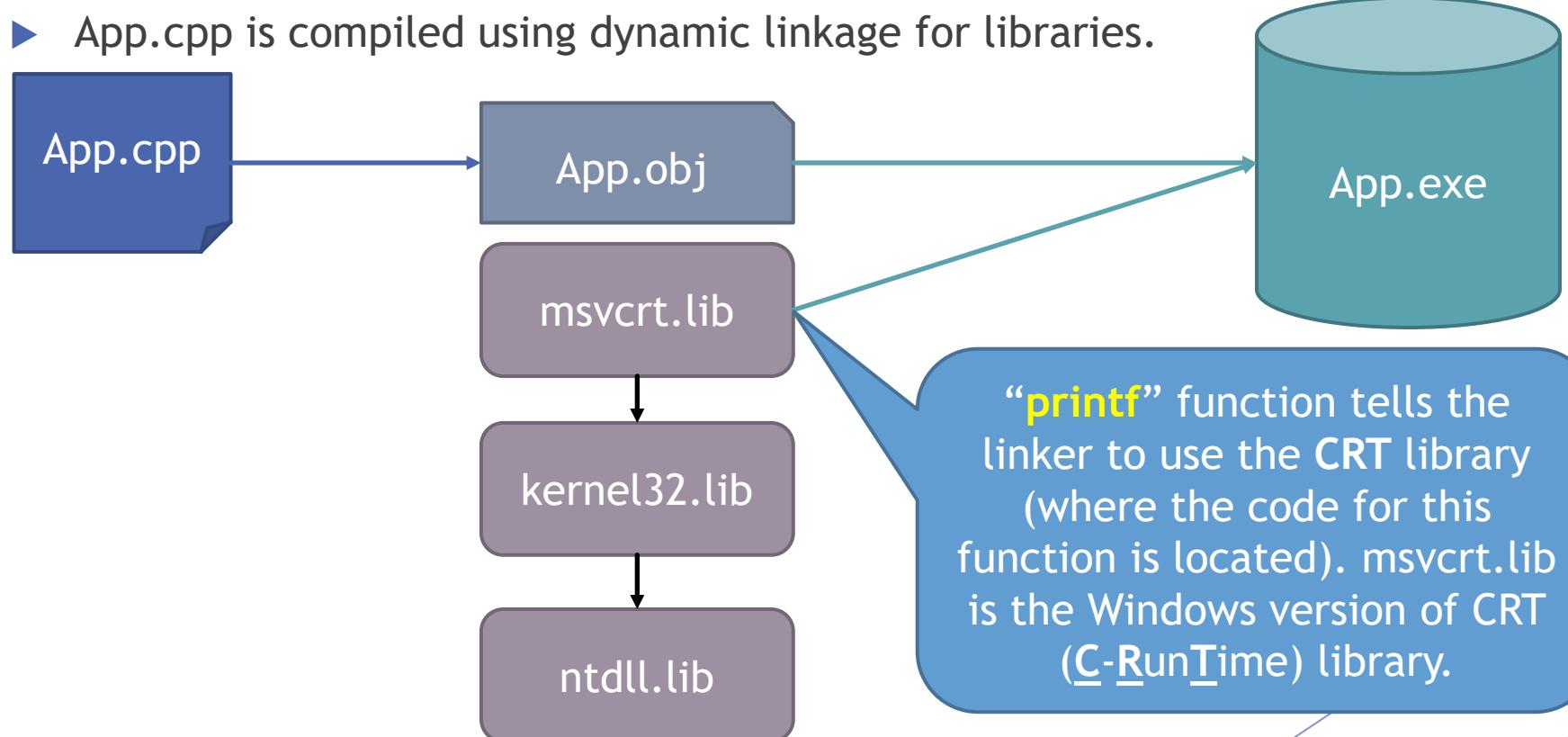
OS Architecture

- ▶ 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.



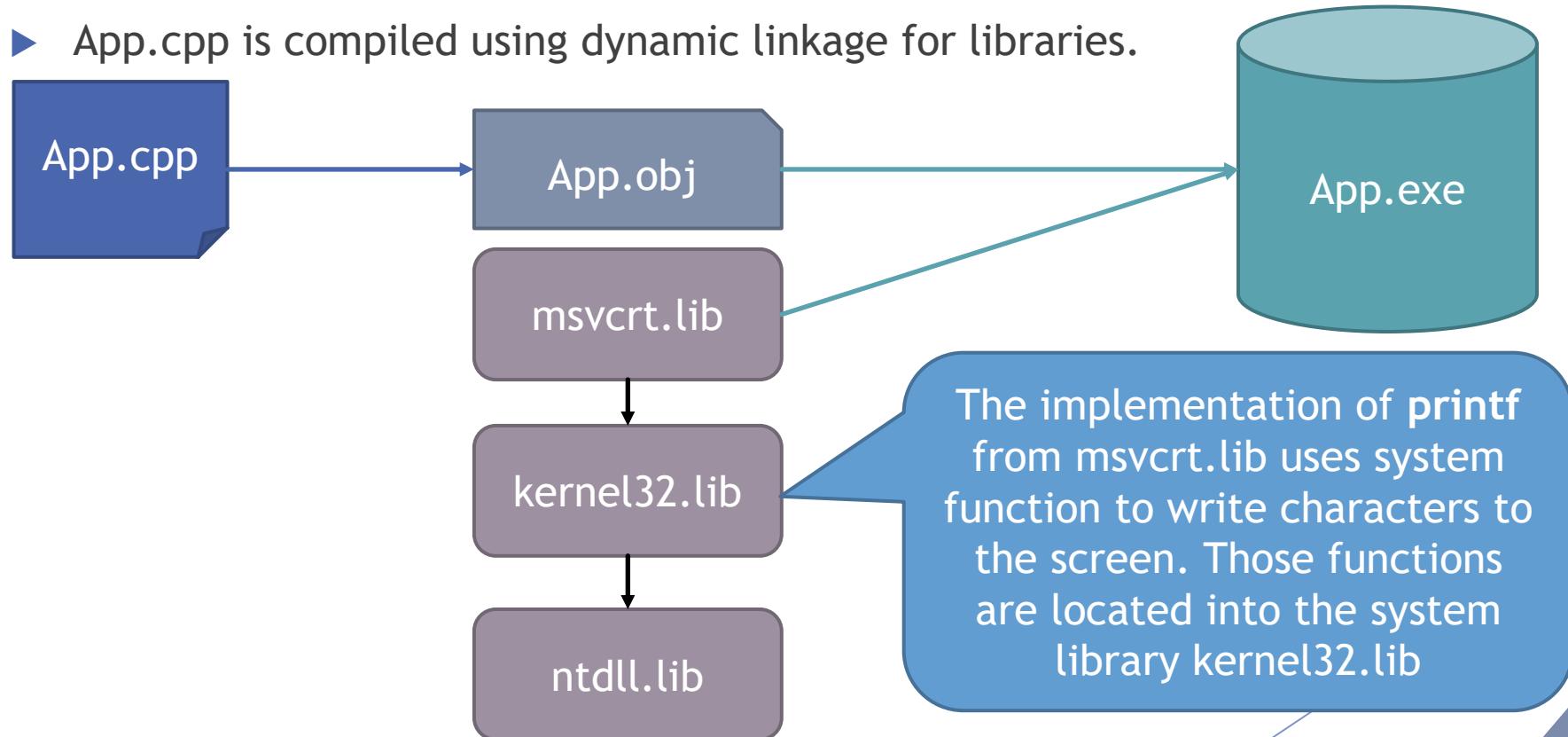
OS Architecture

- ▶ 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.



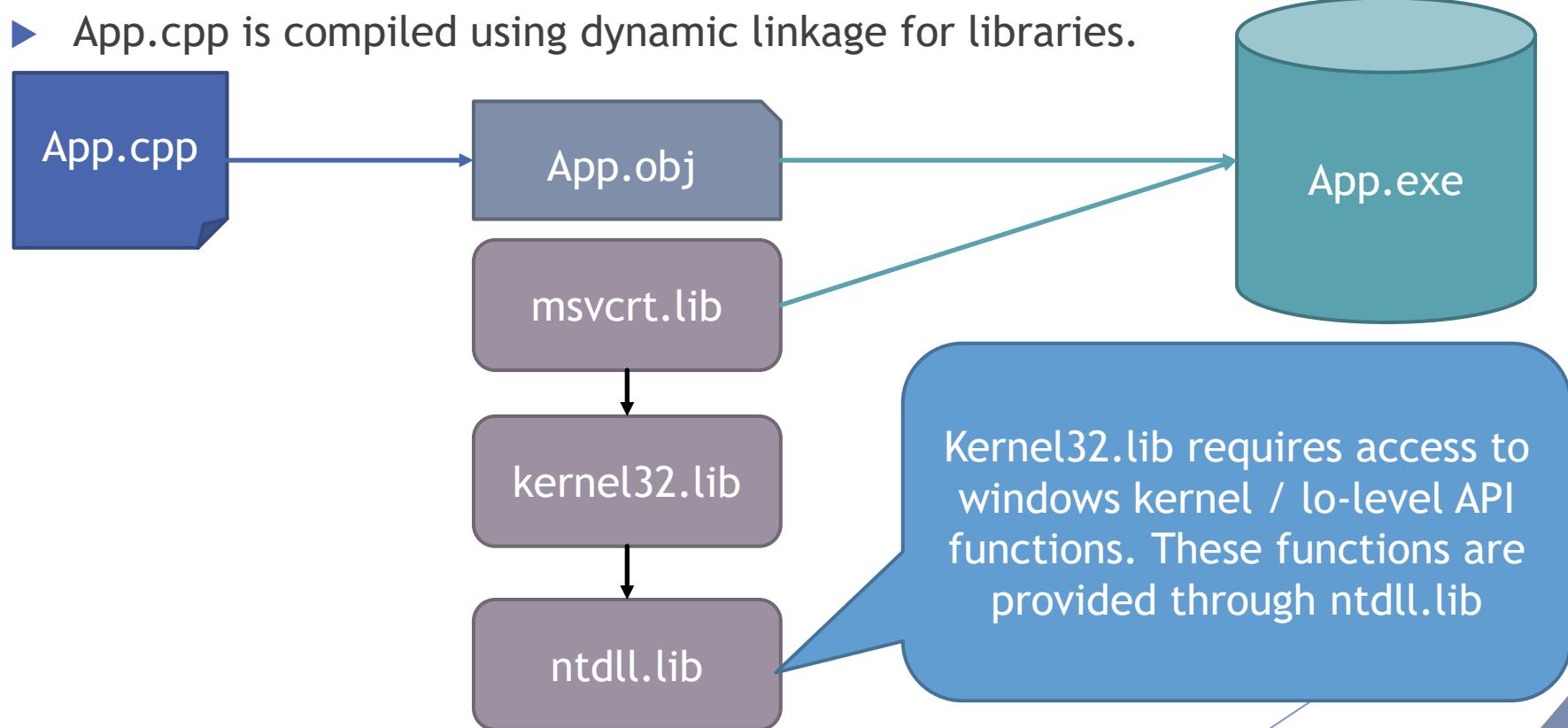
OS Architecture

- ▶ 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.

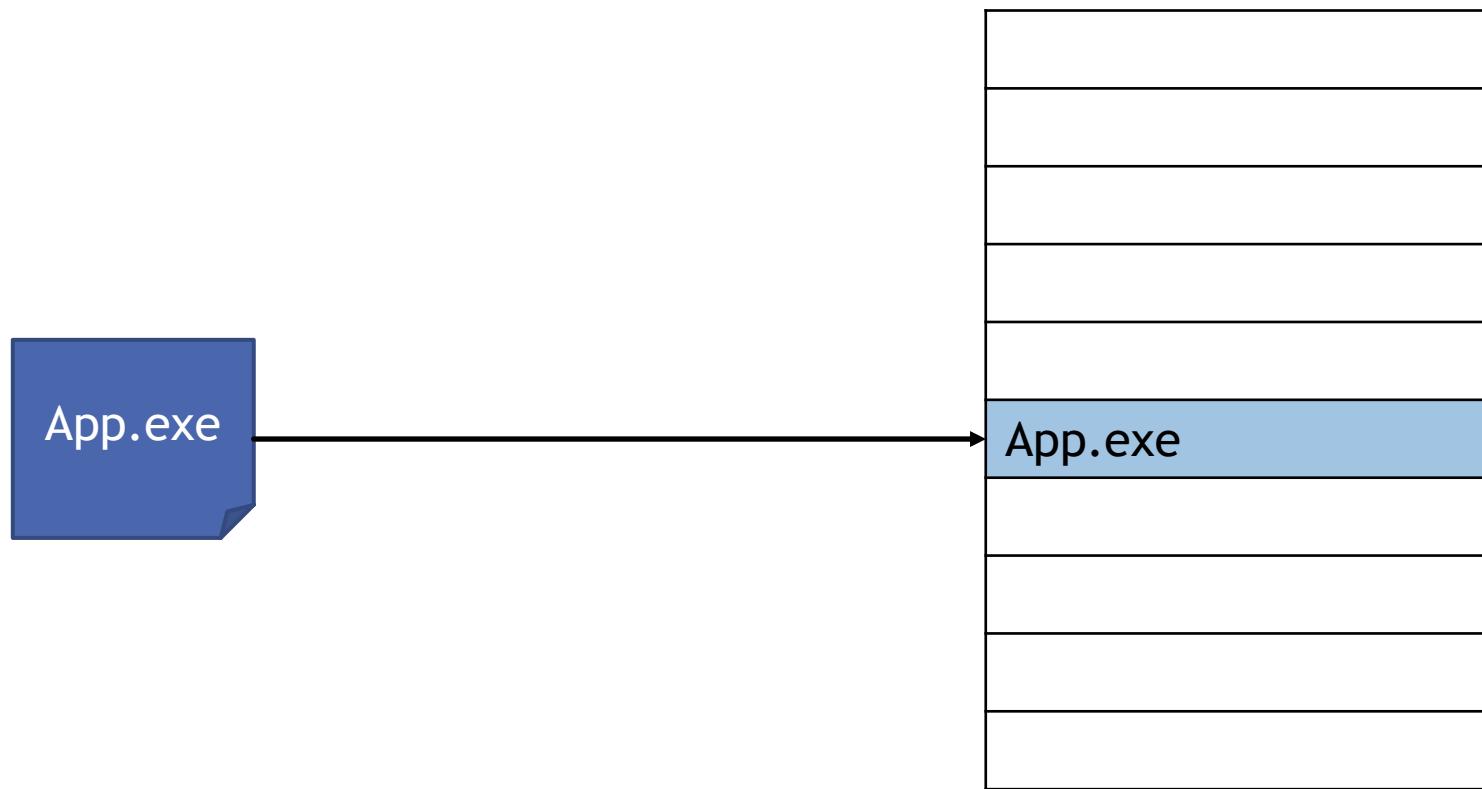


OS Architecture

- ▶ What happens when a.exe is executed:

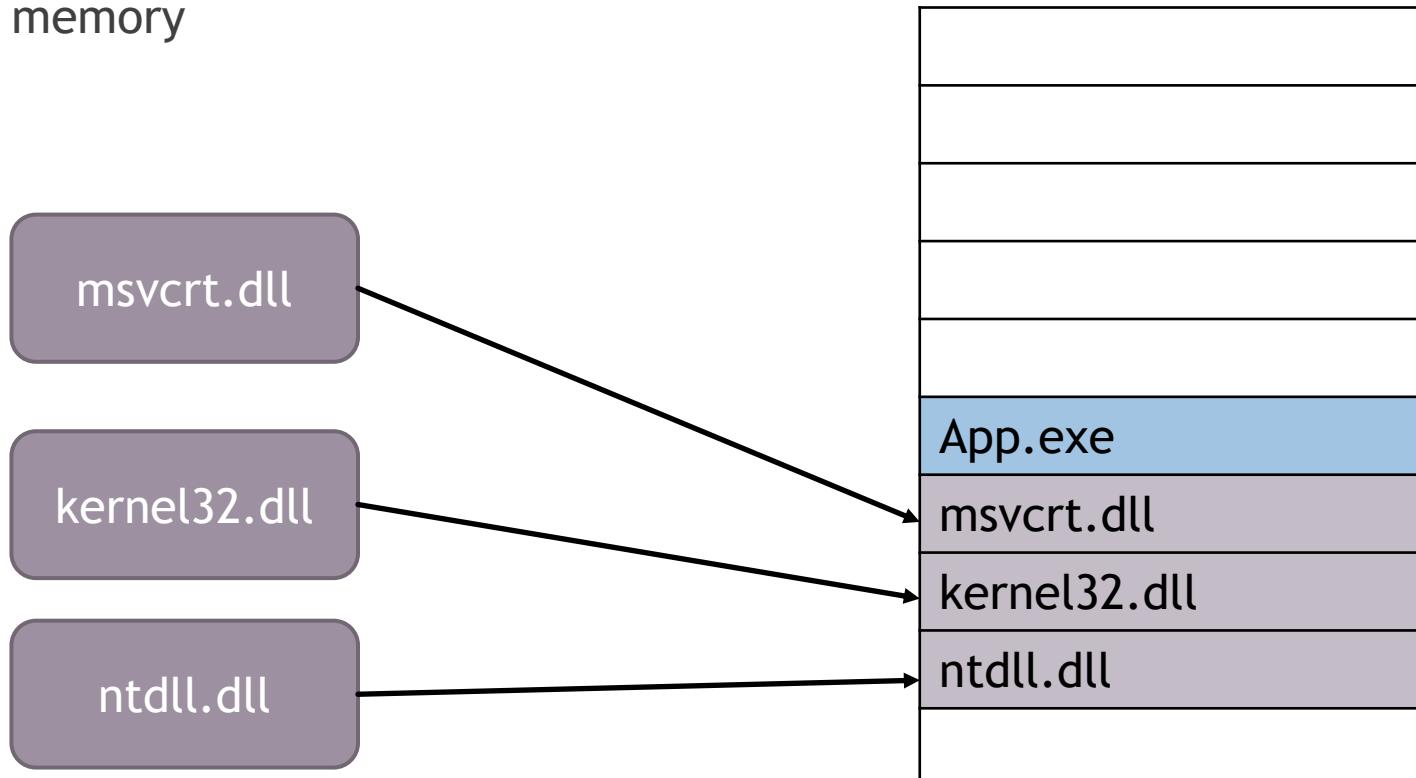
OS Architecture

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



OS Architecture

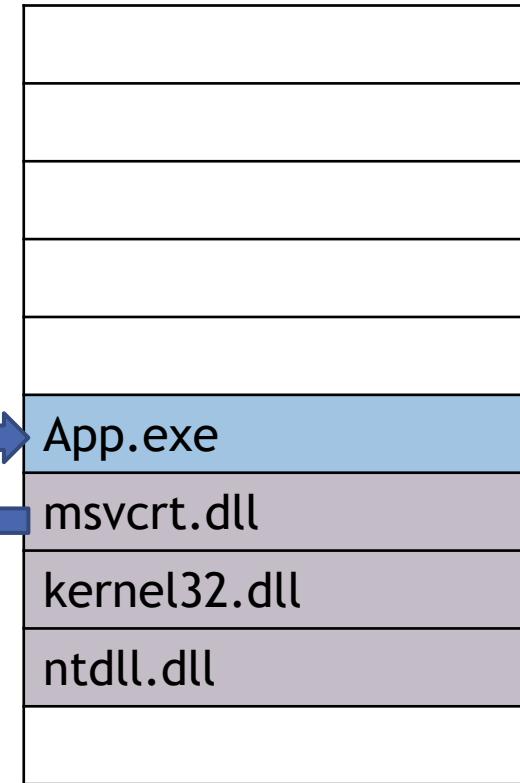
- Content of the libraries that are needed by “a.exe” is copied in the process memory



OS Architecture

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

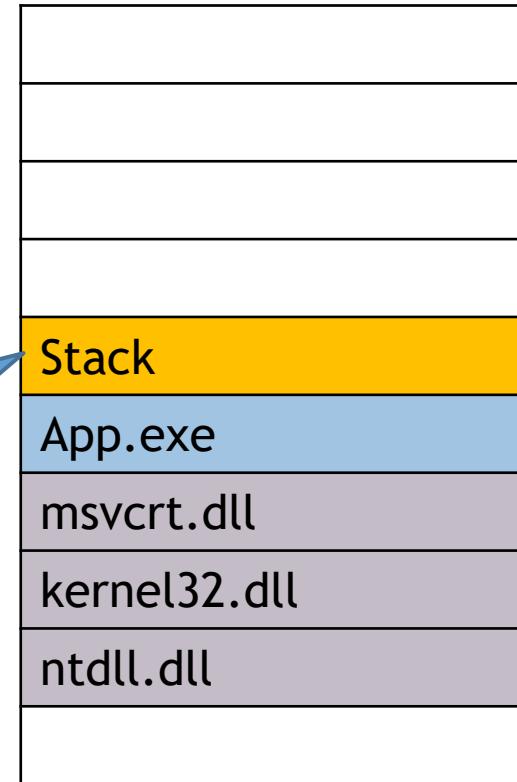
Address of “**printf**” function is imported in App.exe from the msvcrt.dll (CRT library)



OS Architecture

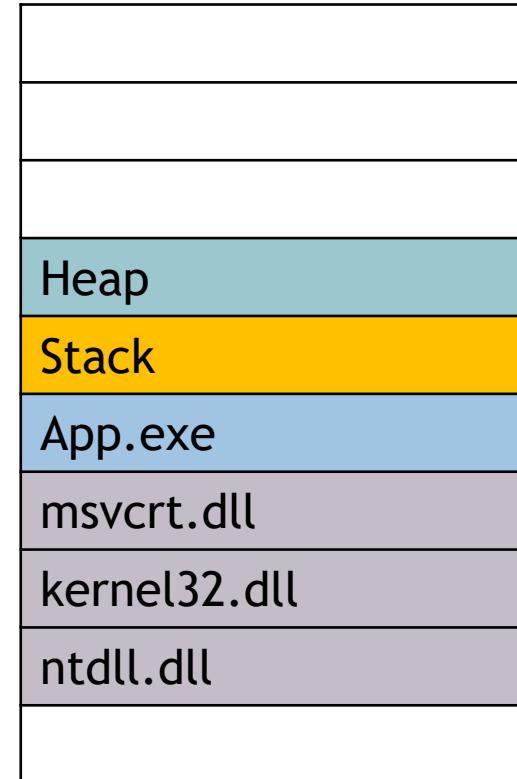
- ▶ Stack memory is created. In our example, variable **poz**, **i**, and parameter **n** will be stored into this memory.
- ▶ This memory is not initialized. That is why local variables have undefined values.
- ▶ Every execution thread has its own stack

A stack memory is allocated for the current thread.
EVERY local variable and function parameters will be stored into this stack



OS Architecture

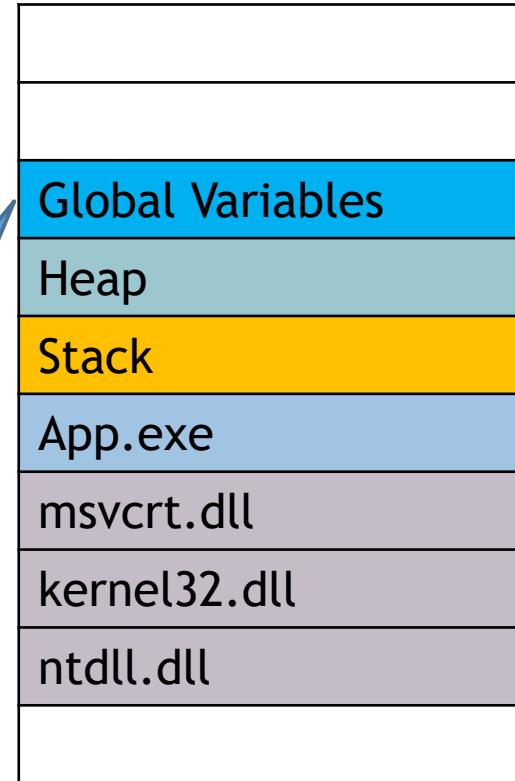
- ▶ Heap memory is allocated. Heap memory is a large memory from where smaller buffers are allocated. Heap is used by the following functions:
 - ▶ Operator new
 - ▶ malloc, calloc, etc
- ▶ Heap memory is not initialized.
- ▶ The same heap can be used by multiple threads



OS Architecture

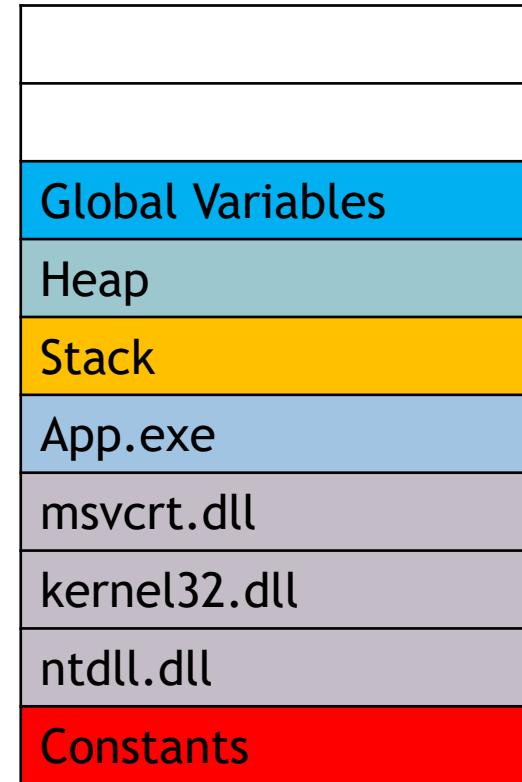
- ▶ A memory for global variable is allocated. This memory is initialized by default with 0 values. This is where all global variables are stored. If a global variable has a default value (different than 0), that value will be set into this memory space.
- ▶ In our case, variable **vector** will be stored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

`int vector[100]`



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	Pseudo - code	Stack Address	Var
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

- ▶ 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'; // Line 4
    s3 = 'c';
    p = &s1;
    *p = '0';
    p[1] = '1';
    *(p+2) = '2';
}
```

Pseudo - code

```
Stack[99] = 'a'  
Stack[98] = 'b'  
Stack[97] = ?  
Stack[96] = ?
```

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`

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

OS Architecture (memory alignment)

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

`sizeof(Test) = 12`

OS Architecture (memory alignment)

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

`sizeof(Test) = 12`

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	s	?	t	t	u	u	u	u							
0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	2	2	2	2	2	2	3	3	3	

OS Architecture (memory alignment)

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

`sizeof(Test) = 24`

OS Architecture (memory alignment)

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

`sizeof(Test) = 12`

OS Architecture (memory alignment)

```
#pragma pack(1)
struct Test
{
    char      x;
    short     y;
    int       z;
    char      t;
};
```

`sizeof(Test) = 8`

OS Architecture (memory alignment)

```
#pragma pack(2)
struct Test
{
    char      x;
    short     y;
    int       z;
    char      t;
};
```

`sizeof(Test) = 10`

OS Architecture (memory alignment)

```
#pragma pack(1)
_declspec(alignment(16)) struct Test
{
    char      x;
    short     y;
    int       z;
    char      t;
};
```

`sizeof(Test) = 16`

OS Architecture (memory alignment)

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

`sizeof(Test) = 20`

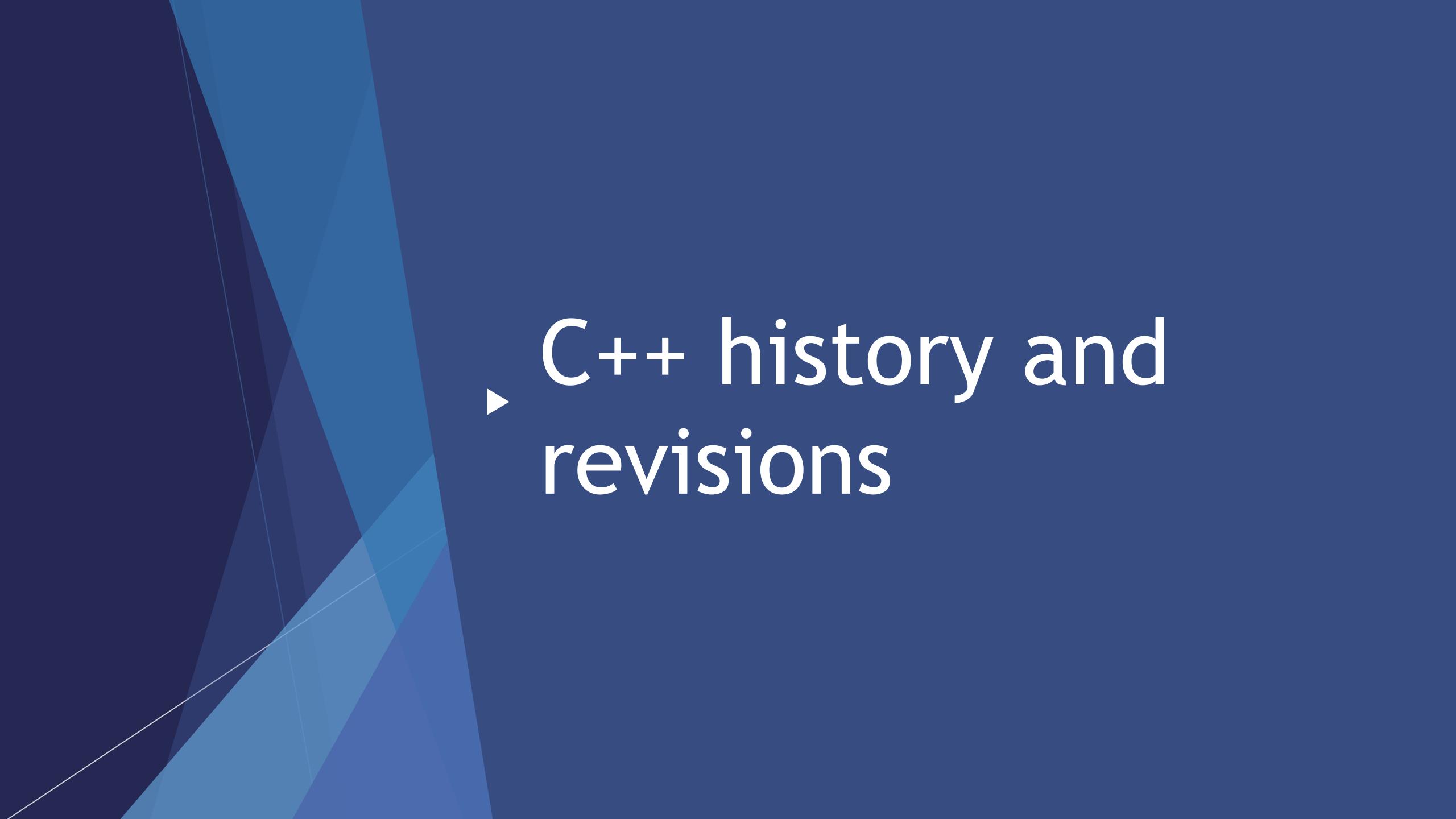
```
struct Test2
{
    char      x
    short     y
    int       z
};
```

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)

A decorative graphic in the top-left corner consists of several overlapping blue triangles of varying shades, creating a geometric pattern.

▶ 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

C++98

Keywords	<code>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</code>
Operators	<code>{ } [] # ## () <: :> <% %> %: %:%: ; : ... new delete ? :: . . * + * / % ^ & ~ ! = < > += = *= /= %= ^= &= = << >> >>= <<= == != <= >= && ++ , >* ></code>

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	2020	C++20
GCC/G++	GNU Compiler	10.2	C++20
Clang (LLVM)		12.0.0 13.0.0 (in progress)	C++20

The background features a dark blue gradient with a subtle geometric pattern of lighter blue triangles and lines.

► 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

```
class Person
{
public:
    int Age;
};
```

```
Person::Person()
{
    Age = 10;
}
```

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

```
int Person::GetAge() const
{
    return Age;
}
```

```
int main()
{
    Person p;
    cout << p.GetAge();
}
```

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

The background features a dark blue gradient with a subtle geometric pattern of lighter blue triangles and lines.

► 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

The background features a large, abstract graphic on the left side composed of overlapping blue and dark blue triangles and trapezoids, creating a sense of depth and motion.

Classes

- ▶ (Access modifiers)

Classes (access modifiers)

- ▶ There 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 accessed 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 accessed 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**.

The background features a large, abstract graphic on the left side composed of overlapping blue triangles of varying shades of blue. It has a subtle, organic, fan-like or leaf-like pattern.

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 cu 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 accessed within the scope of the class or as a member from 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.SetValue(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	?

A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented diagonally, creating a sense of depth and movement.

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

```
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 code 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.

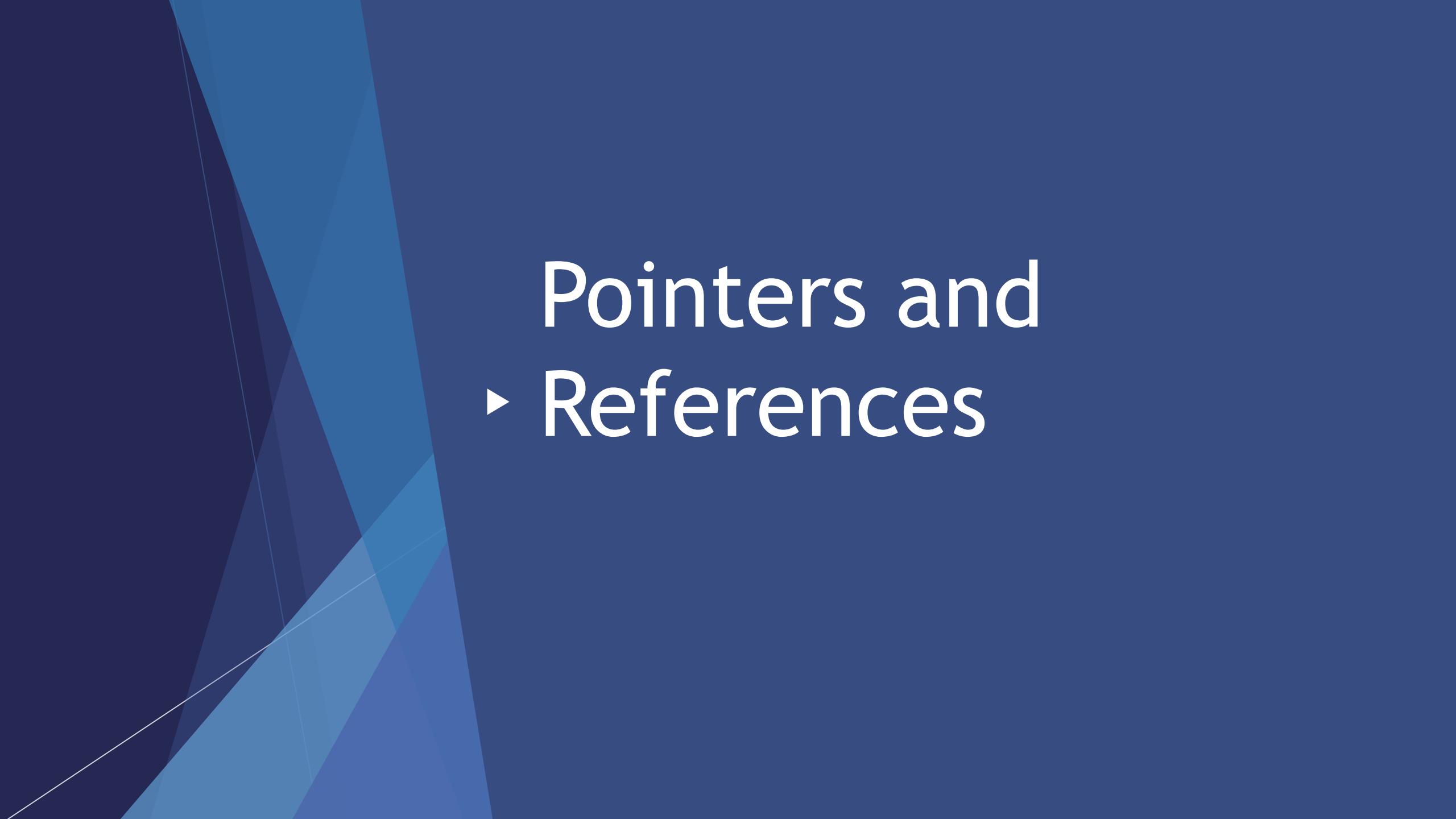
Q & A

OOP

Gavrilut Dragos
Course 2

Summary

- ▶ Pointers and References
- ▶ Method overloading
- ▶ NULL pointer
- ▶ “const” specifier
- ▶ “friend” specifier

A decorative graphic in the top-left corner consists of several overlapping blue triangles of varying shades, creating a geometric pattern.

Pointers and ▶ References

Pointers and References

App-Pointer

```
void SetInt(int *i)
{
    (*i) = 5;
}
void main()
{
    int x;
    Set(&x);
}
```

App-Reference

```
void SetInt(int &i)
{
    i = 5;
}
void main()
{
    int x;
    Set(x);
}
```

App-Pointer (asm - SetInt)

```
SetInt:
    push    ebp
    mov     ebp,esp
    mov     eax,[ebp+8]
    mov     [eax],5
    mov     esp,ebp
    pop    ebp
    ret
```

App-Reference (asm - SetInt)

```
SetInt:
    push    ebp
    mov     ebp,esp
    mov     eax,[ebp+8]
    mov     [eax],5
    mov     esp,ebp
    pop    ebp
    ret
```

Pointers and References

- ▶ The resulted code is identical (both the “Pointer” and “Reference” program will link into the same assembler code).
- ▶ However, from the programmer point of view, using a reference fixes some possible problems (perhaps the most know one is that one does not need to use the “->” operator - instead the “.” operator can be used). Another important one is that a check for NULL pointers is no longer required.

Pointer

```
struct Date
{
    int X;
}
void SetInt(Date *d)
{
    d->X = 5;
}
```

Reference

```
struct Date
{
    int X;
}
void SetInt(Date &d)
{
    d.X = 5;
}
```

Pointers and References

- ▶ References and pointers are created in the following manner:

Pointer

```
int i = 10;  
int *p = &i;
```

Reference

```
int i = 10;  
int &refI = i;
```

- ▶ The difference is that pointers can remain uninitialized.

Pointer

```
int i = 10;  
int *p;
```

Reference

```
int i = 10;  
int &refI;
```

Compile error -
uninitialized reference

This forces the programmer to initialize a reference.

It also guarantees that a reference points to a valid memory location.

Pointers and References

- ▶ A pointer value can be changed (that is a pointer can point to different memory addresses). A reference can only point to a variable and once it is initialized the memory address where it points to can not be changed.

Pointer

```
int i = 10;  
int j = 20;  
int *p = &i;  
p = &j;
```

Reference

```
int i = 10;  
int j = 20;  
int &refI = i;  
&refI = j;
```

Compiler error - trying
to change a reference
that was already
initialized.

- ▶ A pointer can have the value NULL.
A reference can only point to a memory address
that exists.

Pointers and References

- ▶ Pointers accept certain arithmetic operations (+, - , ++, etc). This is not valid for references.

Pointer

```
int i = 10;  
int j = 20;  
int *p = &i;  
p++;  
(*p) = 30;
```

Reference

```
int i = 10;  
int j = 20;  
int &refI = i;  
refI++;  
(&refI)++;
```

Compile error

- ▶ In case of pointers, variable “i” and “j” are allocated consecutively on the stack. The operation “p++” moves the pointer p from the memory address of the variable “i” to the memory address of the variable “j”. At the end of the execution “j” will have the value 30.

Pointers and References

- ▶ A pointer can be converted to another pointer (cast). In particular any pointer can be converted to a void pointer (`void*`). A reference can not be converted to another reference.

Pointer

```
int i = 10;  
char *p = (char *)&i;
```

Reference

```
int i = 10;  
char &refI = i;
```

Compile error

- ▶ This thing guarantees that a reference points to a memory address where a certain type of variable resides.

Pointers and References

- ▶ A pointer may point to another pointer and so on. This is not possible for references - a reference refers only a variable.

Pointer

```
int i = 10;  
int *p = &i;  
int *p_to_p = &p;  
**p_to_p = 20;
```

Reference

```
int i = 10;  
int &refI = i;  
int & &ref_to_refI = refI;
```

Compile error

- ▶ It is important in this example to differentiate between “`& &`” (two references separated with a space (‘ ‘) character) and “`&&`” (two consecutive references).

Pointers and References

- ▶ A pointer can be used in an array and be dynamically initialized. This is not possible for references.

Pointer

```
int *p[100];
```

Reference

```
int &ref[100];
```

Compile error

- ▶ However, a reference may point to a temporary (or constant) value.

Pointer

```
int *p = &int(10);
```

Reference

```
const int &refI = int(12);
```

This code will not compile if the “const” specifier is not used as it refers to a constant numerical value.

Pointers and References

- ▶ A pointer can be used in an array and be dynamically initialized. This is not possible for references.

Pointer

```
int *p[100];
```

Reference

```
int &ref[100];
```

Compile error

- ▶ However, a reference may point to a temporary (or constant) value.

Pointer

```
const int &refI = int(12);  
int *p = (int *)&refI;
```

Reference

```
const int &refI = int(12);
```

It is however possible to create a pointer that points to a reference of a temporary (constant) value.

- ▶ Method overloading

Method overloading

- ▶ Method overloading is a technique used in C++ where one can define 2 or multiple functions/methods with the same name (or operators).
- ▶ A function / method is uniquely identified by its signature:

return-type ***FunctionName (param1-type, param2-type, ...)***

A function/method signature is form out of:
1) function name
2) Parameters type (if parameters are present)

- ▶ Since parameters are part of the function signature, multiple functions/methods with the same name but different parameters are possible.
- ▶ However, this does not apply to return type (meaning that functions with the same name and parameters but different return type can not exit).

Method overloading

App.cpp

```
class Math
{
public:
    int Add (int v1, int v2);
    int Add (int v1, int v2, int v3);
    int Add (int v1, int v2, int v3, int v4);
    float Add (float v1, float v2);
};
int Math::Add(int v1, int v2)
{
    return v1 + v2;
}
int Math::Add(int v1, int v2, int v3)
{
    return v1 + v2 + v3;
}
int Math::Add(int v1, int v2, int v3, int v4)
{
    return v1 + v2 + v3 + v4;
}
float Math::Add(float v1, float v2)
{
    return v1 + v2;
}
```

Method overloading

- ▶ Method overloading is NOT possible if the methods have the same signature (same name, same parameters)
- ▶ In the next case, both methods are named *Add* and have two parameters of type *int*). The return type (even if in this case is different) will not be considered , thus the two *Add* functions are consider duplicates !

App.cpp

```
class Math
{
public:
    int Add(int v1, int v2);
    long Add(int v1, int v2);
};
int Math::Add(int v1, int v2)
{
    return v1 + v2;
}
long Math::Add(int v1, int v2)
{
    return v1 + v2;
}
```

Method overloading

- ▶ Be careful when you are using parameters with default value. From the compiler point of view, using this feature does not mean that a function has fewer parameters !
- ▶ This code will NOT compile as *Add* has the same signature !

App.cpp

```
class Math
{
public:
    int Add(int v1, int v2);
    long Add(int v1, int v2 = 0);
};
int Math::Add(int v1, int v2)
{
    return v1 + v2;
}
long Math::Add(int v1, int v2)
{
    return v1 + v2;
}
```

Method overloading

- ▶ Another special case are methods with variadic parameters (“...”). However, they are not recommended in case of method overloading as the interpretation can be misleading.

App.cpp

```
class Math
{
public:
    int Add(int v1, int v2);
    long Add(int v1, ...);
};

int Math::Add(int v1, int v2)
{
    return v1 + v2;
}

long Math::Add(int v1, ...)
{
    return v1;
}
```

Method overloading

- ▶ When a function/method that was overloaded is called, the compiler determines which one of the existing definitions of that function/method it should use. This process is called **overload resolution**
- ▶ It is possible that the result of this process will be inconclusive (e.g. - the compiler can not decide the best fit for a specific name). In this case a compiler error will be raised, and the ambiguity will be explained.

Method overloading

Overload resolution steps:

1. Check if an exact match is possible (a method exists with the same name and the exact same parameters type)

Defined	<code>void Compute(int x, double y, char z)</code>
Called	<code>Compute(100, 1.5, 'A')</code>

Method overloading

Overload resolution steps:

2. Check if a numerical promotion is possible (convert a type into another one without losing precision and the value).

- ✓ `bool`, `char`, `short`, `unsigned char` and `unsigned short` can be promoted to `int`
- ✓ `float` can be promoted to `double`
- ✓ Any enumeration (`enum`) without an explicit type can be converted to `int`

Defined	<code>void Compute(int x, double y, char z)</code>
Called	<code>Compute(true, 1.5f, 'A')</code>
Promotion	<code>Compute(1, 1.5, 'A')</code>

`true` is promoted to `int` value (1)
`1.5f` (a float value) is promoted to double value 1.5

Method overloading

Overload resolution steps:

3. Check if a numerical conversion is possible (convert a type into another one with the possibility of loosing the actual value / precision).

Defined	<code>void Compute(int x, double y, char z)</code>
Called	<code>Compute(3.5, 1.5, 'A')</code>
Conversion	<code>Compute(3, 1.5, 'A')</code>

3.5 (a double value) will be converted to *int* value 3
(loosing precision)

It is possible that the conversion may apply to several overloaded methods. If this is a case, an ambiguity error will be thrown, and the program will not compile

Method overloading

Overload resolution steps:

4. Casts are attempted:

- ✓ Every non-const pointer can be casted to its const pointer form
- ✓ Every non-const pointer can be casted to `void *` or `const void *`
- ✓ Every const pointer can be casted to `const void *`
- ✓ `NULL` macro (define) can be converted to numerical value `0`

Defined	<code>void Compute(int x, const void* y, char z)</code>
Called	<code>Compute(NULL, "C++", 'A')</code>
Cast	<code>Compute(0, (const void*)"C++", 'A')</code>

`NULL` is converted to `int` value `0`
`"C++"` (a `const char *` pointer) is cast to `const void *`

Method overloading

Overload resolution steps:

5. Explicit casts (if any) are applied. We will discuss more on this topic when we will study inheritance and C++ operators.
6. If none of these attempts result in finding a match - a fallback method / function (if any) is used. A fallback method is a method that only has variadic parameters
7. If there isn't such a method , the compiler will produce an error.

Method overloading

- ▶ 100 is considered an “*int*” type value. Since there is a method by the name *Inc* that has a parameter of type “*int*”, the compiler will use that method.
- ▶ In this case we have an exact-match situation.

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    float Inc(float v1);
};

int Math::Inc(int v1)
{
    return v1 + 1;
}

float Math::Inc(float v1)
{
    return v1+1.0f;
}

void main()
{
    Math m;
    m.Inc(100);
}
```

Method overloading

- ▶ 1.0f is a “*float*” value. Since there is a method by the name *Inc* that has a parameter of type “*float*”, the compiler will use that method.
- ▶ In this case we have an exact-match situation.

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    float Inc(float v1);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
float Math::Inc(float v1)
{
    return v1+1.0f;
}
void main()
{
    Math m;
    m.Inc(1.0f);
}
```

Method overloading

- ▶ ‘a’ is a “*char*” type value. As there is no method with the name *Inc* that has one parameter of type “*char*”, the compiler promotes the *char* value to *int* and uses the *Inc* method with one parameter of type “*int*” .

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    float Inc(float v1);
};

int Math::Inc(int v1)
{
    return v1 + 1;
}

float Math::Inc(float v1)
{
    return v1+1.0f;
}

void main()
{
    Math m;
    m.Inc('a');
}
```

Method overloading

- ▶ ‘a’ is a “*char*” type value. As there is no method with the name *Inc* that has one parameter of type “*char*”, the compiler promotes the *char* value to *int* and tries again. Since there is no *Inc* method that has an *int* parameter (but there are two *Inc* methods, an ambiguity case will be declared, and the code will not compile. Event if, a *char* can fully be converted (without any value lost) into a *short* , promotion only works for *int* and *double* types.

App.cpp

```
class Math
{
public:
    int Inc(short v1);
    float Inc(float v1);
};
int Math::Inc(short v1)
{
    return v1 + 1;
}
float Math::Inc(float v1)
{
    return v1+1.0f;
}
void main()
{
    Math m;
    m.Inc('a');
}
```

Method overloading

- ▶ If during the promotion phase the compiler DOES NOT find any possible promotion, but there are at least *two* methods/functions with the same name as the one attempted to be promoted the compiler will throw an error (this will be considered to be an ambiguity case). Having at least two methods with the same name is an indicator that method overloading is desired and another overload for the specific call is required.
- ▶ However, if the promotion fails and there only **ONE** method with that name, a conversion is attempted (in this case it is considered that method overloading was not something desired by the programmer and the compiler attempts to match the parameters even if this means losing precision / value).

Method overloading

- ▶ 1.0 is a double value. As there is not any *Inc* method that receives a *double* parameter, promotion is attempted. Unfortunately - can not be promoted (without loosing value) to either *int* or *float*.
- ▶ Since there are two *Inc* function, this code will not compile, and an ambiguity case will be explained as an error.

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    float Inc(float v1);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
float Math::Inc(float v1)
{
    return v1+1.0f;
}
void main()
{
    Math m;
    m.Inc(1.0);
}
```

Method overloading

- ▶ In this case, 1.0 is a double value, and as there is no *Inc* function that has one parameter of type double, promotion is attempted. Unfortunately, *double* can not be converted to *char* without losing precision.
- ▶ However, as there is only *ONE* function Inc, the compiler will convert double to char (even if this means losing precision). This code will compile with warnings.

App.cpp

```
class Math
{
public:
    int Inc(char v1) {
        return v1 + 1;
    }
};

void main()
{
    Math m;
    m.Inc(1.0);
}
```

warning C4244: 'argument':
conversion from 'double' to
'char', possible loss of data

Method overloading

- ▶ In this case, 1.0 is a double value, and as there is no *Inc* function that has one parameter of type double, promotion is attempted. Unfortunately, *double* can not be converted to *char* without loosing precision.
- ▶ However, as there is only *ONE* function Inc, the compiler will try to convert *double* to *char **. As this is not possible (only conversions numerical conversions are possible) the compiler will produce an error and the code will not compile.

App.cpp

```
class Math
{
public:
    int Inc(char* v1) { return 1; }
};
void main()
{
    Math m;
    m.Inc(1.0);
}
```

error C2664: 'int Math::Inc(char *)':
cannot convert argument 1 from
'double' to 'char *'

Method overloading

- ▶ Pointer conversions are also impossible. “`&d`” is a “*double **” that can not be converted to “*char **”.
- ▶ This code will produce a compiler error.

App.cpp

```
class Math
{
public:
    int Inc(char* v1);
};

int Math::Inc(char* v1)
{
    return 1;
}

void main()
{
    Math m;
    double d = 1.0;
    m.Inc(&d);
}
```

error C2664: 'int Math::Inc(char *)':
cannot convert argument 1 from
'double *' to 'char *'

Method overloading

- ▶ Pointer conversions are also impossible. “`&d`” is a “*double **” that can not be converted to “*char **”.
- ▶ However, using an **explicit cast** will solve this problem. In this case, the code will compile.

App.cpp

```
class Math
{
public:
    int Inc(char* v1);
};

int Math::Inc(char* v1)
{
    return 1;
}

void main()
{
    Math m;
    double d = 1.0;
    m.Inc( (char *)&d );
}
```

Method overloading

- ▶ However, any non-constant pointer can be converted to “**void ***”. The next example will compile.
- ▶ A constant pointer can not be converted to a non-constant pointer implicitly (without a cast). A non-constant pointer can always be converted to its constant equivalence. That is why, if you don’t need to modify the value where the pointer points, it is best to use **const** pointers for method/function parameters.

App.cpp

```
class Math
{
public:
    int Inc(void* v1);
};

int Math::Inc(void* v1)
{
    return 1;
}

void main()
{
    Math m;
    double d = 1.0;
    m.Inc(&d);
}
```

Method overloading

- ▶ Methods with variadic parameters:
 1. **Fallback methods** → methods with only one parameter that is variadic (with a signature in the form `<name> (...)`). These methods are the last to be used (only if there is no possible conversion, cast or promotion or if there is no ambiguity in terms of promotion/conversion). These functions are not allowed in C language.
 2. **Regular methods** → methods that have at least one parameter that is not variadic and **ONE** variadic parameter (e.g. `<name>(int,...)` or `<name>(char,short,...)`). These methods are used just like the regular methods and the same rules apply to them as well.

Method overloading

- ▶ In case of methods with variadic parameters the compiler will use the best fit (in terms of exact parameters). This code compiles and the method *Inc(int)* is used.

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(123);
}
```

Method overloading

- In this case, as there is no *Inc* method that has a parameter of type *char*, the compiler promotes `a` (char value **97**) to *int* and uses the *Inc(int)* method.

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc('a');
}
```

Method overloading

- ▶ 1.0 is a double value. We can not apply numerical promotion to int to use *Inc(int)* method. However, we can convert the double to an int (with possible loss of value) and then use *Inc(int)* method. The code compiles. The fallback function is used only if no promotion/conversion is possible.

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(1.0);
}
```

Method overloading

- ▶ This is an ambiguous case. There is no promotion possible. However, the double value 1.0 can be converted to both *int* (and use *Inc(int)* method, or *float* and use *Inc(float)* method). As there are two possibilities, this code is considered ambiguous and an error is thrown.

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(float v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(float v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main() {
    Math m;
    m.Inc(1.0);
}
```

error C2668: 'Math::Inc': ambiguous call to overloaded function
note: could be 'int Math::Inc(float)'
note: or 'int Math::Inc(int)'
note: while trying to match the argument list '(double)'

Method overloading

- In this case the parameter used is a *const char ** (a pointer). There is no promotion and no conversion possible. Thus, the compiler must use the fallback method *Inc(...)*

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc("test");
}
```

Method overloading

- ▶ A similar case → there is no method overloaded with 2 parameters, so the compiler uses the fallback method *Inc(...)*

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(1.0,2);
}
```

Method overloading

- ▶ This is an ambiguous case. 123 is an *int* value and there are two methods that match exactly with the call *Inc(123)* : *Inc(int)* and *Inc(int,...)*. The code will NOT compile.

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(int v1,...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(int v1,...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(123);
}
```

```
warning C4326: return type of 'main' should be 'int' instead of 'void'
error C2668: 'Math::Inc': ambiguous call to overloaded function
note: could be 'int Math::Inc(int,...)'
note: or          'int Math::Inc(int)'
note: while trying to match the argument list '(int)'
```

Method overloading

- ▶ This is an ambiguous case. *true* is a *bool* value and we don't have an exact method to match *Inc(bool)*. In this case numerical promotion is apply, *bool* is promoted to *int* and now we have two methods that match: *Inc(int)* and *Inc(int,...)*. The code will NOT compile.

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(int v1,...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(int v1,...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc(true);
}
```

error C2668: 'Math::Inc': ambiguous call to overloaded function
note: could be 'int Math::Inc(int,...)'
note: or 'int Math::Inc(int)'
note: while trying to match the argument list '(bool)'

Method overloading

- ▶ This code will NOT compile. None of the methods *Inc(int)* and *Inc(int,...)* matches the *const char ** parameter and there is no fallback method.

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(int v1,...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(int v1,...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc("test");
}
```

error C2664: 'int Math::Inc(int,...)': cannot convert
argument 1 from 'const char [5]' to 'int'
note: There is no context in which this conversion is
possible

Method overloading

- ▶ This code will compile. ‘a’ (*char*) is promoted to *int* and since there is only one method that accepts two parameters, the compiler will use it.

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(int v1,...);
};
int Math::Inc(int v1)
{
    return v1 + 1;
}
int Math::Inc(int v1,...)
{
    return 1;
}
void main()
{
    Math m;
    m.Inc('a',true);
}
```

Method overloading

- ▶ This code will compile. ‘a’ (*char*) is promoted to *int* and since there is only one method that accepts two parameters, the compiler will use it.
- ▶ Fallback methods (*Inc(...)*) are used only if no match is possible.

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(int v1,...);
    int Inc(...);
};
int Math::Inc(int v1) {
    return v1 + 1;
}
int Math::Inc(int v1,...) {
    return 1;
}
int Math::Inc(...) {
    return 2;
}
void main()
{
    Math m;
    m.Inc('a',true);
}
```

Method overloading

- ▶ This code will compile. However, in this case there is no match possible from *Inc(const char *, bool)* (including promotions and conversions) to the existing methods *Inc(int)* and *Inc(int,...)*. However, as a fallback function is also available , the compiler will choose to use it.

App.cpp

```
class Math
{
public:
    int Inc(int v1);
    int Inc(int v1,...);
    int Inc(...);
};
int Math::Inc(int v1) {
    return v1 + 1;
}
int Math::Inc(int v1,...) {
    return 1;
}
int Math::Inc(...) {
    return 2;
}
void main()
{
    Math m;
    m.Inc("test",true);
}
```

Method overloading

- ▶ When dealing with overloaded methods with multiple parameters, promotion and conversion rules are evaluated for each parameter.
- ▶ **Overload resolution** will choose the combination of promotion/conversions that covers highest number of unique parameters.
- ▶ If there are at least two solutions that have at least one different parameter that each one of the solution can cover (match / promote or convert) an ambiguous case is considered and an error will be thrown.
- ▶ If nothing matches, a promotion is considered stronger than a conversions , and can be used to resolve overload resolution (in fact a promotion is sometimes considered equal as importance with an exact match).

Method overloading

- ▶ This code will compile. There is an exact match (a function *Add* with two parameters, first of type *char* and the second of type *int*).

App.cpp

```
class Math
{
public:
    void Add(char x, int y);
    void Add(int x, char y);
};
void Math::Add(char x, int y)
{
    printf("Add(char,int)");
}
void Math::Add(int x, char y)
{
    printf("Add(int,char)");
}
void main()
{
    Math m;
    m.Add('a',100);
}
```

Method overloading

- ▶ This code compiles. The compiler promotes the second parameter from *bool* to *int* and uses *Add(char,int)*

App.cpp

```
class Math
{
public:
    void Add(char x, int y);
    void Add(int x, char y);
};

void Math::Add(char x, int y)
{
    printf("Add(char,int)");
}
void Math::Add(int x, char y)
{
    printf("Add(int,char)");
}

void main()
{
    Math m;
    m.Add('a',true);
}
```

Method overloading

- ▶ This is an ambiguous case as we have two possibilities:
 - a. $100 = \text{int}$, we convert $200(\text{int})$ to char and use $\text{Add}(\text{int}, \text{char})$
 - b. $100(\text{int})$ is converted to char , 200 is considered an int and we use $\text{Add}(\text{char}, \text{int})$

App.cpp

```
class Math
{
public:
    void Add(char x, int y);
    void Add(int x, char y);
};
void Math::Add(char x, int y)
{
    printf("Add(char,int)");
}
void Math::Add(int x, char y)
{
    printf("Add(int,char)");
}
void main()
{
    Math m;
    m.Add(100,200);
}
```

error C2666: 'Math::Add': 2 overloads have similar conversions
note: could be 'void Math::Add(int,char)'
note: or 'void Math::Add(char,int)'
note: while trying to match the argument list '(int, int)'

Method overloading

- ▶ This code compiles. We also have two possibilities, but the first one is better:
 - 100 = *int*, 1.5 is converted to *char* and we use *Add(int,char)* [one conversion + one exact match]
 - 100 (*int*) is converted to *char*, 1.5 to *int* and we use *Add(char,int)* [two conversions]

App.cpp

```
class Math
{
public:
    void Add(char x, int y);
    void Add(int x, char y);
};
void Math::Add(char x, int y)
{
    printf("Add(char,int)");
}
void Math::Add(int x, char y)
{
    printf("Add(int,char)");
}
void main()
{
    Math m;
    m.Add(100,1.5);
}
```

Method overloading

- ▶ This code compiles. We also have three possibilities:
 - a. ‘a’ = char (exact match), 1.5 (a double is converted to int) and 2.5 (a double is converted to int) → **1 x exact match, 2 x conversion**
 - b. ‘a’ (char is promoted to int), 1.5 (a double is converted to char) and 2.5 (a double is converted to int) → **1 x promotion, 2 x conversion**
 - c. ‘a’ (char is converted to float), 1.5 (a double is converted to bool) and 2.5 (a double is converted to int) → **3 x conversion**
- ▶ Solution a) has an exact match and since there is no other solutions that can match another parameter than the first one, this will be selected.

App.cpp

```
class Math
{
public:
    int Add(char x, int y, int z) { return 1; }
    int Add(int x, char y, int z) { return 2; }
    int Add(float x, bool y, int z) { return 3; }
};
void main()
{
    Math m;
    int x = m.Add('a', 1.5, 2.5);
}
```

Method overloading

- ▶ This code will NOT compile. We also have three possibilities:
 - a. 'a' = char (exact match), 1.5f (a float is converted to int) and 2.5 (a double is converted to int) → **1 x exact match, 2 x conversion**
 - b. 'a' (char is promoted to int), 1.5f (a float is converted to char) and 2.5 (a double is converted to int) → **1 x promotion, 2 x conversion**
 - c. 'a' = char (exact match), 1.5f (a float is converted to bool) and 2.5 (a double is converted to int) → **1 x exact match, 2 x conversion**
- ▶ Since both solutions a) and c) have an exact match for the 1st parameter, this will be considered an ambiguity case.

App.cpp

```
class Math
{
public:
    int Add(char x, int y, int z) { return 1; }
    int Add(int x, char y, int z) { return 2; }
    int Add(char x, bool y, int z) { return 3; }
};
void main()
{
    Math m;
    int x = m.Add('a', 1.5f, 2.5);
}
```

```
error C2668: 'Math::Add': ambiguous call to overloaded function
note: could be 'int Math::Add(char,bool,int)'
note: or      'int Math::Add(char,int,int)'
note: while trying to match the argument list '(char, float, double)'
```

Method overloading

- ▶ This code will NOT compile. We also have three possibilities:
 - a. ‘a’ = char (exact match), ‘a’ (a char is promoted to int) and 2.5 (a double is converted to int) →
1 x exact match, 1 x conversion, 1 x promotion
 - b. ‘a’ (char is promoted to int), ‘a’ = char (exact match) and 2.5 (a double is converted to int) →
1 x exact match, 1 x conversion, 1 x promotion
 - c. ‘a’ = char (exact match), ‘a’ (a char is converted to bool) and 2.5 (a double is converted to int)
→ **1 x exact match, 2 x conversion**
- ▶ Solution a) and c) have an exact match for the 1st parameter (but not for the second one). Solution b) has an exact match for the 2nd parameter (but not for the first one). This is considered an ambiguity.

App.cpp

```
class Math
{
public:
    int Add(char x, int y, int z) { return 1; }
    int Add(int x, char y, int z) { return 2; }
    int Add(char x, bool y, int z) { return 3; }
};
void main()
{
    Math m;
    int x = m.Add('a', 'a', 2.5);
}
```

error C2666: 'Math::Add': 3 overloads have similar conversions
note: could be 'int Math::Add(char,bool,int)'
note: or 'int Math::Add(int,char,int)'
note: or 'int Math::Add(char,int,int)'
note: while trying to match the argument list '(char, char, double)'

Method overloading

- ▶ Let's consider the following code:

App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call `Math::Add` with the following parameters:

```
Math m; m.Add(1.5, true, 1.5)
```

#		X (double)	Y (bool)	Z (double)
1.	Add (<i>char x, int y, int z</i>)	conversion	promotion	conversion
2.	Add (<i>double x, int y, int z</i>)	exact match	promotion	conversion
3.	Add (<i>char x, bool y, char z</i>)	conversion	exact match	conversion

- ▶ Solution 1 (has an exact match for the first parameter), Solution 2 has an exact match for the second parameter → **ambiguity case**

Method overloading

- ▶ Let's consider the following code:

App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call Math::Add with the following parameters:

```
Math m; m.Add(false, true, 1.5)
```

#		X (bool)	Y (bool)	Z (double)
1.	Add (<i>char x, int y, int z</i>)	conversion	promotion	conversion
2.	Add (<i>double x, int y, int z</i>)	conversion	promotion	conversion
3.	Add (<i>char x, bool y, char z</i>)	conversion	exact match	conversion

- ▶ One solution that has a match (solution 3). There is no promotion or exact match for “X” or “Z” in solution 1 or 2 → code will compile

Method overloading

- ▶ Let's consider the following code:

App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call `Math::Add` with the following parameters:

```
Math m; m.Add(1.5, true, 1.5)
```

#		X (double)	Y (bool)	Z (double)
1.	Add (<i>char x, int y, int z</i>)	conversion	promotion	conversion
2.	Add (<i>double x, int y, int z</i>)	exact match	promotion	conversion
3.	Add (<i>char x, bool y, char z</i>)	conversion	exact match	conversion

- ▶ Solution 1 (has an exact match for the first parameter), Solution 2 has an exact match for the second parameter → **ambiguity case**

Method overloading

- ▶ Let's consider the following code:

App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call `Math::Add` with the following parameters:

```
Math m; m.Add('a', true, 1.5)
```

#		X (char)	Y (bool)	Z (double)
1.	Add (<i>char x, int y, int z</i>)	exact match	promotion	conversion
2.	Add (<i>double x, int y, int z</i>)	conversion	promotion	conversion
3.	Add (<i>char x, bool y, char z</i>)	exact match	exact match	conversion

- ▶ Solution 3 has two exact matches (for X and Y), solution 1 has one match (just for X). As solution 3 covers solution 1, and solution 2 does not have a promotion or exact match for “Z”, code will compile and solution 3 will be selected

Method overloading

- ▶ Let's consider the following code:

App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call `Math::Add` with the following parameters:

```
Math m; m.Add('a', true, 100)
```

#		X (char)	Y (bool)	Z (int)
1.	Add (<i>char x, int y, int z</i>)	exact match	promotion	exact match
2.	Add (<i>double x, int y, int z</i>)	conversion	promotion	exact match
3.	Add (<i>char x, bool y, char z</i>)	exact match	exact match	promotion

- ▶ Solution 3 matches parameters 1 and 2, solution 1 matches parameters 1 and 3 (there is no clear solution) → **ambiguity case**

Method overloading

- ▶ Let's consider the following code:

App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call `Math::Add` with the following parameters:

```
Math m; m.Add(1.5, true, 'a')
```

#		X (double)	Y (bool)	Z (char)
1.	Add (<i>char x, int y, int z</i>)	conversion	promotion	promotion
2.	Add (<i>double x, int y, int z</i>)	exact match	promotion	promotion
3.	Add (<i>char x, bool y, char z</i>)	conversion	exact match	exact match

- ▶ Solution 3 matches parameters 2 parameters (Y and Z) but not parameter “X”. Since there is also a solution that could match parameter “X” (solution 2) this will be considered an **ambiguity case**.

Method overloading

- ▶ Let's consider the following code:

App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call Math::Add with the following parameters:

```
Math m; m.Add(100, 1.5, 1.5)
```

#		X (int)	Y (double)	Z (double)
1.	Add (<i>char x, int y, int z</i>)	conversion	conversion	conversion
2.	Add (<i>double x, int y, int z</i>)	conversion	conversion	conversion
3.	Add (<i>char x, bool y, char z</i>)	conversion	conversion	conversion

- ▶ No cases with exact match, all solutions have 3 conversions → **ambiguity case**

Method overloading

- ▶ Let's consider the following code:

App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, char z) { return 3; }
};
```

- ▶ and let's consider that we call `Math::Add` with the following parameters:

```
Math m; m.Add(1.0f, 1.5, 1.5)
```

#		X (float)	Y (double)	Z (double)
1.	Add (<i>char x, int y, int z</i>)	conversion	conversion	conversion
2.	Add (<i>double x, int y, int z</i>)	promotion	conversion	conversion
3.	Add (<i>char x, bool y, char z</i>)	conversion	conversion	conversion

- ▶ No cases with exact match, however there is one case that has an accepted promotion while the rest only have conversions. No promotion/exact match for “Y” and “Z” for Solution 1 and 3. Solution 2 is selected and code compiles.

Method overloading

- ▶ Let's change the previous definitions a little bit:

App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, double z) { return 3; }
};
```

- ▶ and let's consider that we call Math::Add with the following parameters:

```
Math m; m.Add(1.0f, 1.5, 1.0f)
```

#		X (float)	Y (double)	Z (float)
1.	Add (<i>char x, int y, int z</i>)	conversion	conversion	conversion
2.	Add (<i>double x, int y, int z</i>)	promotion	conversion	conversion
3.	Add (<i>char x, bool y, double z</i>)	conversion	conversion	promotion

- ▶ Solution 2 is valid for “X” (due to promotion), Solution 3 is valid for “Z” (due to promotion). → **ambiguity case**

Method overloading

- ▶ Let's change the previous definitions a little bit:

App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, double z) { return 3; }
};
```

- ▶ and let's consider that we call Math::Add with the following parameters:

```
Math m; m.Add(1.0f, 1.5, 1.5)
```

#		X (float)	Y (double)	Z (double)
1.	Add (<i>char x, int y, int z</i>)	conversion	conversion	conversion
2.	Add (<i>double x, int y, int z</i>)	promotion	conversion	conversion
3.	Add (<i>char x, bool y, double z</i>)	conversion	conversion	exact match

- ▶ This is a case where promotion and exact match are seen as equals. Solution 2 is valid for “X” (due to promotion), Solution 3 is valid for “Z” (due to exact match). → **ambiguity case**

Method overloading

- ▶ Let's change the previous definitions a little bit:

App.cpp

```
struct Math
{
    int Add(char x, int y, int z) { return 1; }
    int Add(double x, int y, int z) { return 2; }
    int Add(char x, bool y, double z) { return 3; }
};
```

- ▶ and let's consider that we call `Math::Add` with the following parameters:

```
Math m; m.Add(1.0f, 1.5, 100)
```

#		X (float)	Y (double)	Z (int)
1.	Add (<i>char x, int y, int z</i>)	conversion	conversion	exact match
2.	Add (<i>double x, int y, int z</i>)	promotion	conversion	exact match
3.	Add (<i>char x, bool y, double z</i>)	conversion	conversion	conversion

- ▶ Solution 2 covers “X” (due to promotion) and “Z” (due to exact match). There is no other solution better , or one that can cover “Y” → Solution 2 is selected and the code compiles.

Method overloading

- ▶ When dealing with the `const` keyword there are also some differences in terms of method overloading and overload resolution
- ▶ For numerical types (types that are transmitted to a method by value) `const` is ignored from the method / function signature
- ▶ For pointers and references, `const` is used in the method / function signature.

Method overloading

- ▶ This case will NOT compile - but not due to a ambiguity problem, but rather to the fact that both *Inc(int)* and *Inc(const int)* are considered to have the same signature: *Inc(int)*

App.cpp

```
class Math
{
public:
    int Inc(int x) { return x + 2; }
    int Inc(const int x) { return x + 1; }
};
void main()
{
    Math m;
    int x = 10;
    m.Inc(x);
}
```

error C2535: 'int Math::Inc(int)': member function already defined or declared
note: see declaration of 'Math::Inc'

Method overloading

- In this case , the two *Inc* methods are considered to have a different signature and therefore are used in the overload resolution. As “&d” is an *int ** than the best match (meaning *Inc(int *)* will be chosen). The code compiles.

App.cpp

```
class Math
{
public:
    int Inc(int * x)
    {
        return *x + 2;
    }

    int Inc(const int * x)
    {
        return *x + 1;
    }
};

void main()
{
    Math m;
    int x = 10;
    m.Inc(&x);
}
```

Method overloading

- ▶ Similarly, if we change “x” local variable from main function to be a constant, the second function *Inc(const int *)* will be chosen as a perfect match.

App.cpp

```
class Math
{
public:
    int Inc(int * x)
    {
        return *x + 2;
    }

    int Inc(const int * x)
    {
        return *x + 1;
    }
};

void main()
{
    Math m;
    const int x = 10;
    m.Inc(&x);
}
```

Method overloading

- ▶ The same logic applies for references as well.

App.cpp

```
class Math
{
public:
    int Inc(int & x)
    {
        return x + 2;
    }

    int Inc(const int & x)
    {
        return x + 1;
    }
};

void main()
{
    Math m;
    const int x = 10;
    m.Inc(x);
}
```

App.cpp

```
class Math
{
public:
    int Inc(int & x)
    {
        return x + 2;
    }

    int Inc(const int & x)
    {
        return x + 1;
    }
};

void main()
{
    Math m;
    int x = 10;
    m.Inc(x);
}
```

Method overloading

- ▶ The same logic applies for references as well.
- ▶ In particular, when dealing with constant numerical values they will always be translated into a const reference.

App.cpp

```
class Math
{
public:
    int Inc(int & x)
    {
        return x + 2;
    }

    int Inc(const int & x)
    {
        return x + 1;
    }
};

void main()
{
    Math m;
    m.Inc(100);
}
```

Method overloading

- In this case the code will not compile as a constant (const) value CAN NOT be converted to a non-constant value.

App.cpp

```
class Math
{
public:
    int Inc(int & x)
    {
        return x + 2;
    }
};
void main()
{
    Math m;
    m.Inc(100);
}
```

error C2664: 'int Math::Inc(int &)': cannot
convert argument 1 from 'int' to 'int &'

Method overloading

- ▶ The rest of the promotion / conversion rules apply.
- ▶ In this example, ‘a’ is of type **char**. As there is no **Inc** method that receives a char parameter, ‘a’ will be converted to an **int** and then to a **const int &**. This code will compile.

App.cpp

```
class Math
{
public:
    int Inc(int & x)
    {
        return x + 2;
    }

    int Inc(const int & x)
    {
        return x + 1;
    }
};

void main()
{
    Math m;
    m.Inc('a');
}
```

The background features a dark blue gradient with a subtle geometric pattern of lighter blue triangles and lines.

► NULL pointer

NULL pointer

- ▶ Let's consider the following code:

App.cpp

```
void Print(int value)
{
    printf("Number: %d\n", value);
}
void Print(const char* text)
{
    printf("Text: %s\n", text);
}

void main()
{
    Print(10);
    Print("C++ test");

    Print(NULL);
}
```

- ▶ The code compiles correctly. What is the output of this code ?

NULL pointer

- ▶ Let's consider the following code:

App.cpp

```
void Print(int value)
{
    printf("Number: %d\n", value);
}
void Print(const char* text)
{
    printf("Text: %s\n", text);
}

void main()
{
    Print(10);
    Print("C++ test");
    Print(NULL);
}
```

Output

```
Number: 10
Text: C++ test
Number: 0
```

- ▶ Why the last call of Print function is considered to be a number ?

NULL pointer

- ▶ Let's consider the following code:

App.cpp

```
void Print(int value)
{
    printf("Number: %d\n", value);
}
void Print(const char* text)
{
    printf("Text: %s\n", text);
}

void main()
{
    Print(10);
    Print("C++ tutorial");
    Print(NULL);
}
```

```
#ifndef NULL
#ifndef __cplusplus
#define NULL 0
#else
#define NULL ((void *)0)
#endif
#endif
```

- ▶ Why the last call of Print function is considered to be a number ?

NULL pointer

- ▶ So - NULL is defined as a number. While during promotion, value 0 can be translated into a NULL pointer, there are often cases (similar to previous one) where the intended parameter is a pointer (a NULL pointer) and not a number.
- ▶ The solution was to create a new constant (keyword) that refers only to null pointers. This constant is called **nullptr**

App.cpp

```
void Print(int value) { ... }
void Print(const char* text) { ... }

void main()
{
    Print(nullptr);
}
```

- ▶ In the previous example, the compiler will now call “Print(const char*)” function.

NULL pointer

- ▶ The following assignments are valid for NULL constant and all variable will be set to 0, false or a null pointer.

App.cpp

```
void main()
{
    int x = NULL;
    char y = NULL;
    float f = NULL;
    bool b = NULL;
    const char* p = NULL;
    int * i = NULL;
}
```

- ▶ The following assignments are invalid (code will NOT compile):

App.cpp

```
void main()
{
    int x = nullptr;
    char y = nullptr;
    float f = nullptr;
}
```

NULL pointer

- ▶ The following assignments are valid and the code will compile.

App.cpp

```
void main()
{
    bool b = nullptr;
    const char* p = nullptr;
    int * i = nullptr;
}
```

- ▶ Keep in mind that **nullptr** can still be used as a **bool** value (equal to **false**). However, even if this cast is possible, **nullptr** will always chose a pointer to a **bool**. The following example works and does not yield any ambiguity:

App.cpp

```
void Print(bool value) { ... }
void Print(const char* text) { ... }
```

```
void main()
{
    Print( nullptr );
}
```

The compiler will choose to call “Print (const char*)” function

NULL pointer

- ▶ However, the following example will produce an ambiguity and the code will not compile:

App.cpp

```
void Print(bool value) { ... }

void Print(const char* text) { ... }
void Print(int* value) { ... }

void main()
{
    Print( nullptr );
}
```

- ▶ The compiler will yield an error that states that it does not know what to chose for the call of “Print (nullptr)” and that it has two possible variants to chose from.

- 
- The background features a large, abstract graphic on the left side composed of various shades of blue and dark blue triangles and trapezoids, creating a layered, geometric pattern.
- ▶ “const” specifier

Classes (methods) - the "const" specifier

- ▶ Whenever a method is declared within a class, a special keyword can also be used to specify a certain behavior for that method: “**const**”
- ▶ The following code compiles without problem. At the end of execution member x from object “d” will be 1;

App.cpp

```
class Date
{
    private:
        int x;
    public:
        int& GetX();
};

int& Date::GetX()
{
    x = 0;
    return x;
}

void main()
{
    Date d;
    d.GetX()++;
}
```

Classes (methods) - the "const" specifier

- ▶ This code will not compile because GetX function () returns a constant reference to a number. This means that the operator "++" from "d.GetX () ++" has to modify a number that is considered constant.

App.cpp

```
class Date
{
    private:
        int x;
    public:
        const int& GetX();
};

const int& Date::GetX()
{
    x = 0;
    return x;
}

void main()
{
    Date d;
    d.GetX()++;
}
```

Classes (methods) - the "const" specifier

- ▶ The code compiles. Method GetX () returns a reference to a constant integer whose value is 0. In the main function, “x” maintains a copy of the value returned by GetX() function (a copy that can be modified).
- ▶ This is the recommended solution if we want to give **read-only access** to a member variable (in particular if it is NOT a basic type).

App.cpp

```
class Date
{
    private:
        int x;
    public:
        const int& GetX();
};

const int& Date::GetX()
{
    x = 0;
    return x;
}

void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

Classes (methods) - the "const" specifier

- When dealing with pointers or references, “const” specifier can be used in the following ways:

App.cpp

```
void main()
{
    int x;
    const int * ptr;
    ptr = &x;
    *ptr = 1;
}
```

This code will not compile as *ptr* points to a constant int that CAN NOT BE modified.

- In the previous example - the const specifier is part of the value. This means that we can modify the pointer (NOT the value) without any issue.

App.cpp

```
void main()
{
    int x;
    const int * ptr;
    ptr = &x;
    ptr += 1;
}
```

This code will run as we DO NOT modify the actual value, we just modify the pointer.

Classes (methods) - the "const" specifier

- When dealing with pointers or references, “const” specifier can be used in the following ways:

App.cpp

```
void main()
{
    int x;
    int * const ptr;
    ptr = &x;
}
```

This code will not compile as *ptr* is a constant pointer that points towards a non-constant value.

- In the previous example - the const specifier refers to the pointer and NOT the value it points to.

App.cpp

```
void main()
{
    int x;
    int * const ptr = &x;
    *ptr = 1;
}
```

This code will run as it initializes the constant pointer from the beginning.

This code will also run. “ptr” pointer points towards a non-const value that can be modified.

Classes (methods) - the "const" specifier

- When dealing with pointers or references, “const” specifier can be used in the following ways:

App.cpp

```
void main()
{
    int x;
    int * const ptr;
    ptr = &x;
}
```

This code will not compile as *ptr* is a constant pointer that points towards a non-constant value.

- In the previous example - the const specifier refers to the pointer and NOT the value it points to.

App.cpp

```
void main()
{
    int x;
    int * const ptr = &x;
    ptr += 1;
}
```

This code will run as it initializes the constant pointer from the beginning.

This code will NOT run as we try to modify a constant pointer.

Classes (methods) - the "const" specifier

- When dealing with pointers or references, “const” specifier can be used in the following ways:

App.cpp

```
void main()
{
    int x;
    const int * const ptr = &x;
    *ptr = 1;
    ptr += 1;
}
```

In this case both the pointer and the value it points to are constant. The code will not compile - one can not modify the pointer or the value.

Classes (methods) - the "const" specifier

"const" specifier respects the **Clockwise/Spiral Rule** for C language.

C/C++ expression	Explanation	Change value	Change Pointer
int * ptr;	Non-const pointer to a non-const value	YES	YES
const int * ptr;	Non-const pointer to a const value	NO	YES
int const * ptr;	Non-const pointer to a const value	NO	YES
int * const ptr;	Const pointer to a non-const value	YES	NO
const int * const ptr;	Const pointer to a const value	NO	NO

In particular, a syntax like "*int * const ptr*" is equivalent to a reference (*int &*) and "*const int * const ptr*" to "*const int &*"

Classes (methods) - the "const" specifier

“const” specifier respects the **Clockwise/Spiral Rule** for C language.

C/C++ expression	Explanation
int ** ptr;	Non-const pointer to a non-const pointer to a non-const value
const int ** ptr;	Non-const pointer to a non-const pointer to a const value
int ** const ptr;	Const pointer to a non-const pointer to a non-const value
int * const * const ptr;	Const pointer to a const-pointer to a non-const value
const int * const * const ptr;	Const pointer to a const-pointer to a const value

Classes (methods) - the "const" specifier

- ▶ This code will not compile. The usage of “const” keyword at the end of the method declaration specifies that within that method data members of that class **can not** be modified. In the next example, “x” is a data member from class Data and assigning value 0 to it contradicts the “const” keyword from method definition.

App.cpp

```
class Date
{
    private:
        int x;
    public:
        const int& GetX() const;
};

const int& Date::GetX() const
{
    x = 0;
    return x;
}

void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

error C3490: 'x' cannot be modified because
it is being accessed through a const object

Classes (methods) - the "const" specifier

- ▶ Let's assume that we have the following code:

App.cpp

```
class Date
{
private:
    int x;
    int y,z,t;
public:
    const int& GetX() const;
};
const int& Date::GetX() const
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

- ▶ And we want to make sure that access to data members “y”, “z” and “t” are ready only, but for data member “x” we have read/write access.
- ▶ If we use a “const” function (as define in this example) ➔ “x” will be read-only as well.

Classes (methods) - the "const" specifier

- ▶ Let's assume that we have the following code:

App.cpp

```
class Date
{
private:
    mutable int x;
    int y,z,t;
public:
    const int& GetX() const;
};
const int& Date::GetX() const
{
    x = 0;
    return x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

- ▶ Starting with C++11 there is a new specifier called “**mutable**” that allows write access to a data member even if “**const**” specifier is used.
- ▶ This code will compile.

Classes (methods) - the "const" specifier

- ▶ Let's assume that we have the following code:

App.cpp

```
class Date
{
private:
    const mutable int * x;
    int y,z,t;
public:
    const int& GetX() const;
};
const int& Date::GetX() const
{
    x = &y;
    return *x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

- ▶ “*const*” can be used with “*mutable*”. In the previous example *mutable* refers to the value of the pointer and does not interfere with the *const* qualifier. This translates that you can modify the pointer (through the *mutable* qualifier) but you can not modify the value (due to the *const* qualifier at the end of the *GetX()* method).

Classes (methods) - the "const" specifier

- ▶ Let's assume that we have the following code:

App.cpp

```
class Date
{
private:
    const mutable int * const x;
    int y,z,t;
public:
    const int& GetX() const;
};
const int& Date::GetX() const
{
    x = &y;
    return *x;
}
void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

- ▶ This code will not compile as “x” being a const pointer (not a pointer to a const value) can not be mutable at the same time (it will imply that it can be changed). At the same time, “x=&y” can not run as “x” is a const pointer.

Classes (methods) - the "const" specifier

Usually *mutable* specifier is used when:

- ▶ A class is run in a multi-threaded environment and you need a variable that can be used between multiple threads
- ▶ Lambda expressions
- ▶ As a way to control what data members can be modified within a class from a const method.

Classes (methods) - the "const" specifier

- ▶ The code compiles correctly because "x" is no longer a member of an instance but a global static member (it does not belong to the object).

App.cpp

```
class Date
{
    private:
        static int x;
    public:
        const int& GetX() const;
};

int Date::x = 100;
const int& Date::GetX() const
{
    x = 0;
    return x;
}

void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

Classes (methods) - the "const" specifier

- ▶ The code does not compile because the “const” modifier from the end of GetX declaration can not be used for static functions as it needs and instance to apply to (access to **this** pointer that is impossible if the method is declared as **const**)

App.cpp

```
class Date
{
    private:
        static int x;
    public:
        static const int& GetX() const;
};

int Date::x = 100;
static const int& Date::GetX() const
{
    x = 0;
    return x;
}

void main()
{
    Date d;
    int x = d.GetX();
    x++;
}
```

Classes (methods) - the "const" specifier

- ▶ This code compiles. “const” specifier refers to the current object-instance alone. It does not apply to another instance of a different type (it will only apply to the instance represented by “this”).

App.cpp

```
class Date
{
private:
    int x;
public:
    void ModifyX(Date * d) const
    {
        d->x = 0;
    }
};
void main()
{
    Date d1,d2;
    d1.ModifyX(&d2);
}
```

Classes (methods) - the "const" specifier

- ▶ This code will NOT compile as a “const” method refers to the current instance (“this” pointer).

App.cpp

```
class Date
{
private:
    int x;
public:
    void ModifyX(Date * d) const
    {
        this->x = 0;
    }
};
void main()
{
    Date d1,d2;
    d1.ModifyX(&d2);
}
```

Classes (methods) - the "const" specifier

- ▶ “const” is part of object type
- ▶ A class method/function can not modify a parameters if it is defined as “const”

Without const specifier

```
class Date
{
private:
    int x;
public:
    void Inc();
};

void Date::Inc()
{
    x++;
}

void Increment(Date &d)
{
    d.Inc();
}

void main()
{
    Date d;
    Increment(d);
}
```

With const specifier

```
class Date
{
private:
    int x;
public:
    void Inc();
};

void Date::Inc()
{
    x++;
}

void Increment(const Date &d)
{
    d.Inc();
}

void main()
{
    Date d;
    Increment(d);
}
```

Compile error, d is const

Classes (data members) - the "const" specifier

- ▶ “const” can be used for data members as well. The following code will not compile as the const value is not initialized.

App.cpp

```
class Data
{
    const int x;
public:
    int GetX() { return x; }
};
void main()
{
    Data d;
}
```

- ▶ To instantiate such a code , a value has to be added in to the const data member in the class definition (more on this topic in the course related to constructors).

App.cpp

```
class Data
{
    const int x = 10;
public:
    int GetX() { return x; }
};
```

- 
- The background features a large, abstract graphic on the left side composed of various shades of blue and dark blue triangles and trapezoids, creating a layered, geometric pattern.
- ▶ “friend” specifier

“friend” specifier

- ▶ For a class a “friend” function is a function that can access methods and data members that with private modifier define within that class.
- ▶ A “friend” function does not belong to the class (in this case to the Date class). From this point of view access specifier is irrelevant (it doesn't matter if the “friend” function is written in the private or the public section)

App.cpp

```
class Date
{
    int x;
public:
    Date(int value) : x(value) {}
    void friend PrintDate(Date &d);
};

void PrintDate(Date &d)
{
    printf("X = %d\n", d.x);
}

void main()
{
    Date d1(1);
    PrintDate(d1);
}
```

“friend” specifier

App.cpp

```
class Date
{
    int x;
public:
    Date(int value) : x(value) {}
    friend class Printer;
};

class Printer
{
public:
    void PrintDecimal(Date &d);
    void PrintHexazecimal(Date &d);
};

void Printer::PrintDecimal(Date &d)
{
    printf("x = %d\n", d.x);
}

void Printer::PrintHexazecimal(Date &d)
{
    printf("x = %x\n", d.x);
}

void main()
{
    Date d1(123);
    Printer p;
    p.PrintDecimal(d1);
    p.PrintHexazecimal(d1);
}
```

- ▶ “friend” specifier can be applied to an entire class
- ▶ In this case , all methods from the “friend” class can access the members from the original class (e.g. all methods from class Printer can access the private data from class Data).

“friend” specifier

App.cpp

```
class Data;
class Modifier
{
public:
    void SetX(Data & d, int value);
};

class Data
{
    int x;
    int& GetXRef() { return x; }
public:
    int GetX() { return x; }
    friend void Modifier::SetX(Data &, int);
};

void Modifier::SetX(Data & d, int value)
{
    d.GetXRef() = value;
}

void main()
{
    Data d;
    Modifier m;
    m.SetX(d, 10);
    printf("%d\n", d.GetX());
}
```

- ▶ A method from a class can also be declared as friend for a class.
- ▶ The declaration must include the exact method signature and the return type.
- ▶ In this case, method *SetX(Data& , int)* from class **Modifier** can access private data from class **Data**.

Q & A

OOP

Gavrilut Dragos
Course 3

Summary

- ▶ Initialization lists (recap)
- ▶ Constructors
- ▶ Const & Reference data members
- ▶ Delegating constructor
- ▶ Initialization lists for classes
- ▶ Value Types
- ▶ Copy & Move Constructors
- ▶ Constrains

The background features a large, abstract graphic on the left side composed of overlapping blue triangles of varying shades of blue. It has a subtle, organic, fan-like or leaf-like pattern.

Initialization lists

- ▶ (recap)

Initialization lists

- ▶ “{“ and “}” can now be used to initialize values.
This method is called: “Initialization lists”

App.cpp

```
void main()
{
    int x = 5;
    int y = { 5 };
    int z = int { 5 };
}
```

- ▶ In all of these cases “x”, “y” and “z” will have a value of 5.

Assembly code generated

```
mov     dword ptr [x],5
mov     dword ptr [y],5
mov     dword ptr [z],5
```

Initialization lists

- ▶ “{“ and “}” can be used for array initialization as well:

App.cpp

```
void main()
{
    int x[3] = { 1, 2, 3 };
    int y[] = { 4, 5, 6 };
    int z[10] = {};
    int t[10] = { 1, 2 };
    int u[10] = { 15 };
    int v[] = { 100 };
}
```

Variable	Values
X [3]	[1, 2, 3]
Y [3]	[4, 5, 6]
Z [10]	[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
T [10]	[1, 2, 0, 0, 0, 0, 0, 0, 0, 0]
U [10]	[15, 0, 0, 0, 0, 0, 0, 0, 0, 0]
V [1]	[100]

- ▶ If possible, the compiler tries to deduce the size of the array from the declaration. If the initialization list is too small, the rest of the array will be filled with the default value for that type (in case of “int” with value 0 → values that are grayed in the table).

Initialization lists

- ▶ “{“ and “}” can be used to initialize a matrix as well.

App.cpp

```
void main()
{
    int x[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
    int y[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
}
```

- ▶ However, only the first dimension of the matrix can be left unknown. The following code will not compile as the compiler can not deduce the size of the matrix.

App.cpp

```
void main()
{
    int x[][] = { { 1, 2, 3 }, { 4, 5, 6 } };
}
```

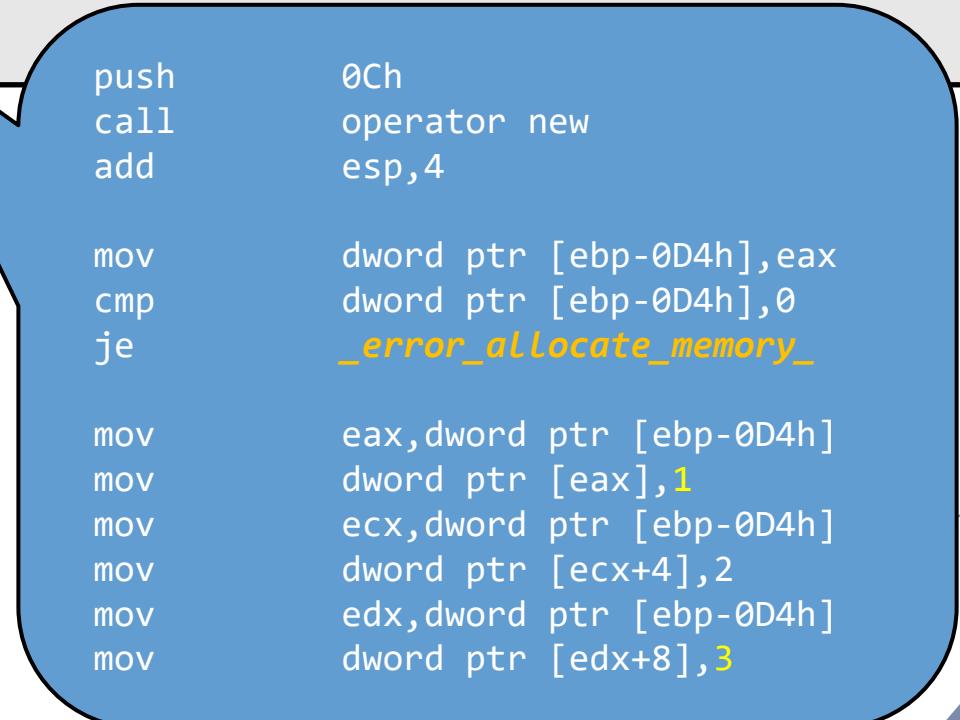
error C2087: 'x': missing subscript
error C2078: too many initializers

Initialization lists

- ▶ Initialization lists can also be used when creating a pointer:

App.cpp

```
void main()
{
    int *x = new int[3] {1, 2, 3};
```



The callout bubble points from the initialization list in the C++ code to the corresponding assembly instructions.

push	0Ch
call	operator new
add	esp,4
mov	dword ptr [ebp-0D4h],eax
cmp	dword ptr [ebp-0D4h],0
je	<i>_error_allocate_memory_</i>
mov	eax,dword ptr [ebp-0D4h]
mov	dword ptr [eax],1
mov	ecx,dword ptr [ebp-0D4h]
mov	dword ptr [ecx+4],2
mov	edx,dword ptr [ebp-0D4h]
mov	dword ptr [edx+8],3

A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented diagonally, creating a sense of depth and movement.

► Constructors

Constructors

- ▶ A constructor is a type-less function that is called whenever a class is created.
- ▶ A class may contain multiple constructors (with different initialization parameters)
- ▶ A class does not need to have a constructor. However, if it has at least one, then its initialization should be based on that constructor parameters.
- ▶ If one class contains several member data that have their own constructors, those constructors will be called in the same order of their declaration.
- ▶ A constructor **can not be static or constant**
- ▶ A class that contains at least a “*const*” data member or a data member that is a reference must have a constructor where these data members are initialized.
- ▶ A constructor without any parameters is often called the default constructor.
- ▶ A constructor may have an access modifier (public, private, protected).

Constructors

- ▶ The **constructor** is defined as a function *with the same name as the class* and *no return value*
- ▶ A class :

- May have NO constructors
- May have only one constructor
- May have multiple constructors
- May have constructors that are not *public*

```
App.cpp
class MyClass
{
    int x;
public:
};
```

```
App.cpp
class MyClass
{
    int x;
public:
    MyClass ();
};
```

```
App.cpp
class MyClass
{
    int x;
public:
    MyClass ();
    MyClass (int value);
    MyClass (float value);
};
```

```
App.cpp
class MyClass
{
    int x;
private:
    MyClass ();
};
```

Type of constructors

Constructors:

- ▶ Default constructor (without any parameters)
- ▶ Copy constructor
- ▶ Move constructor

App.cpp

```
class MyClass
{
    int x;
public:
    → MyClass ();
    → MyClass (const MyClass & objToCopyFrom);
    → MyClass (const MyClass && objToMoveFrom);
};
```

A class can have none, one , some or all of these types of constructors.

Constructors

- ▶ A constructor is called whenever an object of that class is created (this means local object - create on local stack, heap allocated objects or global variables).
- ▶ If we define an array of object, then the constructor will be called for every object in this array.
- ▶ However, if we create a pointer to a specific object, the constructor (if any) will not be called.

App.cpp

```
class Date
{
    ...
}
void main()
{
    Date d;                                // constructor is called
    Date *d2 = new Date();                  // constructor is called
    Date arr[100];                         // constructor is called 100 times
    Date *d3;                               // un-initialized pointer - the constructor will not be called
}
```

Constructors

- ▶ This code will compile correctly and produce the following output:

App.cpp

```
class MyClass
{
public:
    MyClass(const char * text) { printf("Ctor for: %s\n", text); }

    MyClass global("global variable");

    void main()
    {
        printf("Entering main function \n");

        MyClass local("local variable");

        MyClass * m = new MyClass("Heap variable");
    }
}
```

Ctor for: global variable
Entering main function
Ctor for: local variable
Ctor for: Heap variable

Constructors

- ▶ This code will compile correctly and produce the following output:

App.cpp

```
class MyClass
{
public:
    MyClass(const char * text) { printf("C\n"); }
};

MyClass global("global variable");

void main()
{
    printf("Entering main function \n");

    MyClass local("local variable");

    MyClass * m = new MyClass("Heap variable");
}
```

Global variables are instantiated before the main function is called. In this case since the global variable has a constructor, that constructor is called before the main function is called.

Constructors

- ▶ This code will compile correctly and produce the following assembly output:

App.cpp

```
class MyClass
{
public:
    MyClass(const char * text) { printf("Ctor for: %s\n",
};

MyClass global("global variable");

void main()
{
    printf("Entering main function \n");
    MyClass local("local variable");
    MyClass * m = new MyClass("Heap variable");
}
```

```
push    offset string "Entering main function \n"
call   _printf
add    esp,4

push    offset string "local variable"
lea     ecx,[local]
call   MyClass::MyClass

push    1
call   operator new
add    esp,4
mov    dword ptr [ebp-4Ch],eax
cmp    dword ptr [ebp-4Ch],0
je    null_Asignament
push    offset string "Heap variable"
mov    ecx,dword ptr [ebp-4Ch]
call   MyClass::MyClass
mov    dword ptr [ebp-50h],eax
jmp    asign_from_temp_to_m
null_Asignament:
    mov    dword ptr [ebp-50h],0
asign_from_temp_to_m:
    mov    eax,dword ptr [ebp-50h]
    mov    dword ptr [m],eax
```

Constructors

- ▶ This code will compile correctly and produce the following assembly output.

App.cpp

```
class MyClass
{
public:
    MyClass(const char * text) { printf("Ctor for: %s\n",
};

MyClass global("global variable");

void main()
{
    printf("Entering main function \n");

    MyClass local("local variable");

    MyClass * m = new MyClass("Heap variable");
}
```

```
push      offset string "Entering main function \n"
call      _printf
add      esp,4

push      offset string "local variable"
lea       ecx,[local]
call      MyClass::MyClass

push      1
call      operator new
add      esp,4
mov      dword ptr [ebp-4Ch],eax
cmp      dword ptr [ebp-4Ch],0
je       null_Asignament
push      offset string "Heap variable"
mov      ecx,dword ptr [ebp-4Ch]
call      MyClass::MyClass
mov      dword ptr [ebp-50h],eax
jmp      asign_from_temp_to_m
null_Asignament:
    mov      dword ptr [ebp-50h],0
asign_from_temp_to_m:
    mov      eax,dword ptr [ebp-50h]
    mov      dword ptr [m],eax
```

Constructors

- ▶ This code will compile correctly and produce the following assembly output.

App.cpp

```
class MyClass
{
public:
    MyClass(const char * text) { printf("Ctor for: %s\n",
};

MyClass global("global variable");

void main()
{
    printf("Entering main function \n");

    MyClass local("local variable");

    MyClass * m = new MyClass("Heap variable");
}
```

```
push          offset string "Entering main function \n"
call         _printf
add          esp,4

push          offset string "local variable"
lea           ecx,[local]
call         MyClass::MyClass

push          1
call         operator new
add          esp,4
mov          dword ptr [ebp-4Ch],eax
cmp          dword ptr [ebp-4Ch],0
je           null_Asignament
push          offset string "Heap variable"
mov           ecx,dword ptr [ebp-4Ch]
call         MyClass::MyClass
mov          dword ptr [ebp-50h],eax
jmp           asign_from_temp_to_m
null_Asignament:
    mov          dword ptr [ebp-50h],0
asign_from_temp_to_m:
    mov          eax,dword ptr [ebp-50h]
    mov          dword ptr [m],eax
```

Constructors

- ▶ This code will compile correctly and produce valid assembly output.

App.cpp

```
class MyClass
{
public:
    MyClass(const char * text) { printf("Ctor for: %s\n", text); }

    MyClass global("global variable");

    void main()
    {
        printf("Entering main function \n");

        MyClass local("local variable");

        MyClass * m = new MyClass("Heap variable");
    }
}
```

push offset string "Entering main function \n"
call _printf
add esp,4

push offset string "local variable"
lea ecx,[local]
call MyClass::MyClass

push 1
call operator new
add esp,4
mov dword ptr [ebp-4Ch],eax
cmp dword ptr [ebp-4Ch],0
je null_Asignament
push offset string "Heap variable"
mov ecx,dword ptr [ebp-4Ch]
call MyClass::MyClass
mov dword ptr [ebp-50h],eax
jmp asign_from_temp_to_m

null_Asignament:

mov dword ptr [ebp-50h],0

asign_from_temp_to_m:

mov eax,dword ptr [ebp-50h]
mov dword ptr [m],eax

sizeof (MyClass)

Has **new** returned nullptr ?

Constructors

- ▶ In this case the default constructor is called and value of *d.x* is set to 10.

App.cpp

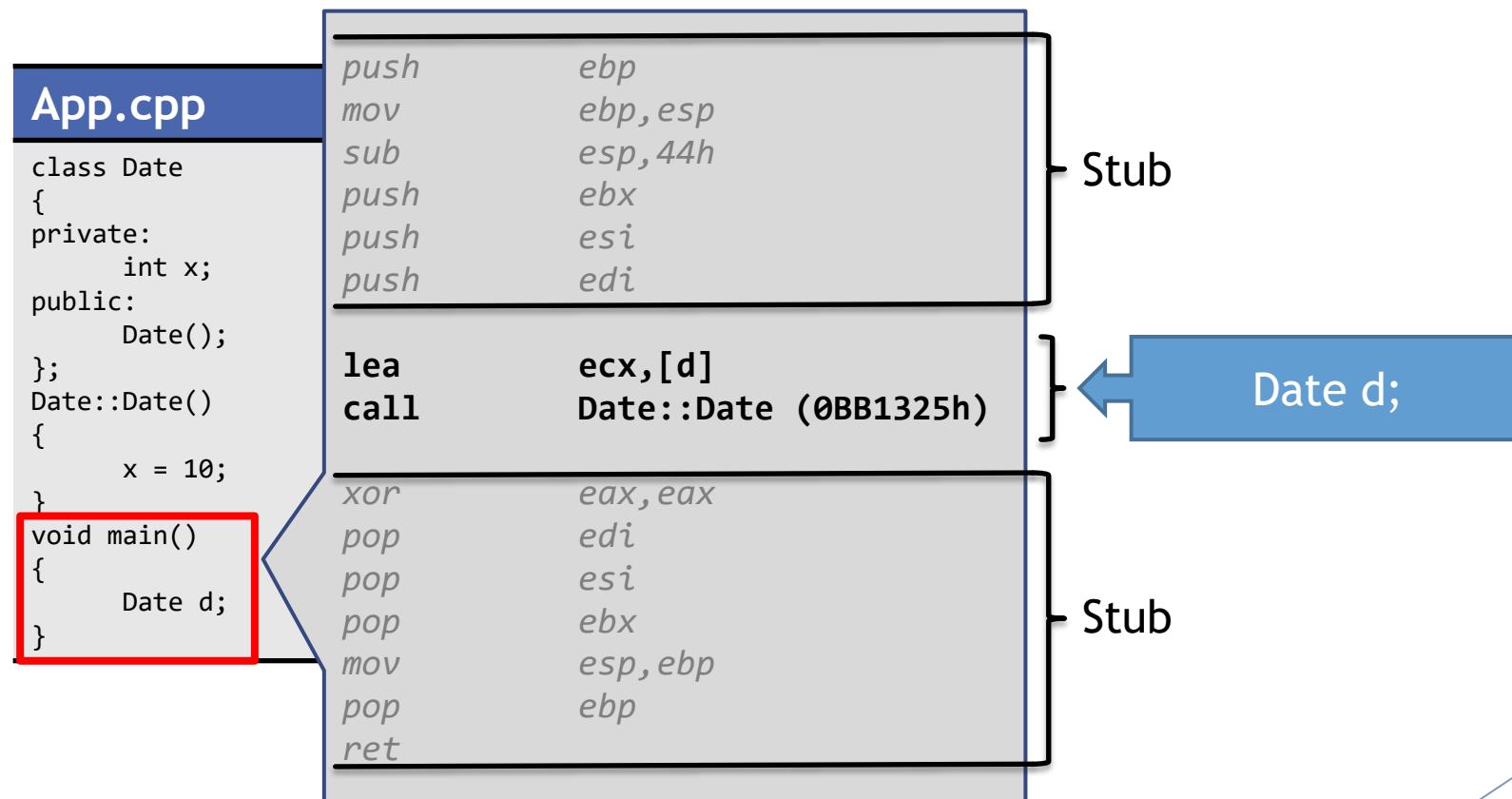
```
class Date
{
private:
    int x;
public:
    Date();
};

Date::Date()
{
    x = 10;
}

void main()
{
    Date d;
}
```

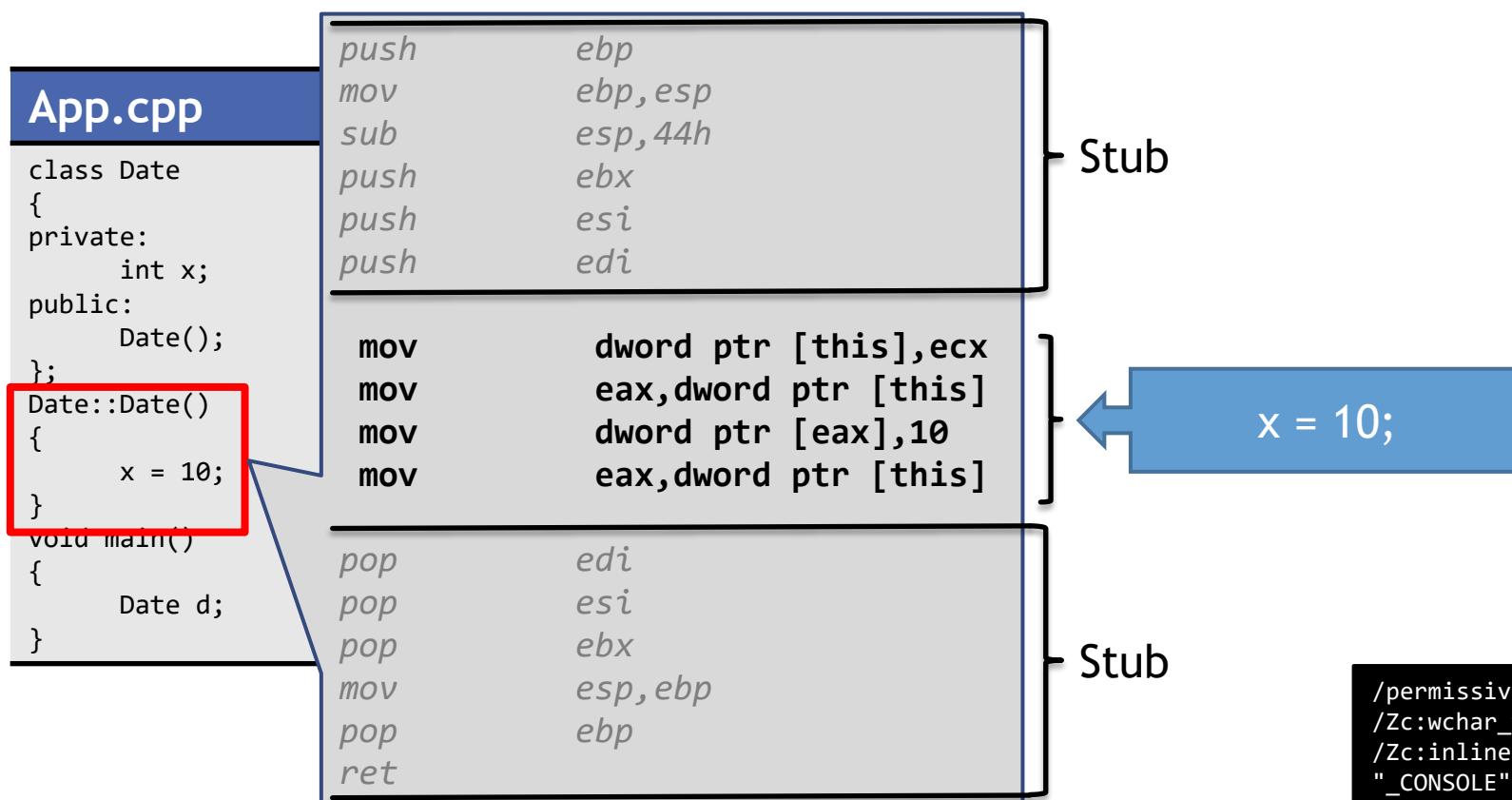
Constructors

- In this case the default constructor is called and value of *d.x* is set to 10.



Constructors

- In this case the default constructor is called and value of *d.x* is set to 10.



```
/permissive- /Yu"pch.h" /GS- /analyze- /W3
/Zc:wchar_t /ZI /Gm- /Od /sdl /Fd"Debug\vc141.pdb"
/Zc:inline /fp:precise /D "WIN32" /D "_DEBUG" /D
"_CONSOLE" /D "_UNICODE" /D "UNICODE"
/errorReport:prompt /WX- /Zc:forScope /RTCu
/arch:IA32 /Gd /Oy- /MDd /FC /Fa"Debug\" /nologo
/Fo"Debug\" /Fp"Debug\ConsoleApplication7.pch"
/diagnostics:classic
```

Constructors

- ▶ “d” is build using the default constructor (`d.x = 10`)
- ▶ “d2” is build using the constructor with one parameter (`d2.x = 100`)

App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
    Date(int value);
};
Date::Date()
{
    x = 10;
}
Date::Date(int value)
{
    x = value;
}
void main()
{
    Date d;
    Date d2(100);
}
```

Constructors

- ▶ Every member data defined in a class can be automatically instantiated in every constructor if we add some parameters after its name (like in the example below).

App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
};

Date::Date() : x(100)

void main()
{
    Date d;
}
```

Constructors

- ▶ If the data member type is another class, the constructor of that class can be called in a similar way.

App.cpp

```
class MyClass
{
    int x;
public:
    MyClass(int value) { this->x = value; }
};

class Date
{
    MyClass m;
public:
    Date(): m(100) { }
};

void main()
{
    Date d;
}
```



The code illustrates the construction of objects of derived classes. In the `Date` class, the constructor `Date(): m(100) { }` calls the constructor of the base class `MyClass` with the argument `100`. This is indicated by a red box around the constructor call in `Date` and another red box around the constructor definition in `MyClass`, with an arrow pointing from the call to the definition.

Constructors

- ▶ If a class does not have a *default constructor* (a constructor without any parameters) but has at least another constructor, another class that has a data member of the same type **HAS** to:
 - A) Explicitly call that constructor in all of its defined constructors

```
App.cpp
class MyClass
{
    int x;
public:
    MyClass(int value) { this->x = value; }

class Date
{
    MyClass m;
public:
    Date() { }
};

void main()
{
    Date d;
}
```

error C2512: 'MyClass': no appropriate default constructor available

- ▶ This code will NOT compile !!!

Constructors

- ▶ If a class does not have a *default constructor* (a constructor without any parameters) but has at least another constructor, another class that has a data member of the same type **HAS** to:
 - A) Explicitly call that constructor in all of its defined constructors

App.cpp

```
class MyClass
{
    int x;
public:
    MyClass(int value) { this->x = value; }

class Date
{
    MyClass m;
public:
    Date() : m(123) { }
};

void main()
{
    Date d;
}
```

- ▶ This code will compile properly

Constructors

- ▶ If a class does not have a *default constructor* (a constructor without any parameters) but has at least another constructor, another class that has a data member of the same type **HAS** to:
 - A) Explicitly call that constructor in all of its defined constructors

App.cpp

```
class MyClass
{
    int x;
public:
    MyClass(int value) { this->x = value; }

class Date
{
    MyClass m;
public:
    Date() : m(123) { }
    Date(int value) { }
};

void main()
{
    Date d;
}
```

error C2512: 'MyClass': no appropriate default constructor available

- ▶ This code will NOT compile. There is at least one constructor that does not instantiate data member “*m*” from class Date.

Constructors

- ▶ If a class does not have a *default constructor* (a constructor without any parameters) but has at least another constructor, another class that has a data member of the same type **HAS** to:
 - A) Explicitly call that constructor in all of its defined constructors

App.cpp

```
class MyClass
{
    int x;
public:
    MyClass(int value) { this->x = value; }

class Date
{
    MyClass m;
public:
    Date() : m(123) { }
    Date(int value) : m(value+10) { }
};

void main()
{
    Date d;
}
```

- ▶ Now the code compiles correctly

Constructors

- ▶ If a class does not have a *default constructor* (a constructor without any parameters) but has at least another constructor, another class that has a data member of the same type **HAS** to:

B) Add a default constructor

App.cpp

```
class MyClass
{
    int x;
public:
    MyClass(int value) { this->x = value; } ←
    MyClass() { this->x = 0; } ←
};

class Date
{
    MyClass m;
public:
    Date() { }
    Date(int value) : m(value+10) { }
};
```

- ▶ This code compiles correctly.

Constructors

- ▶ If a class does not have a *default constructor* (a constructor without any parameters) but has at least another constructor, another class that has a data member of the same type **HAS** to:

C) Remove all constructors

App.cpp

```
class MyClass
{
    int x;
public:
};

class Date
{
    MyClass m;
public:
    Date() { }
    Date(int value) { }
};
```

- ▶ This code compiles correctly.

Constructors

- If a class does not have a *default constructor* (a constructor without any parameters) but has at least another constructor, another class that has a data member of the same type **HAS** to:

D) Use initialization lists

App.cpp

```
class MyClass
{
    int x;
public:
    MyClass(int value) { this->x = value; }
    MyClass() { this->x = 0; }
};

class Date
{
    MyClass m = { 123 };
public:
    Date() { }
    Date(int value) { }
};
```

App.cpp

```
class MyClass
{
    int x;
public:
    MyClass(int value) { this->x = value; }
    MyClass() { this->x = 0; }
};

class Date
{
    MyClass m = 123;
public:
    Date() { }
    Date(int value) { }
};
```

OR

- This code compiles correctly. In this case a call to a constructor is no longer needed as the variable is instantiated with an *initialization list*.

Constructors

- ▶ This code will not compile because data members *obj.t* , *obj.c* and *obj.a* need a custom call to their own constructor.

App.cpp

```
class Tree {  
public:  
    Tree(const char * name) { printf("Tree: %s\n", name); }  
};  
class Car {  
public:  
    Car(const char * name) { printf("Car: %s\n", name); }  
};  
class Animal {  
public:  
    Animal(const char * name) { printf("Animal: %s\n", name); }  
};  
class Object  
{  
    Tree t;  
    Car c;  
    Animal a;  
public:  
};  
void main()  
{  
    Object obj;  
}
```

error C2280: 'Object::Object(void)': attempting to reference a deleted function
note: compiler has generated 'Object::Object' here
note: 'Object::Object(void)': function was implicitly deleted because a data member 'Object::a' has either no appropriate default constructor or overload resolution was ambiguous
note: see declaration of 'Object::a'

Constructors

- Now this code compiles. The constructors for Tree, Car and Animal are called from in the order of their definition in Object (Tree is first, Car is second and Animal is third).

App.cpp

```
class Tree {  
public:  
    Tree(const char * name) { printf("Tree: %s\n", name); }  
};  
class Car {  
public:  
    Car(const char * name) { printf("Car: %s\n", name); }  
};  
class Animal {  
public:  
    Animal(const char * name) { printf("Animal: %s\n", name); }  
};  
class Object  
{  
    Tree t;  
    Car c;  
    Animal a;  
public:  
    Object(): t("oak"), a("fox"), c("Toyota") {}  
};  
void main()  
{  
    Object obj;  
}
```

Tree t → is the first data member from Object

Car c → is the second data member from Object

Output:
Tree: oak
Car: Toyota
Animal: fox

Notice that the calling
order in the constructor
is different (tree, animal
and car)

Constructors

- ▶ This code compiles. If no constructor is present and all data members have either no constructors or a default constructor, the compiler will generate a default constructor that will call the default constructor from that class.

App.cpp

```
class Tree {  
public:  
    Tree() { printf("CTOR: Tree\n"); }  
};  
class Car {  
public:  
    Car() { printf("CTOR: Car\n"); }  
};  
class Animal {  
public:  
    Animal() { printf("CTOR: Animal\n"); }  
};  
class Object  
{  
    Tree t1,t2;  
    Car c;  
    Animal a;  
};  
void main()  
{  
    Object obj;  
}
```

Output:
CTOR: Tree
CTOR: Tree
CTOR: Car
CTOR: Animal

Constructors

- ▶ This code compiles. “x” , “y” and “z” are initialized in the order of their definition. It is important to keep this in mind when you call the constructor with default values for “x” , “y” and “z”

App.cpp

```
class Object
{
    int x, y, z;
public:
    Object(int value) : x(value), y(x*x), z(value*y) {}
};

void main()
{
    Object o(10);
}
```

- ▶ As a result:
 - $o.x = 10$ (the first one to be computed)
 - $o.y = o.x * o.x = 10 * 10 = 100$ (the second one to be computed)
 - $o.z = 10 * o.y = 10 * 100 = 1000$ (the third one to be computed)

Constructors

- ▶ This code compiles. “x” , “y” and “z” are initialized in the order of their definition. However, the results is **inconsistent** as “x” is the first one to be computed !!!

App.cpp

```
class Object
{
    int x, y, z;
public:
    Object(int value) : y(value), z(value/2), x(y*z) {}
};

void main()
{
    Object o(10);
}
```

- ▶ As a result:
 - $o.x = o.y * o.z = \text{unknown results}$ (it depends on the values that resides on the stack when the instance “o” is created). (the first one to be computed !)
 - $o.y = 10$ (the second one to be computed)
 - $o.z = 10 (\text{value}) / 2 = 10 / 2 = 5$ (the third one to be computed)

The background features a large, abstract graphic on the left side composed of overlapping blue and dark blue geometric shapes, including triangles and trapezoids, creating a layered effect.

Const & Reference Data members

Const & Reference data members

- ▶ This code will not compile because class Date has a const member (y) (exempla (A)) or a reference (example (B)) that should be initialized.

note: 'Date::Date(void)': function was implicitly deleted because 'Date' has an uninitialized const-qualified data member 'Date::y'

App.cpp (A)

```
class Date
{
private:
    int x;
    const int y;
public:

};

void main()
{
    Date d;
}
```

App.cpp (B)

```
class Date
{
private:
    int x;
    int & y;
public:

};

void main()
{
    Date d;
}
```

error C2280: 'Date::Date(void)': attempting to reference a deleted function
note: compiler has generated 'Date::Date' here
note: 'Date::Date(void)': function was implicitly deleted because 'Date' has an uninitialized data member 'Date::y' of reference type
note: see declaration of 'Date::y'

Const & Reference data members

- ▶ The code will not compile. While the class Data has a public constructor it does not initialize the value of **y** (a *const member* in example (A) and a *reference* in example (B)).

App.cpp (A)

```
class Date
{
private:
    int x;
    const int y;
public:
    Date();
};
Date::Date() : x(100)
{
}

void main()
{
    Date d;
}
```

App.cpp (B)

```
class Date
{
private:
    int x;
    int & y;
public:
    Date();
};
Date::Date() : x(100)
{
}

void main()
{
    Date d;
}
```

Const & Reference data members

- ▶ The code compiles - y is initialized with value 123 in example (A) and with a reference to data member “X” in example (B)

App.cpp (A)

```
class Date
{
private:
    int x;
    const int y;
public:
    Date();
};
Date::Date() : x(100), y(123)
{
}

void main()
{
    Date d;
}
```

App.cpp (B)

```
class Date
{
private:
    int x;
    int & y;
public:
    Date();
};
Date::Date() : x(100), y(x)
{
}

void main()
{
    Date d;
}
```

Const & Reference data members

- ▶ This code will not compile
- ▶ Every const data member or reference data member defined within a class has to be initialized in every constructor defined in that class.

App.cpp (A)

```
class Date
{
private:
    int x;
    const int y;
public:
    Date();
    Date(int value);
};
Date::Date() : x(100), y(123)
{
}
Date::Date(int value) : x(value)
{}
```

App.cpp (B)

```
class Date
{
private:
    int x;
    int & y;
public:
    Date();
    Date(int value);
};
Date::Date() : x(100), y(x)
{
}
Date::Date(int value) : x(value)
{}
```

error C2789: 'Date::y': an object
of const-qualified type must be
initialized

error C2530: 'Date::y': references
must be initialized

Const & Reference data members

- ▶ This code compiles and runs correctly. One observation here is that a constant value (data member) can be initialized with a non-constant value (in this example with `value*value`) - see example (A)

App.cpp (A)

```
class Date
{
private:
    int x;
    const int y;
public:
    Date();
    Date(int value);
};
Date::Date() : x(100), y(123)
{
}
Date::Date(int value) : x(value), y(value*value)
{
}
void main()
{
    Date d;
    Date d2(100);
}
```

App.cpp (B)

```
class Date
{
private:
    int x;
    int & y;
public:
    Date();
    Date(int value);
};
Date::Date() : x(100), y(x)
{
}
Date::Date(int value) : x(value), y(value)
{
}
void main()
{
    Date d;
    Date d2(100);
}
```

Const & Reference data members

- ▶ This code will not compile. A constant or reference defined within a constructor must be initialized using either an initialization list or the current class constructor definition.

App.cpp

```
class Date
{
private:
    int x;
    const int y;
public:
    Date();
};
Date::Date() : x(100)
{
    y = 123;
}
void main()
{
    Date d;
}
```

App.cpp

```
class Date
{
private:
    int x;
    int & y;
public:
    Date();
};
Date::Date() : x(100)
{
    y = x;
}
void main()
{
    Date d;
}
```

error C2789: 'Date::y': an object
of const-qualified type must be
initialized

error C2530: 'Date::y': references
must be initialized

Const & Reference data members

- ▶ This code compiles correctly. Data member “y” is initialized directly in the definition of the class. This way of initializing data members (either constant or references) is available starting with C++11 standard.

App.cpp

```
class Date
{
private:
    int x;
    const int y = 123;
public:
    Date();
    Date(int value);
};
Date::Date() : x(100)
{
}
Date::Date(int value) : x(value), y(value*value)
{
}
void main()
{
    Date d;
    Date d2(100);
}
```

App.cpp

```
class Date
{
private:
    int x;
    int & y = x;
public:
    Date();
    Date(int value);
};
Date::Date() : x(100)
{
}
Date::Date(int value) : x(value), y(value)
{
}
void main()
{
    Date d;
    Date d2(100);
}
```

Const & Reference data members

- ▶ References that are not constant can not be instantiated with a constant value !

App.cpp (A)

```
class Date
{
    int & y;
public:
    Date() : y(123) {}
};

void main()
{
    Date d;
}
```

App.cpp (B)

```
class Date
{
    const int & y;
public:
    Date() : y(123) {}
};

void main()
{
    Date d;
}
```

```
error C2440: 'initializing': cannot convert from 'int' to 'int &'
error C2439: 'Date::y': member could not be initialized
note: see declaration of 'Date::y'
```

This code compiles !

- ▶ However, ***it is not recommended*** to instantiate a constant reference in this way as it will create a pointer / reference to a value located on the stack !

Const & Reference data members

- ▶ Let's analyze the following code. What will be printed on the screen upon the execution of this code ?

App.cpp (B)

```
class Date
{
public:
    const int & y;
    Date() : y(123) {}
    void Test() {
        int a[1000];
        for (int tr = 0; tr < 1000; tr++)
            a[tr] = 50;
    }
};
void main()
{
    Date d;
    printf("%d\n", d.y);
    d.Test();
    printf("%d\n", d.y);
}
```

Const & Reference data members

- ▶ Let's analyze the following code. What will be printed on the screen upon the execution of this code ?

App.cpp (B)

```
class Date
{
public:
    const int & y;
    Date() : y(123) {}
    void Test() {
        int a[1000];
        for (int tr = 0; tr < 1000;
             a[tr] = 50;
    }
    void main()
    {
        Date d;
        printf("%d\n", d.y);
        d.Test();
        printf("%d\n", d.y);
    }
}
```

```
push    ebp
mov     ebp,esp
sub    esp,48h
push    ebx
push    esi
push    edi
```

```
mov     dword ptr [this],ecx
mov     dword ptr [ebp-8],123
mov     eax,dword ptr [this]
lea     ecx,[ebp-8]
mov     dword ptr [eax],ecx
eax,dword ptr [this]
```

```
pop    edi
pop    esi
pop    ebx
mov    esp,ebp
pop    ebp
ret
```

Stub

C++ code translation:

```
{  
    int temp = 123;  
    this->y = &temp;  
}
```

Stub

- ▶ After the constructor is called, *d.y* will point to an address on the stack that holds value 123.

Const & Reference data members

- ▶ Let's analyze the following code. What will be printed on the screen upon the execution of this code ?

App.cpp (B)

```
class Date
{
public:
    const int & y;
    Date() : y(123) {}
    void Test()
    {
        int a[1000];
        for (int tr = 0; tr < 1000; tr++)
            a[tr] = 50;
    }
};
void main()
{
    Date d;
    printf("%d\n", d.y);
    d.Test();
    printf("%d\n", d.y);
}
```

Will print **123** to the screen

Const & Reference data members

- ▶ Let's analyze the following code. What will be printed on the screen upon the execution of this code ?

App.cpp (B)

```
class Date
{
public:
    const int & y;
    Date() : y(123) {}
    void Test() {
        int a[1000];
        for (int tr = 0; tr < 1000; tr++)
            a[tr] = 50;
    }
};
void main()
{
    Date d;
    printf("%d\n", d.y);
    d.Test();
    printf("%d\n", d.y);
}
```

When *d.Test()* is called, the stack will be re-written with 1000 values of 50. As *d.y* is located on the stack, the value will be changed.

As a result, *d.y* will be 50 and the value written on the screen the second time will be 50 !!!

The background features a large, abstract graphic on the left side composed of overlapping blue and dark blue triangles and trapezoids, creating a sense of depth and motion.

Delegating constructor

Delegating constructor

- ▶ A constructor can call another constructor during its initialization.

App.cpp

```
class Object
{
    int x, y;
public:
    Object(int value) : x(value), y(value) {}

    Object() : Object(0) { }

void main()
{
    Object o;
}
```

- ▶ In this case , when we create “*Object o*” the default constructor will be called that in terms will call the second constructor (*Object(int)*)

Delegating constructor

- ▶ This code will not compile. When calling a constructor from another constructor initialization list, other initializations are not possible.

App.cpp

```
class Object
{
    int x, y;
public:

    Object(int value) : x(value), y(value) {}

    Object() : Object(0) , y(1) { }

};

void main()
{
    Object o;
}
```

error C3511: 'Object': a call to a delegating constructor shall be the
only member-initializer
error C2437: 'y': has already been initialized

Delegating constructor

- ▶ The same error is provided even if we do not initialize “y” in the Object(int) constructor.

App.cpp

```
class Object
{
    int x, y;
public:

    Object(int value) : x(value) {}

    Object() : Object(0), y(1) { }

};

void main()
{
    Object o;
}
```

CL (Windows)

```
error C3511: 'Object': a call to a delegating constructor
shall be the only member-initializer
error C2437: 'y': has already been initialized
```

gcc (Linux)

```
error: mem-initializer for 'Object::y' follows constructor
delegation
```

clang (MAC/OSX)

```
error: an initializer for a delegating constructor must
appear alone
```

```
Object() : Object(0), y(0) { }
```

Delegating constructor

- ▶ This code will compile. In this case, the “y” data member is initialized in the code of the constructor.

App.cpp

```
class Object
{
    int x, y;
public:

    Object(int value) : x(value), y(value) {}

    Object() : Object(0) { y = 1; }
};

void main()
{
    Object o;
}
```

- ▶ Keep in mind that “y” is initialized twice. Once in “Object(int)” call (the delegation call), and then in the default constructor body.

Delegating constructor

- ▶ This code will compile. “y” is first initialized by the delegation (“`y(value+5)`”) → meaning that `y` will be 5 before running the code from the default constructor.
- ▶ As a results, when running “`y+=5`”, “y” already has a value and its final value will be 10.

App.cpp

```
class Object
{
    int x, y;
public:

    Object(int value) : x(value), y(value+5) {}

    Object() : Object(0) { y += 5; }

void main()
{
    Object o;
}
```

Delegating constructor

- ▶ This code will also compile.
- ▶ However, since the delegation was removed, “y+=5” does not have an already uses a “y” that a stack value (something that we can not approximate).
- ▶ This code will compile, but the value of “y” is undetermined.

App.cpp

```
class Object
{
    int x, y;
public:

    Object(int value) : x(value), y(value+5) {}

    Object() { y += 5; }

};

void main()
{
    Object o;
}
```

Delegating constructor

- Constant or reference data members must NOT be instantiated on all constructors if delegation is used.

App.cpp

```
class Object
{
    int x, y;
    const int z;
public:

    Object(int value) : x(value), y(value), z(value) {}

    Object() : Object(0) { y = 1; }
};

void main()
{
    Object o;
}
```

- In this case, the default constructor MUST not instantiate “z” as “z” is already instantiated in constructor *Object(int)* that is called by the default constructor.

Delegating constructor

- ▶ It is possible to create a circular reference (as in the example below). The default constructor is calling the *Object(int)* that in terms calls the default constructor.

App.cpp

```
class Object
{
    int x, y;
public:
    Object(int value) : Object() {}
    Object() : Object(0) { }
};

void main()
{
    Object o;
}
```

- ▶ This code will compile, but the execution will initially freeze and after the stack is filled in due to recursive calls between constructors, it will create a run-time error (e.g. segmentation error in linux)

The background features a large, abstract graphic on the left side composed of overlapping blue triangles of varying shades of blue. It has a subtle, organic, and geometric feel.

Initialization lists ► for classes

Initialization lists for classes

- ▶ Classes and structures can be initialized using initialization lists:

App.cpp

```
struct Data
{
    int x;
    char t;
    const char* m;
};

void main()
{
    Data d1{ 10, 'A', "test" };

    Data d2 = { 5, 'B', "C++" };

    Data array[] = {
        { 1, 'A', "First element" },
        { 2, 'B', "Second element" },
        { 3, 'C', "Third element" },
    };
}
```

Initialization lists for classes

- ▶ Classes and structures can be initialized using initialization lists:

App.cpp

```
struct Data
{
    int x;
    char t;
    const char* m;
};

void main()
{
    Data d1{ 10, 'A', "test" };

    Data d2 = { 5, 'B', "C++" };

    Data array[] = {
        { 1, 'A', "First element" },
        { 2, 'B', "Second element" },
        { 3, 'C', "Third element" },
    };
}
```

```
Data d1{ 10, 'A', "test" };
mov    dword ptr [ebp-10h],0Ah
mov    byte ptr [ebp-0Ch],41h
mov    dword ptr [ebp-8],address of "test"

Data d2 = { 5, 'B', "C++" };
mov    dword ptr [ebp-24h],5
mov    byte ptr [ebp-20h],42h
mov    dword ptr [ebp-1Ch],address of "C++"
```

Initialization lists for classes

- ▶ Classes and structures can be initialized using initialization lists:

App.cpp

```
class Data
{
public:
    int x;
    char t;
    const char* m;
};

void main()
{
    Data d1{ 10, 'A', "test" };

    Data d2 = { 5, 'B', "C++" };

    Data array[] = {
        { 1, 'A', "First element" },
        { 2, 'B', "Second element" },
        { 3, 'C', "Third element" },
    };
}
```

- ▶ This code works, but it is important for data members to be **public**

Initialization lists for classes

- ▶ Classes and structures can be initialized using initialization lists:

App.cpp

```
class Data
{
    int x;
public:
    char t;
    const char* m;
};
void main()
{
    Data d1{ 10, 'A', "test" };

    Data d2 = { 5, 'B', "C++" };

    Data array[] = {
        { 1, 'A', "First element" },
        { 2, 'B', "Second element" },
        { 3, 'C', "Third element" },
    };
}
```

error C2440: 'initializing': cannot convert from
'initializer list' to 'Data'
note: No constructor could take the source type,
or constructor overload resolution was ambiguous

- ▶ This code will not work as “x” is not public ! If a class has at least one member that is **NOT** public and no matching constructor, these assignments will not be possible.

Initialization lists for classes

- ▶ Classes and structures can be initialized using initialization lists:

App.cpp

```
class Data
{
    int x;
public:
    char t;
    const char* m;

    Data(int xx, char tt, const char * mm) : x(xx), t(tt), m(mm) {};
};

void main()
{
    Data d1{ 10, 'A', "test" };

    Data d2 = { 5, 'B', "C++" };
    Data array[] = {
        { 1, 'A', "First element" },
        { 2, 'B', "Second element" },
        { 3, 'C', "Third element" },
    };
}
```

- ▶ This code will compile because a proper public constructor has been added.

Initialization lists for classes

- ▶ This code will not compile. If there at least one constructor and its parameters do **NOT** match the ones from the initialization list, the compiler will throw an error !

App.cpp

```
class Data
{
    int x;
public:
    char t;
    const char* m;

    Data(int xx, char tt) : x(xx), t(tt), m(nullptr) {};

};

void main()
{
    Data d1{ 10, 'A', "test" };

    Data d2 = { 5, 'B', "C++" };
    Data array[] = {
        { 1, 'A', "First element" },
        { 2, 'B', "Second element" },
        { 3, 'C', "Third element" },
    };
}
```

error C2440: 'initializing': cannot convert from
'initializer list' to 'Data'
note: No constructor could take the source type,
or constructor overload resolution was ambiguous

Initialization lists for classes

- ▶ Promotion and casting rules work in a similar way as for a regular method call.
- ▶ In this case, *true* is promoted to *int*, ‘A’ (a *char*) is promoted to *int* and “test” (a *const char ** pointer) is casted to a *const void **, allowing the compiler to call the existing constructor.

App.cpp

```
class Data
{
    int x;
public:
    char t;
    const char* m;

    Data(int xx, int tt, const void * p) : x(xx), t(tt), m((const char *)p) {};
};

void main()
{
    Data d1{ true, 'A', "test" };
    Data d2 = { true, 'A', "test" };
}
```

- ▶ This code compiles correctly

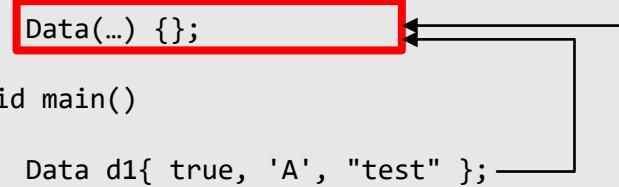
Initialization lists for classes

- ▶ Promotion and casting rules work in a similar way as for a regular method call.
- ▶ In this case a constructor that works like a fallback method exists and since there is no good match, it will be used to initialize *d1* and *d2* instances of Data.

App.cpp

```
class Data
{
    int x;
public:
    char t;
    const char* m;

    Data(...){};
```



```
};

void main()
{
    Data d1{ true, 'A', "test" };
    Data d2 = { true, 'A', "test" };
}
```

- ▶ This code compiles correctly

Initialization lists

- ▶ Trying to initialize this class with an *empty initialization list {}* will result in an error if no default constructor is present
- ▶ This code will NOT compile.

App.cpp

```
class Data
{
    int x;
public:
    char t;
    const char* m;

    Data(int xx, int tt, const void * p) : x(xx), t(tt), m((const char *)p) {};
};

void main()
{
    Data d1{};
    Data d2 = {};
}
```

error C2512: 'Data': no appropriate default constructor available
note: No constructor could take the source type,
or constructor overload resolution was ambiguous

Initialization lists for classes

- ▶ If however, either a *default constructor* or a constructor that models a *fallback* function is present, the following code will compile and run correctly.

App.cpp

```
class Data
{
    int x;
public:
    char t;
    const char* m;

    Data() {};
};

void main()
{
    Data d1{};
    Data d2 = {};
}
```

App.cpp

```
class Data
{
    int x;
public:
    char t;
    const char* m;

    Data(...) {};
};

void main()
{
    Data d1{};
    Data d2 = {};
}
```

Initialization lists for classes

- ▶ If no constructor is present:

App.cpp

```
class Data
{
    int x;
public:
    char t;
    const char* m;
};
void main()
{
    Data d1 {};
```

- ▶ This code will compile. “d1” object will have the following values after the execution:
 - d1.x = 0
 - d1.t = ‘\0’
 - d1.m = nullptr;

Initialization lists for classes

- ▶ If no constructor is present:

App.cpp

```
class Data
{
    int x;
public:
    char t;
    const char* m;
};
void main()
{
    Data d1 {};
}
```

The diagram illustrates the assembly code generated for the initialization of object `d1`. A callout bubble points from the curly braces in the C++ code to the corresponding assembly instructions:

- `xor eax,eax`
- `mov dword ptr [d1],eax`
- `mov dword ptr [ebp-8],eax`
- `mov dword ptr [ebp-4],eax`

A red box highlights the curly braces in the C++ code, and a blue arrow points from this box to the assembly code.

- ▶ This code will compile. “`d1`” object will have the following values after the execution:

- `d1.x = 0`
- `d1.t = '\0'`
- `d1.m = nullptr;`

Initialization lists for classes

- ▶ If no constructor is present:

App.cpp

```
class Data
{
    int x;
public:
    char t;
    const char* m;
};
void main()
{
    Data d1 {};
```

`memset (&d1, 0, sizeof(d1))`

- ▶ The compiler creates a code that fills the entire content of Data with 0 (similar to a `memset` call).

Initialization lists for classes

- If a constructor with only one parameter is present, the following initialization is also possible:

App.cpp

```
class Data
{
    int x;
public:
    Data(int value) : x(value) {}
};
void main()
{
    Data d = 10;
}
```

push
lea
call
10
ecx, [d]
Data::Data

Initialization lists for classes

- ▶ Promotion and conversion rules also work in this case
 - In case (A) → a char ('A') is promoted to int → d.x = 65 (Ascii code of 'A')
 - In case (B) → a bool (true) is promoted to int → d.x = 1
 - In case (C) → a double is converted to int → d.x = 4 (*int(4.5)=4*)

App.cpp (A)

```
class Data
{
    int x;
public:
    Data(int value) :
        x(value) {}
};

void main()
{
    Data d = 'A';
}
```

App.cpp (B)

```
class Data
{
    int x;
public:
    Data(int value) :
        x(value) {}
};

void main()
{
    Data d = true;
}
```

App.cpp (C)

```
class Data
{
    int x;
public:
    Data(int value) :
        x(value) {}
};

void main()
{
    Data d = 4.5;
}
```

Initialization lists for classes

- ▶ Initialization lists can also be applied to initialized array defined as a class member:

App.cpp

```
class Data
{
    int x[4];
public:
    Data() : x{ 1, 2, 3, 4 } {}
};

void main()
{
    Data d;
}
```

App.cpp

```
class Data
{
    int x[4] = { 1, 2, 3, 4 };
public:

};

void main()
{
    Data d;
}
```

- ▶ The previous code will NOT compile on VS 2013 (as that version does not implement the full specification of Cx++11). However, it will work on VS 2017 or later and g++ >16.0.0 (g++ (Ubuntu 5.4.0-6ubuntu1~16.04.2) 5.4.0 20160609). This feature is available as part of C++11 standard.

Initialization lists for classes

- ▶ Other data members can be initialized in a similar manner. The following example shows how to initialize basic types data members.

App.cpp

```
class Data
{
    int x = 5;
    float y = 10.5f;
    bool t = false;
};

void main()
{
    Data d;
}
```

lea ecx,[d]
call Data::Data

- ▶ The compiler will add a new default constructor (as there isn't one defined already in the class). That new constructor will instantiate the values for "x", "y" and "t"

Initialization lists for classes

- ▶ Other data members can be initialized in a similar manner. The following example shows how to initialize basic types data members.

App.cpp

```
class Data
{
    int x = 5;
    float y = 10.5f;
    bool t = false;
};

void main()
{
    Data d;
}
```

The diagram illustrates the assembly code generated for the Data constructor. The code is as follows:

```
push    ebp  
mov     ebp,esp  
  
mov     dword ptr [this],ecx  
mov     eax,dword ptr [this]  
mov     dword ptr [eax],5  
mov     eax,dword ptr [this]  
movss   xmm0,dword ptr ds:[0E36AE4h]  
movss   dword ptr [eax+4],xmm0  
mov     eax,dword ptr [this]  
mov     byte ptr [eax+8],0  
mov     eax,dword ptr [this]  
  
mov     esp,ebp  
pop    ebp  
ret
```

Annotations with arrows point from specific assembly instructions to their corresponding initial values:

- An arrow points from the first `mov` instruction (initializing `x`) to a blue box labeled `x=5`.
- An arrow points from the `movss` instruction (initializing `y`) to a blue box labeled `y=10.5`.
- An arrow points from the second `mov` instruction (initializing `t`) to a blue box labeled `t = false (0)`.

- ▶ The compiler will add a new default constructor (if there is none already in the class). That new constructor will initialize the "x", "y" and "t"

Initialization lists for classes

- ▶ Other data members can be initialized in a similar manner. The following example shows how to initialize basic types data members.

App.cpp

```
class Data
{
    int x = 5;
    float y = 10.5f;
    bool t = false;
public:
    Data() {
        _asm nop;
        _asm nop;
    }
};

void main()
{
    Data d;
```

- ▶ Adding a constructor will force the compiler to modify that constructor to integrate the default initialization as well.

Initialization lists for classes

- ▶ Other data members can be initialized in a similar manner. The following example shows how to initialize basic types data members.

App.cpp

```
class Data
{
    int x = 5;
    float y = 10.5f;
    bool t = false;
public:
    Data() {
        _asm nop;
        _asm nop;
    }
};

void main()
{
    Data d;
}
```

```
mov     dword ptr [this],ecx
mov     eax,dword ptr [this]
mov     mov     dword ptr [eax],5
mov     eax,dword ptr [this]
movss  xmm0,dword ptr ds:[0E36AE4h]
movss  dword ptr [eax+4],xmm0
mov     eax,dword ptr [this]
mov     byte ptr [eax+8],0
nop
nop
mov     eax,dword ptr [this]
```

- ▶ Adding a constructor will force the compiler to integrate the default initialization code.

Initialization lists for classes

- ▶ Initialization lists can also be applied to initialized array defined as a class member:

App.cpp

```
class Data
{
    int x = 5;
    float y = 10.5f;
    bool t = false;
public:
    Data() : x(10) { }
};

void main()
{
    Data d;
}
```

- ▶ Furthermore, the default values can be overridden in the constructor list. In this case, “x” will be initialized with 10, “y” with 10.5 and “t” with false

Initialization lists for classes

- ▶ Initialization lists can also be used for pointers.

App.cpp

```
class Date
{
    int * x = new int[10];
public:
};

void main()
{
    Date d;
}
```

App.cpp

```
class Date
{
    int * x = new int[10] { 1,2,3,4,5,6,7,8,9,10 };
public:
};

void main()
{
    Date d;
}
```

- ▶ In both of these cases, a *default constructor* is created that will call *new* operator and instantiate pointer “x”

Initialization lists for classes

- ▶ The same thing can be done for pointers by using constructor initialize list:

App.cpp

```
class Date
{
    int * x;
public:
    Date(int count): x(new int[count]) { }
};

void main()
{
    Date d(3);
}
```

App.cpp

```
class Date
{
    int * x;
public:
    Date(int count): x(new int[count] {1,2,3} ) { }
};

void main()
{
    Date d(3);
}
```

- ▶ In this case, the call to the *new* operator will be added in the constructor.

Initialization lists for classes

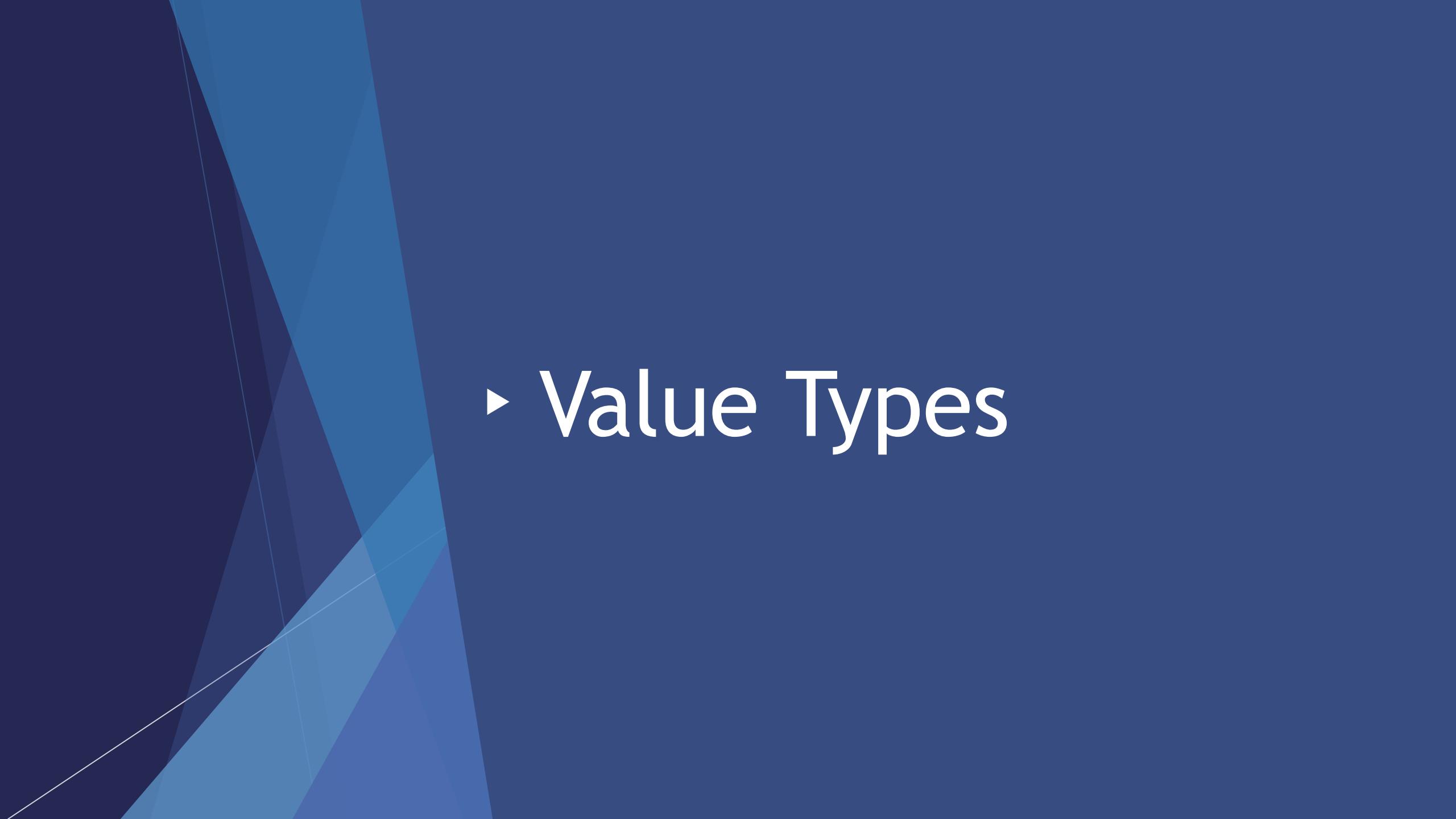
- ▶ Initialization lists can also be used to return a value from a function:

App.cpp

```
struct Student
{
    const char * Name;
    int Grade;
};

Student GetStudent()
{
    return { "Popescu", 10 };
}

void main()
{
    Student s;
    s = GetStudent();
}
```

A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented diagonally, creating a sense of depth and movement.

► Value Types

Value Types

- ▶ When an expression is evaluated, each of its terms are associated with a value type. This helps the compiler to understand how to use that value (and also what kind of methods / functions can be used for overload resolution).
- ▶ Currently, there are 5 such types:
 1. glvalue
 2. prvalue
 3. xvalue
 4. lvalue
 5. rvalue
- ▶ The way these types work, and what they represent has been changed from standard to standard.

Value Types: glvalue (generalized lvalue)

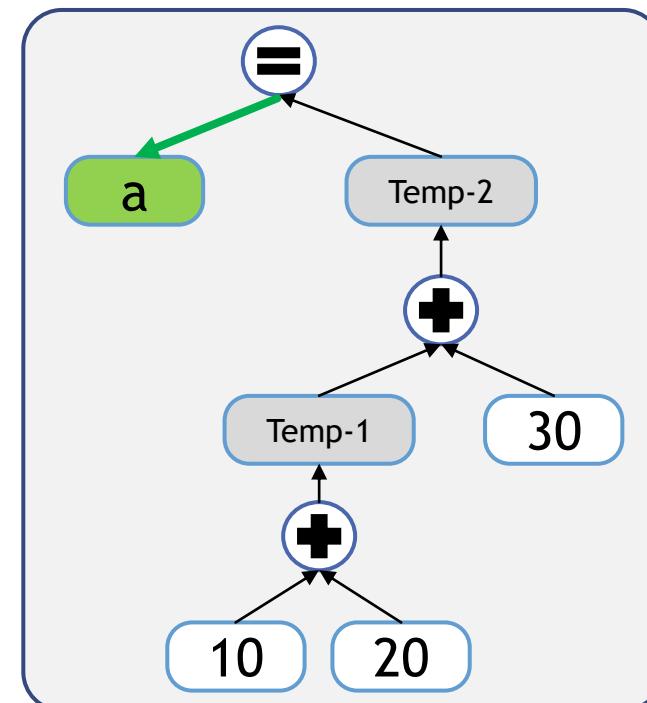
- ▶ A **glvalue** is an expression that results in an object
- ▶ Examples:
 1. Assignment → *a = <expression>*, where “a” is a variable/data member (“a” in this context will be a **glvalue**). It’s valid for other type of assignments operators such as “+=, -= , *= , etc)
 2. Pre-increment/decrement → *++a, --a* where “a” is a variable/data member (“a” in this context will be a **glvalue**).
 3. Array members → *a[n]* where “a” is an array (“a[n]” in this context will be a **glvalue**).
 4. A method/function that returns a reference → *int& GetSomething()*
 5. ...
- ▶ To simplify this observation, consider a **glvalue** an expression that refers to a memory offset of a variable / data member).

Value Types: xvalue (eXpiring value)

- ▶ A **xvalue** is an expression that results in an object that can be reused (a temporary object).
- ▶ Let's consider the following example:

```
int a = 10+20+30;
```

- ▶ This code will be evaluated in the following way:
 1. We first add “10” with “20”
 2. The result is a temporary value (Temp-1)
 3. Then we add Temp-1 with “30”
 4. The result is another temporary value (Temp-2)
 5. Finally - we copy the value from Temp-2 into “a”
- ▶ Both “Temp-1” and “Temp-2” are **xvalues**
- ▶ “a” is a **glvalue**
- ▶ “10”, “20” and “30” are **prvalues**



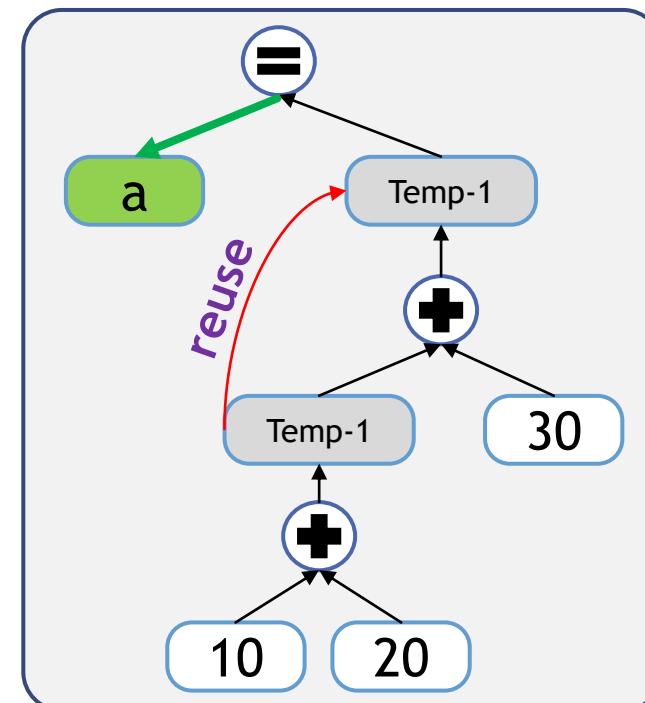
Value Types: xvalue (eXpiring value)

- ▶ A **xvalue** is an expression that results in an object that can be reused (a temporary object).
- ▶ Let's consider the following example:

```
int a = 10+20+30;
```

- ▶ In practice, “Temp-1” only exists until the next operation (addition with “c” is completed).
- ▶ In this case, “Temp-1” can be reused (rather than create another temporary variable). This can improve the evaluation performance.

```
Temp-1 = 10 + 20  
Temp-1 = Temp-1 + 30  
a = Temp-1
```



Value Types: prvalue (pure rare value)

- ▶ A **prvalue** is an expression that reflects a value
- ▶ Examples:
 1. Numerical constants → *10, 100, true, false, nullptr*
 2. Post-increment/decrement → *a++, a--* where “a” is a variable/data member (“a” in this context will be a **prvalue**).
 3. A method/function that returns a value → *int GetSomething()*
 4. ...
- ▶ an “glvalue” can be transformed in a “prvalue” (this is often call lvalue-to-rvalue conversion). This is normal (if the glvalue refers to a location of memory , it can be transformed in a prvalue if it refers to the value that resides in that memory location).

Value Types: lvalue (left value)

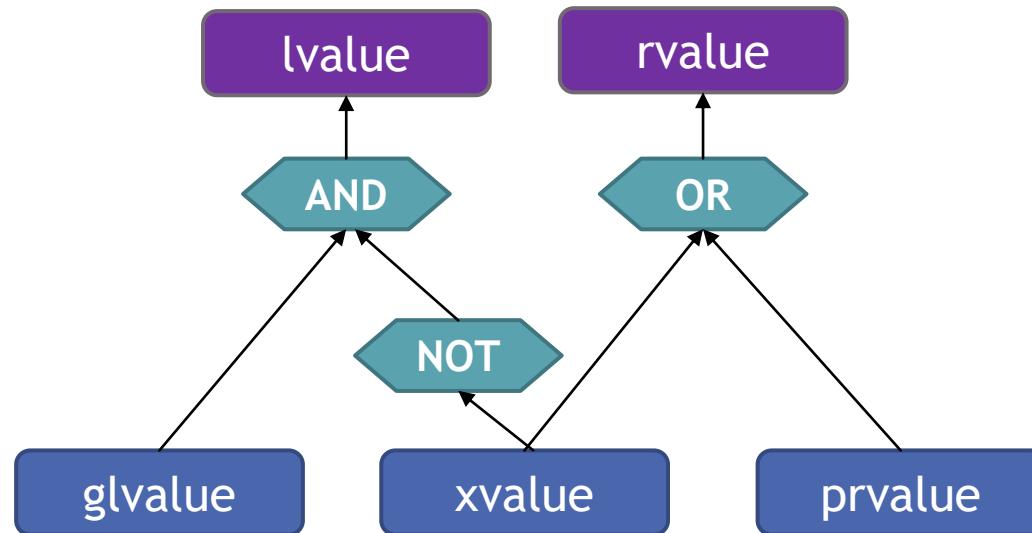
- ▶ A **lvalue** is a **glvalue** that is **NOT** an **xvalue**
- ▶ lvalue got its name because it refers mostly to the left location within an expression.

```
int a = 10;  
int b;  
b = (a += 20)+10;
```

- ▶ In this example, both “a” and “b” are **lvalues** (and **glvalues**), “20” and “10” are **prvalues** (and **rvalues**).

Value Types: rvalue (right value)

- ▶ A **rvalue** is a **prvalue** **OR** an **xvalue**
- ▶ rvalue got its name because it refers mostly to the right location within an expression.
- ▶ **lvalue** and **rvalue** are considered mixed categories of type values
- ▶ **glvalue**, **xvalue** and **prvalue** are considered primary categories



The background features a large, abstract graphic on the left side composed of various shades of blue and dark blue triangles, creating a geometric pattern that tapers towards the bottom-left corner.

Copy & Move ▶ Constructors

Copy constructor

- ▶ A *copy constructor* is a constructor that has only one parameter that is a reference (const or not-const) to the same class as the current one.

App.cpp

```
class Date
{
public:
    Date(const Date & d);
};
```

Copy Constructor

- ▶ It is usually used in the following way:

App.cpp

```
class Date
{
    int value;
public:
    Date(const Date &d) { value = d.value; }
    Date(int v) { value = v; }
};
int main()
{
    Date d(1);
    Date d2 = d;
    return 0;
}
```

Copy Constructor

Copy constructor

- ▶ A *copy constructor* is a constructor that has only one parameter that is a reference (const or not-const) to the same class as the current one.

App.cpp

```
class Date
{
public:
    Date(const Date & d);
};
```

Copy Constructor

- ▶

App.cpp

```
class Date
{
    int value;
public:
    Date(const Date &d)
    Date(int v) { value = v };
};
int main()
{
    Date d(1);
    Date d2 = d;
    return 0;
}
```

lea
push
lea
call
Date::Date

Copy constructor

- ▶ The copy constructor is also used whenever a function/method has a parameter of that class type that is not send via reference !

App.cpp

```
class Date
{
    int x,y,z,t;
public:
    Date(const Date &d) { x = d.x; y = d.y; z = d.z; t = d.y; }
    Date(int v) { x = y = z = t = v; }
};

void Process(Date d) { ... }

int main()
{
    Date d(1);
    Process(d);
    return 0;
}
```

In this case , a copy of “d” is made, and that copy is pass to function Process. More on this mechanisms on next course.

Copy constructor

- ▶ Similarly, if a function returns an object (not a reference or a pointer) the copy constructor is used.

App.cpp

```
class Date
{
    int x,y,z,t;
public:
    Date(const Date &d) { x = d.x; y = d.y; z = d.z; t = d.y; }
    Date(int v) { x = y = z = t = v; }
};

Date Process()
{
    Date d(1);
    return d;
}
```

This is where the copy construction will be called. More on this topic on the next course

Copy constructor

- ▶ A copy constructor can be declared in two ways:
 - With a const parameter (this is the most generic usage)
 - With a non-const parameter

App.cpp

```
class Date
{
    int x;
public:
    Date(const Date &d) { x = d.x; }
    Date(Date &d) { x = d.x * 2; }
    Date(int v) { x = v; }
};
```

Both declarations are copy constructors.

- ▶ Some compilers might produce a warning in this case: “**warning C4521: ‘Date’: multiple copy constructors specified**”
- ▶ It’s best to use the copy constructor that uses a constant reference as a parameter (this is more generic, and any non-constant references can be converted to a const reference).

Copy constructor

- ▶ If both types of copy constructors are present (with const and non-const parameter), the compiler will choose the best fit.
- ▶ In this case, since “d” is not a constant, the non-constant form of the copy constructor will be used.

App.cpp

```
class Date
{
    int x;
public:
    Date(const Date &d) { x = d.x; }
    Date(Date &d) { x = d.x * 2; } ←
    Date(int v) { x = v; }
};

int main()
{
    Date d(1);
    Date d2 = d; →
    return 0;
}
```

Copy constructor

- In this case, even if the “d” is not a constant, its non-const reference can be converted to a constant reference and then used the copy constructor with a constant parameter.

App.cpp

```
class Date
{
    int x;
public:
    Date(const Date &d) { x = d.x; } ←
    Date(int v) { x = v; }
};

int main()
{
    Date d(1);
    Date d2 = d; ←
    return 0;
}
```

Copy constructor

- In this case the compiler will call the copy constructor that has a “const” parameter.

App.cpp

```
class Date
{
    int x;
public:
    Date(const Date &d) { x = d.x; } ←
    Date(Date &d) { x = d.x * 2; }
    Date(int v) { x = v; }
};

int main()
{
    const Date d(1);

    Date d2 = d; ←

    return 0;
}
```

Copy constructor

- In this case the code will not compile. There a copy constructor defined, but it does not accept const parameters !

App.cpp

```
class Date
{
    int x;
public:

    Date(Date &d) { x = d.x * 2; }
    Date(int v) { x = v; }
};

int main()
{
    const Date d(1);
    Date d2 = d; // Error occurs here
    return 0;
}
```

error C2440: 'initializing': cannot convert from 'const Date' to
'Date'
note: Cannot copy construct class 'Date' due to ambiguous copy
constructors or no available copy constructor

Copy constructor

- ▶ This code will compile.
- ▶ Since there is no copy constructor defined, the compiler will generate a code that copies the data from “d” to “d2” (similar to what *memcpy* function does).

App.cpp

```
class Date
{
    int x;
public:

    Date(int v) { x = v; }
};

int main()
{
    const Date d(1);
    Date d2 = d;           ← mov    eax,dword ptr [d]
                           mov    dword ptr [d2],eax
    return 0;
}
```

Move constructor

- ▶ Let's analyze the following code:

App.cpp

```
char * DuplicateString(const char * string) {
    char * result = new char[strlen(string) + 1];
    memcpy(result, string, strlen(string) + 1);
    return result;
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }
int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Move constructor

- ▶ Let's analyze the following code :

App.cpp

```
char * DuplicateString(const char * string) {
    char * result = new char[strlen(string) + 1];
    memcpy(result, string, strlen(string) + 1);
    return result;
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }

int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

CTOR: Allocate sir to 00AB0610
COPY-CTOR: Copy sir from 00AB0610 to 00AB0648

Move constructor

- ▶ Let's analyze the following code :

App.cpp

```
char * DuplicateString(const char * string)
{
    char * result = new char[strlen(string) + 1];
    memcpy(result, string, strlen(string) + 1);
    return result;
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }

int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Step What happened

1. A temporary object is created (it's temporary because it is not assigned to any variable)

CTOR: Allocate sir to 00AB0610

Move constructor

- ▶ Let's analyze the following code :

App.cpp

```
char * DuplicateString(const char * string)
{
    char * result = new char[strlen(string)];
    memcpy(result, string, strlen(string));
    return result;
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }

int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Step What happened

1. A temporary object is created (it's temporary because it is not assigned to any variable)
2. That temporary object is sent to Get(...) function

CTOR: Allocate sir to 00AB0610

Move constructor

- ▶ Let's analyze the following code :

App.cpp

```
char * DuplicateString(const char * string)
{
    char * result = new char[strlen(string)];
    memcpy(result, string, strlen(string));
    return result;
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }

int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Step What happened

1. A temporary object is created (it's temporary because it is not assigned to any variable)
2. That temporary object is sent to Get(...) function
3. Get(...) function returns it (so it calls the copy-ctor)

CTOR: Allocate sir to 00AB0610

Move constructor

- ▶ Let's analyze the following code :

App.cpp

```
char * DuplicateString(const char * string)
{
    char * result = new char[strlen(string)];
    memcpy(result, string, strlen(string));
    return result;
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }

int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Step What happened

1. A temporary object is created (it's temporary because it is not assigned to any variable)
2. That temporary object is sent to Get(...) function
3. Get(...) function returns it (so it calls the copy-ctor)
4. Within the copy-ctor the string is copied again

CTOR: Allocate sir to 00AB0610

COPY-CTOR: Copy sir from 00AB0610 to 00AB0648

Move constructor

So ... what is the problem ?

- ▶ We allocate memory and we copy the same string twice !!!
- ▶ First as part of the constructor
- ▶ Second as part of the copy-constructor

This is not unusual; however, the first constructor creates a temporary object (an object that only exists during the evaluation of the following expression:

```
Date d = Get(Date("test"));
```

So ... we know that we have allocated memory for an object that we can not control after the expression is evaluated.

Q: Do we need to allocate the memory twice ? (or can't we just use the original memory that was allocated ?)

Move constructor

- ▶ A move constructor is declared using “`&&`” to refer to temporary value. It is mostly used to reuse an allocated memory.

App.cpp

```
class Date
{
    char * pointer;
public:
    Date(Date && d) { char * temp = d.pointer; d.pointer = nullptr; this->pointer = temp; }
};
```

- ▶ If no “move” constructor is provided, but a “copy” constructor exists, the compiler will use the copy-constructor. This is valid only for temporary values (such as an **xvalue**). Move constructor is never used for a glvalue or a rvalue.
- ▶ A move constructor can be used with a `const` parameter

App.cpp

```
class Date
{
    Date(const Date && d) { ... }
};
```

However, as usually in the “move-constructor” the parameter received will be modified, the “`const`” form is not used.

Move constructor

- ▶ Let's analyze the following code :

App.cpp

```
char * DuplicateString(const char * string) {
    ...
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p\n", d.sir, sir);
    }

    Date(Date &&d) { printf("Move from %p to %p\n", &d, this);sir = d.sir;d.sir = nullptr; }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p\n", sir);
    }
};

Date Get(Date d) { return d; }

int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Move Constructor

Move constructor

- ▶ Let's analyze the following code :

App.cpp

```
char * DuplicateString(const char * string) {
    ...
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p\n", d.sir, sir);
    }

    Date(Date &&d) { printf("Move from %p to %p\n", &d, this);sir = d.sir;d.sir = nullptr; }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p\n", sir);
    }
};

Date Get(Date d) { return d; }

int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Move Constructor

CTOR: Allocate sir to 00AB0610
Move from 00AB0610 to 00D8FE04

A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented diagonally, creating a sense of depth and movement.

► Constraints

Constraints

Singleton pattern.

- ▶ Problem: what if we want to model a class that can only have one instance ?
The solution is to combine a private constructor with a static function:

App.cpp

```
class Object
{
    int value;
    static Object* instance;
    Object() { value = 0; }
public:
    static Object* GetInstance();
};

Object* Object::instance = nullptr;

Object* Object::GetInstance()
{
    if (instance == nullptr)
        instance = new Object();
    return instance;
}

void main()
{
    Object *obj = Object::GetInstance();
}
```

The default constructor is private - thus an object of this type can not be created !

As **GetInstance** is a static method of class Object, it can access any private constructors. However, as **Object::instance** is a static variable, the **new** operator will only be called once (when the first instance is requested → therefore the name **Singleton**).

Constraints

Singleton pattern.

- ▶ Problem: what if we want to model a class that can only have one instance ?
The solution is to combine a private constructor with a static function:

App.cpp

```
class Object
{
    int value;
    static Object* instance;
    Object() { value = 0; }
public:
    static Object* GetInstance() { ... }
};
Object* Object::instance = nullptr;

void main()
{
    Object *obj1 = Object::GetInstance();
    Object *obj2 = Object::GetInstance();
    Object *obj3 = Object::GetInstance();
}
```

```
Object* Object::GetInstance() {
    if (instance == nullptr)
        instance = new Object();
    return instance;
}
```

- ▶ Both *obj1* , *obj2* and *obj3* are in reality the same pointer. When *obj1* is first requested, *Object::instance* is first allocated, then it gets returned for *obj2* and *obj3*.

Constraints

Singleton pattern.

- ▶ Problem: what if we want to model a class that can only have one instance ?
The solution is to combine a private constructor with a static function:

App.cpp

```
class Object
{
    int value;
    static Object* instance;
    Object() { value = 0; }
public:
    static Object* GetInstance() { ... }
};
Object* Object::instance = nullptr;

void main()
{
    Object obj1;
    Object * obj = new Object();
}
```

```
Object* Object::GetInstance() {
    if (instance == nullptr)
        instance = new Object();
    return instance;
}
```

error C2248: 'Object::Object': cannot access private
member declared in class 'Object'
note: see declaration of 'Object::Object'
note: see declaration of 'Object'

- ▶ This code will not compile !

Constraints

- ▶ Private constructors can also be used with friend function. This is useful if we want the entire functionality of a class to have a limited availability (only a couple of classes can use its functionality).

App.cpp

```
class Object
{
    int value;
    Object(): value(0) { }
    friend class ObjectUser;
};

class ObjectUser
{
public:
    int GetValue();
};

int ObjectUser::GetValue()
{
    Object o;
    return o.value;
}

void main()
{
    ObjectUser ou;
    printf("%d\n", ou.GetValue());
}
```

- ▶ This code will compile !

Constraints

- ▶ Private constructors can also be used with friend function. This is useful if we want the entire functionality of a class to have a limited availability (only a couple of classes can use its functionality).

App.cpp

```
class Object
{
    int value;
    Object(): value(0) { }
    friend class ObjectUser;
};

class ObjectUser
{
public:
    int GetValue();
};

int ObjectUser::GetValue()
{
    Object o;
    return o.value;
}

void main()
{
    Object obj;
}
```

error C2248: 'Object::Object': cannot access private member
declared in class 'Object'
note: see declaration of 'Object::Object'
note: see declaration of 'Object'

- ▶ This code will NOT compile !

Constraints

- ▶ Let's analyze the following code:

App.cpp

```
class Date
{
public:
    static int Suma(int x, int y) { return x + y; }
    static int Dif(int x, int y) { return x - y; }
    static int Mul(int x, int y) { return x * y; }
};

int main()
{
    Date d;
    printf("%d\n", Date::Suma(10, 20));
}
```

- ▶ Since class Data only has static functions, it makes no sense to allow creating instances of this class. However, with the current code, this is possible ! What can we do so that a programmer **CAN NOT** create an instance of type Data ?

Constraints

- ▶ Solution 1 → make the default constructor private:

App.cpp

```
class Date
{
    Date(); // Constructor highlighted with a red box
public:
    static int Suma(int x, int y) { return x + y; }
    static int Dif(int x, int y) { return x - y; }
    static int Mul(int x, int y) { return x * y; }
};

int main()
{
    Date d; // Declaration of Date object highlighted with a red box
    printf("%d\n", Date::Suma(10, 20));
}
```

- ▶ This code will not compile. However, a static method can still create an instance of Data.

Constraints

- ▶ Solution 2 → use the keyword **delete**

App.cpp

```
class Date
{
public:
    Date() = delete;
    static int Suma(int x, int y) { return x + y; }
    static int Dif(int x, int y) { return x - y; }
    static int Mul(int x, int y) { return x * y; }
};

int main()
{
    Date d;
    printf("%d\n", Date::Suma(10, 20));
}
```

- ▶ In this case we are telling the compiler that there is NO default constructor and it (the compiler) should not create one by default.

Constraints

- ▶ Let's analyze the following code:

App.cpp

```
class Date
{
    int value;
public:
    Date(int x) { value = x; }
};

int main()
{
    Date d('0');

    return 0;
}
```

- ▶ This code works, due to promotion mechanism ('a' (a char) is promoted to an int).
- ▶ What can we do if we **do not want** to allow creating objects with a char parameter, but we **do want** to allow creating objects with an int parameter ?

Constraints

- ▶ The solution is similar as with the previous cases:

App.cpp

```
class Date
{
    int value;
public:
    Date(char x) = delete;
    Date(int x) { value = x; }
};

int main()
{
    Date d('0');
    return 0;
}
```

error C2280: 'Date::Date(char)': attempting to reference a deleted function
note: see declaration of 'Date::Date'
note: 'Date::Date(char)': function was explicitly deleted

- ▶ This code will not compile.
- ▶ Using the **delete** keyword in this manner tells the compiler that there is a constructor that has a **char** parameter, but it can not be used !

Constraints

- ▶ Let's analyze the following code:

App.cpp

```
class Date
{
    int value;
public:
    Date(int v) { value = v; }
};

int main()
{
    Date d = 100;
    return 0;
}
```

push 100
lea [d]
call Date::Date

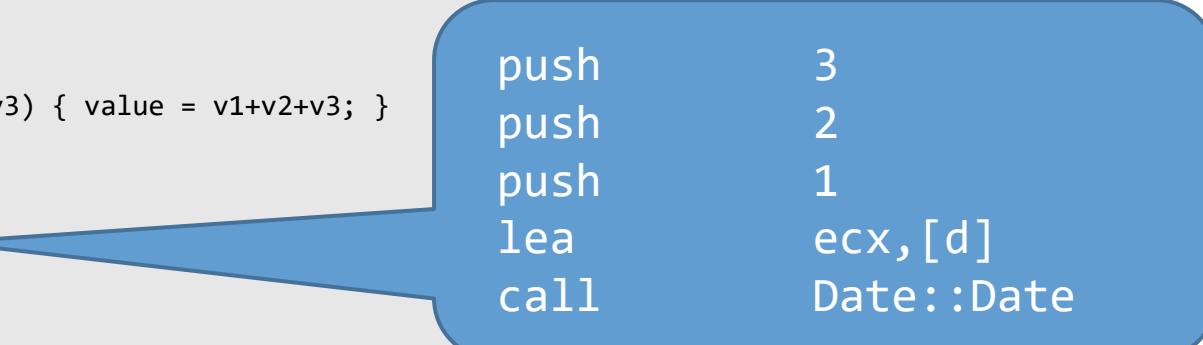
- ▶ This code compiles. Due to the initialization lists methods, the constructor that has an `int` parameter is called.

Constraints

- ▶ Similarly

App.cpp

```
class Date
{
    int value;
public:
    Date(int v1, int v2, int v3) { value = v1+v2+v3; }
};
int main()
{
    Date d = { 1, 2, 3 };
    return 0;
}
```



push	3
push	2
push	1
lea	ecx,[d]
call	Date::Date

- ▶ This code compiles. Again, due to the initialization lists method, if there is constructor that has 3 int parameter, it will be used.
- ▶ What can we do to force the usage of the constructor and not the initialization list (e.g. if we want to have a code that is compatible with older standards → this will only work for C++11 and after).

Constraints

- ▶ Similarly

App.cpp

```
class Date
{
    int value;
public:
    explicit Date(int v1, int v2, int v3) { value = v1+v2+v3; }
};
int main()
{
    Date d = { 1, 2, 3 };
    return 0;
}
```

error C3445: copy-list-initialization of 'Date' cannot use an
explicit constructor
note: see declaration of 'Date::Date'

- ▶ The solution is to use the keyword **explicit**. In this case we tell the compiler that it should use the constructor based initialization and not the initialization list method.
- ▶ This code does not compile. However, if we replace “**Date d = {1,2,3};**” with “**Date d(1, 2, 3);**” the code will compile !

Q & A

OOP

Gavrilut Dragos
Course 4

Summary

- ▶ Destructor
- ▶ C/C++ operators
- ▶ Operators for classes
- ▶ Operations with Objects

A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented diagonally, creating a sense of depth and movement.

► Destructor

Destructor

- ▶ A destructor function is called whenever we want to free the memory that an object occupies.
- ▶ A destructor (if exists) is only one and has no parameters.
- ▶ A destructor can not be static
- ▶ A destructor can have different access modifiers (public/private/protected).

App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
    ~Date(); // Red box highlights this line
};

Date::Date() : x(100) { ... }
Date::~Date() { ... }

void main()
{
    Date d;
}
```

Destructor

- ▶ The most common usage of the destructor is to deallocate the memory that has been allocated within the constructor or other functions.

App.cpp

```
class String
{
    char * text;
public:
    String(const char * s)
    {
        text = new char[strlen(s) + 1];
        memcpy(text, s, strlen(s) + 1);
    }
    ~String()
    {
        delete text;
        text = nullptr;
    }
};

void main()
{
    String * s = new String("C++");
    // some operations
    delete s;
}
```

Destructor

- ▶ This code will not compile. The destructor of class Date is private.

App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
private:
    ~Date();
};

Date::Date() : x(100) { ... }
Date::~Date() { ... }

void main()
{
    Date d;
}
```

error C2248: 'Date::~Date': cannot access private member
declared in class 'Date'
note: compiler has generated 'Date::~Date' here
note: see declaration of 'Date'

Destructor

- ▶ This code will compile - because the destructor will (even if private) is not be called (the object is created in the heap memory and it is never deallocated).

App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
private:
    ~Date();
};

Date::Date() : x(100) { ... }
Date::~Date() { ... }

void main()
{
    Date *d = new Date();
}
```

Destructor

- ▶ This code will not compile because “*delete d*” call will attempt to use a private destructor.

App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
private:
    ~Date();
};

Date::Date() : x(100) { ... }
Date::~Date() { ... }

void main()
{
    Date *d = new Date();
    delete d;
}
```

error C2248: 'Date::~Date': cannot access private member
declared in class 'Date'
note: compiler has generated 'Date::~Date' here
note: see declaration of 'Date'

Destructor

- ▶ This code will compile. The destructor is private , but it can be access by a method from its class (In this case DestroyData).

App.cpp

```
class Date
{
private:
    int x;
public:
    Date();
    static void DestroyData(Date *d);
private:
    ~Date();
};

Date::Date() : x(100) { ... }
Date::~Date() { ... }
void Date::DestroyData(Date *d)
{
    delete d;
}
void main()
{
    Date *d = new Date();
    Date::DestroyData(d);
}
```

Destructor

Let's consider the following class:

App.cpp

```
class Date {  
public:  
    ~Date() { printf("dtor was called \n"); }  
};
```

The destructor is called when:

- A. When program ends, for every global variable

```
Date d;  
int main() { return 0; }
```

- B. When a function/method ends for every local variable

```
int main() {  
    Date d;  
    return 0;  
}
```

- C. When the execution exists a scope (for variable defined within a specific scope)

```
int main() {  
    for (int tr=0;tr<10;tr++) {  
        Date d;  
    }  
    return 0;  
}
```

- D. When the **delete** operator is called over a heap allocated instance

```
int main() {  
    Date *d = new Date();  
    delete d;  
    return 0;  
}
```

Destructor

- Objects are destroyed in the reverse order of their creation (similar to the way a stack works → first created is the last destroyed).

App.cpp

```
class Tree {  
public:  
    ~Tree() { printf("dtor: Tree\n"); }  
};  
class Car {  
public:  
    ~Car() { printf("dtor: Car\n"); }  
};  
class Animal {  
public:  
    ~Animal() { printf("dtor: Animal\n"); }  
};  
class Date  
{  
    Tree t;  
    Car c;  
    Animal a;  
public:  
    ~Date() { printf("dtor: Date\n"); }  
};  
void main()  
{  
    Date d;  
}
```

Outputs:

```
dtor: Date  
dtor: Animal  
dtor: Car  
dtor: Tree
```

Destructor

- Objects are destroyed in the reverse order of their creation (similar to the way a stack works → first created is the last destroyed).

App.cpp

```
class Tree {  
public:  
    ~Tree() { printf("dtor: Tree\n"); }  
};  
class Car {  
public:  
    ~Car() { printf("dtor: Car\n"); }  
};  
class Animal {  
public:  
    ~Animal() { printf("dtor: Animal\n"); }  
};  
class Date  
{  
    Tree t;  
    Car c;  
    Animal a;  
public:  
    ~Date() { printf("dtor: Date\n"); }  
};  
void main()  
{  
    Date d;  
}
```

Outputs:
dtor: Date
dtor: Animal

mov	dword ptr [this],ecx
push	offset string "dtor: Date\n"
call	_printf
add	esp,4
mov	ecx,dword ptr [this]
add	ecx,2
call	Animal::~Animal
mov	ecx,dword ptr [this]
add	ecx,1
call	Car::~Car
mov	ecx,dword ptr [this]
call	Tree::~Tree

Destructor

- ▶ If the destructor is missing, but the class has data members that have their own destructors, one will be created by default !

App.cpp

```
class Tree {  
public:  
    ~Tree() { printf("dtor: Tree\n"); }  
};  
class Car {  
public:  
    ~Car() { printf("dtor: Car\n"); }  
};  
class Animal {  
public:  
    ~Animal() { printf("dtor: Animal\n"); }  
};  
class Date  
{  
    Tree t;  
    Car c;  
    Animal a;  
public:  
};  
void main()  
{  
    Date d;  
}
```

Outputs:
dtor: Animal
dtor: Car
dtor: Tree

Destructor

- ▶ Let's analyze the following code. Each object created has its unique ID. Upon execution the following code will output:

App.cpp

```
int global_id = 0;
class Date
{
    int id;
public:
    Date() { global_id++; id = global_id; printf("ctor id:%d\n", id); }
    ~Date() { printf("dtor id:%d\n", id); }
};
void main()
{
    Date *d = new Date();
    delete d;
}
```

Outputs:
ctor id: 1
dtor id: 1

Destructor

- ▶ Let's analyze the following code. Each object created has its unique ID. Upon execution the following code will output:

App.cpp

```
int global_id = 0;
class Date
{
    int id;
public:
    Date() { global_id++; id = global_id; printf("ctor id:%d\n", id); }
    ~Date() { printf("dtor id:%d\n", id); }
};
void main()
{
    Date *d = new Date[5];
    delete d;
}
```

An array of 5 instances of type Date is created.

Outputs:
ctor id: 1
ctor id: 2
ctor id: 3
ctor id: 4
ctor id: 5
dtor id: 1

- ▶ This program will crash as only the first object in the “d” array is destroyed. And it is not in the right order anyway.

Destructor

- ▶ Let's analyze the following code. Each object created has its unique ID. Upon execution the following code will output:

App.cpp

```
int global_id = 0;
class Date
{
    int id;
public:
    Date() { global_id++; id = global_id; printf("ctor id:%d\n", id); }
    ~Date() { printf("dtor id:%d\n", id); }
};
void main()
{
    Date *d = new Date[5];
    delete [] d;
}
```

Outputs:

```
ctor id: 1
ctor id: 2
ctor id: 3
ctor id: 4
ctor id: 5
dtor id: 5
dtor id: 4
dtor id: 3
dtor id: 2
dtor id: 1
```

- ▶ Now the program runs correctly.
- ▶ Whenever an array of instances is created into the heap, use ***delete[]*** operator to destroy it and not ***delete*** operator.
- ▶ ***delete[]*** operator will call the destructor function (if any) for every object in the array in the reverse order (starting from the last and moving forward to the first).

- ▶ C/C++ operators

Operators

- ▶ Depending on that operator's necessary number of parameters there are:
 - ❖ Unary
 - ❖ Binary
 - ❖ Ternary
 - ❖ Multi parameter
- ▶ Depending on the operation type, there are:
 - ❖ Arithmetic
 - ❖ Relational
 - ❖ Logical
 - ❖ Bitwise operators
 - ❖ Assignment
 - ❖ Others
- Depending on the overloading possibility
 - ❖ Those that can be overloaded
 - ❖ Those that can NOT be overloaded

Arithmetic operators

Operator	Type	Overload	Format	Returns
+	Binary	Yes	A + B	Value/reference
-	Binary	Yes	A - B	Value/reference
*	Binary	Yes	A * B	Value/reference
/	Binary	Yes	A / B	Value/reference
%	Binary	Yes	A % B	Value/reference
++ (post/pre-fix)	Unary	Yes	A++ or ++A	Value/reference
-- (post/pre-fix)	Unary	Yes	A-- or --A	Value/reference

Relational operators

Operator	Type	Overload	Format	Returns
<code>==</code>	Binary	Yes	<code>A == B</code>	bool or Value/reference
<code>></code>	Binary	Yes	<code>A > B</code>	bool or Value/reference
<code><</code>	Binary	Yes	<code>A < B</code>	bool or Value/reference
<code><=</code>	Binary	Yes	<code>A <= B</code>	bool or Value/reference
<code>>=</code>	Binary	Yes	<code>A >= B</code>	bool or Value/reference
<code>!=</code>	Binary	Yes	<code>A != B</code>	bool or Value/reference

Logical operators

Operator	Type	Overload	Format	Returns
<code>&&</code>	Binary	Yes	<code>A && B</code>	bool or Value/reference
<code> </code>	Binary	Yes	<code>A B</code>	bool or Value/reference
<code>!</code>	Unary	Yes	<code>!</code>	bool or Value/reference

Bitwise operators

Operator	Type	Overload	Format	Returns
&	Binary	Yes	A & B	Value/reference
	Binary	Yes	A B	Value/reference
^	Binary	Yes	A ^ B	Value/reference
<<	Binary	Yes	A << B	Value/reference
>>	Binary	Yes	A >> B	Value/reference
~	Unary	Yes	~A	Value/reference

Assignment operators

Operator	Type	Overload	Format	Returns
=	Binary	Yes	A = B	Value/reference
+=	Binary	Yes	A += B	Value/reference
-+	Binary	Yes	A -= B	Value/reference
*=	Binary	Yes	A *= B	Value/reference
/=	Binary	Yes	A /= B	Value/reference
%=	Binary	Yes	A %= B	Value/reference
>>=	Binary	Yes	A >>= B	Value/reference
<<=	Binary	Yes	A <<= B	Value/reference
&=	Binary	Yes	A &= B	Value/reference
^=	Binary	Yes	A ^= B	Value/reference
=	Binary	Yes	A = B	Value/reference

Operators (others)

Operator	Type	Overload	Format	Returns
sizeof	Unary	No	sizeof(A)	Value
new	Unary	Yes	new A	pointer (A*)
delete	Unary	Yes	delete A	<None>
Condition (?)	Ternary	No	C ? A:B	A or B depending on the evaluation of C
:: (scope)		No	A::B	
Cast (type)	Binary	Yes	(A)B or A(B)	B casted to A
-> (pointer)	Binary	Yes	A->B	B from A
. (member)	Binary	Yes	A.B	B from A
[] (index)	Binary	Yes	A[B]	Value/reference
() (function)	Multi	Yes	A(B,C,...)	Value/reference
, (list)	Binary	Yes	(A,B)	Val/ref for (A follow by B)

Operators (evaluation order)

- 1. :: (scope)
- 2. () [] -> . ++ - -
- 3. + - ! ~ ++ - - (type)* & sizeof
- 4. * / %
- 5. + -
- 6. << >>
- 7. < <= > >=
- 8. == !=
- 9. &
- 10. ^
- 11. |
- 12. &&
- 13. ||
- 14. ?:
- 15. = += -= *= /= %= >>= <<= &= ^= |=
- 16. ,

A decorative graphic in the top-left corner consists of several overlapping blue triangles of varying shades, creating a geometric pattern.

Operators for ► classes

Operators for classes

- ▶ A class can define a series of special functions that behave exactly as an operator - that is to allow the program to explain how the compiler should understand certain operations between classes
- ▶ Use keyword: “*operator*”
- ▶ These functions can have various access operators (and they comply to rules imposed by the operators - if an operator is declared private then it can only be accessed within the class)
- ▶ Operators can be implemented outside the classes - in this case, if it is needed, they can be declared as “*friend*” functions in order to be accessed by private members from a class

Operators for classes

- In this case the *operator+* is overloaded allowing addition operation between an *Integer* and another *Integer*

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (const Integer &i);
};

int Integer::operator+(const Integer &i)
{
    return value + i.value;
}

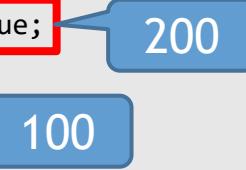
void main()
{
    Integer n1(100);
    Integer n2(200);
    int x = n1 + n2;
}
```

Operators for classes

- ▶ In this case the *operator+* is overloaded allowing addition operation between an *Integer* and another *Integer*
- ▶ The addition operation is applied for the left parameter, the right parameter being the argument. In other words: “*n1+n2*” ⇔ “*n1.operator+(n2)*”

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (const Integer &i);
};
int Integer::operator+(const Integer &i)
{
    return value + i.value;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    int x = n1 + n2;
}
```



Operators for classes

- ▶ Parameters don't have to be a const or a reference. Using the operator is similar to using a function (all of the promotion rules apply).
- ▶ It is however recommended to use const references when the result of an operator does not modify the arguments

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (Integer &i);
};
int Integer::operator+(Integer &i)
{
    return value + i.value;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    int x = n1 + n2;
}
```

Operators for classes

- ▶ Similarly, the return value does not have a predefine type (e.g. while the usual understanding is that adding, multiplying, etc of two values of the same type will produce a result of the same type, this is not mandatory).

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer operator+ (Integer i);
};

Integer Integer::operator+(Integer i)
{
    Integer res(value+i.value);
    return res;
}

void main()
{
    Integer n1(100);
    Integer n2(200);
    Integer n3(0);
    n3 = n1 + n2;
}
```

Operators for classes

- ▶ Operators work as a function. They also can be overloaded.
- ▶ In this case the Integer class supports an addition operation between two Integer objects, or between an Integer object and a float variable

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (const Integer &i) { ... };
    int operator+(float nr);
};

int Integer::operator+(float nr)
{
    return value + (int)nr;
}

void main()
{
    Integer n1(100);
    Integer n2(200);
    int x = n1 + n2;
    int y = n1 + 1.2f;
}
```

Operators for classes

- ▶ Operators work as a function. They also can be overloaded.
- ▶ In this case the code does not compile because there already is an “*operator+*” function with a “float” parameter

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (const Integer &i) { ... };
    int operator+(float nr);
    float operator+(float nr);
};
int Integer::operator+(float nr)
{
    return value + (int)nr;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    int x = n1 + n2;
    int y = n1 + 1.2f;
}
```

Operators for classes

- ▶ Pay attention at operators' usage order and don't assume bijection. In this case the code does NOT compile. The Integer class handles the addition between an Integer and a float, but not the other way around (between a float and an Integer). This is not possible with a function from the class.

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (const Integer &i) { ... };
    int operator+(float nr);
};
int Integer::operator+(float nr)
{
    return value + (int)nr;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    int x = n1 + n2;
    int y = 1.2f + n1; // Error line
}
```

error C2677: binary '+': no global operator found which takes type 'Integer' (or there is no acceptable conversion)

Operators for classes

- ▶ The code compiles - friend functions solve both cases (*Integer+float* and *float+Integer*)

App.cpp

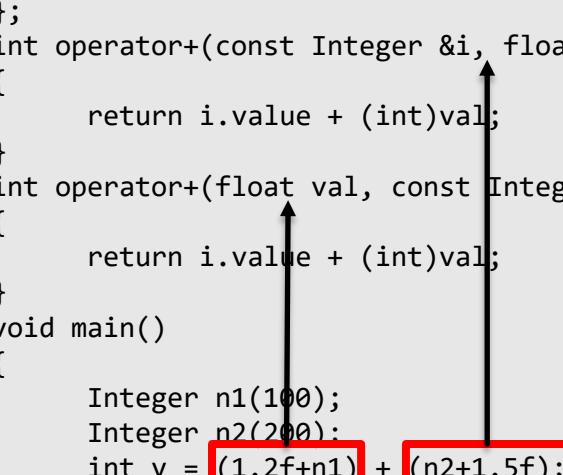
```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend int operator+ (const Integer &i, float val);
    friend int operator+ (float val, const Integer &i);
};
int operator+(const Integer &i, float val)
{
    return i.value + (int)val;
}
int operator+(float val, const Integer &i)
{
    return i.value + (int)val;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    int y = (1.2f+n1)+(n2+1.5f);
}
```

Operators for classes

- ▶ The code compiles - friend functions solve both cases (*Integer+float* and *float+Integer*)

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend int operator+ (const Integer &i, float val);
    friend int operator+ (float val, const Integer &i);
};
int operator+(const Integer &i, float val)
{
    return i.value + (int)val;
}
int operator+(float val, const Integer &i)
{
    return i.value + (int)val;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    int y = (1.2f+n1) + (n2+1.5f);
}
```



Operators for classes

- ▶ This code will NOT compile. There are two operators defined (one as part of the class, and the other one as a friend function), both of them referring to the same operation (*Integer + Integer*).

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator+ (Integer i);
    friend int operator+ (Integer n1, Integer n2);
};
int Integer::operator+(Integer i)
{
    return this->value + i.value;
}
int operator+ (Integer n1, Integer n2)
{
    return n1.value + n2.value;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    Integer n3(0);
    n3 = n1 + n2;
}
```

error C2593: 'operator +' is ambiguous
note: could be 'int Integer::operator +(Integer)'
note: or 'int operator +(Integer, Integer)'

Operators for classes

- Relational operators are defined exactly as the arithmetic ones. From the compiler point of view, there is no real difference between those two.

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator > (const Integer & i);
};

bool Integer::operator > (const Integer & i)
{
    if (value > i.value)
        return true;
    return false;
}

void main()
{
    Integer n1(100);
    Integer n2(200);
    if (n2 > n1)
        printf("n2 mai mare ca n1");

}
```

Operators for classes

- Relational operators do not need to return a bool even though this is what is expected from them. Keep in mind that the compiler does not differentiate between arithmetic or logical operator. In this case, the *operator>* returns an object.

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer operator > (const Integer & i);
    void PrintValue();
};
void Integer::PrintValue()
{
    printf("Value is %d", value);
}
Integer Integer::operator > (const Integer & i)
{
    Integer res(this->value + i.value);
    return res;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    (n1 > n2).PrintValue();
}
```

Operators for classes

- ▶ The same logic applies for logical operators as well (from the compiler point of view they are not different from the arithmetic or relational operators).

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer operator && (const Integer & i)
    void PrintValue();
};
void Integer::PrintValue()
{
    printf("Value is %d", value);
}
Integer Integer::operator && (const Integer & i)
{
    Integer res(this->value + i.value);
    return res;
}
void main()
{
    Integer n1(100);
    Integer n2(200);
    (n1 && n2).PrintValue();
}
```

Operators for classes

- ▶ An unary operator does not have a parameter (if it is defined within the class) or *one* parameter if it is defined as a “friend” function.
- ▶ Similar to the binary operators, there is no restriction for what these methods return. In the case below x will have the value 80.

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator ! () ;
};

int Integer::operator ! ()
{
    return 100 - this->value;
}

void main()
{
    Integer n1(20);
    int x = !n1;
}
```

Operators for classes

- ▶ The presented methods can be applied in the same way for the following operators:

+	-	*	/	%	>	<	>=	<=	!=
==	&		&&		^	!	~		

- ▶ For these cases, it is recommended to use *friend* functions and not to create methods of the class within the class
- ▶ It is indicated, as much as possible, to add such functions with parameter combinations (class with int, int with class, class with double, double with class, etc)
- ▶ The operators can also return objects and/or references to an object. In these cases the returned object is then further used in the evaluation of the expression of which it is a part of.

Operators for classes

- ▶ In case of assignment, it is recommended to return a reference to the object that gets a value assign to. This will allow that reference to be further used in other expression.
- ▶ There is a special assignment operator called ***move assignment*** that can be used with a parameter that is a temporary reference (“`&&`”)

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer& operator = (int val);
};
Integer& Integer::operator = (int val)
{
    value = val;
    return (*this);
}
void main()
{
    Integer n1(20);
    n1 = 20;
}
```

Operators for classes

- ▶ However, it is NOT mandatory to return a reference. The code below returns a bool value.
- ▶ After the execution of the code, *n1.value* will be 30, and *res* will be true

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator = (int val);
};
bool Integer::operator = (int val)
{
    value = val;
    return (val % 2) == 0;
}
void main()
{
    Integer n1(20);
    bool res = (n1 = 30);
}
```

Operators for classes

- ▶ Some operators (operator=, operator[], operator(), operator->) can not be a static function (be used outside the class through **friend** specifier).
- ▶ This case will not compile.

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend bool operator= (Integer &i, int val) { i.value = val; return false; }
};

void main()
{
    Integer n1(20);
    bool res = (n1 = 30);
}
```

error C2801: 'operator =' must be a non-static member

Operators for classes

- ▶ However, the rest of assignment operators (`+=`, `-=`, `*=`, etc) can be implemented in this way.
- ▶ In this case the code compiles, `res` will have the value true and `n1.value` will be 30

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend bool operator += (Integer & i, int val);
};
bool operator += (Integer &i, int val)
{
    i.value = val;
    return true;
}
void main()
{
    Integer n1(20);
    bool res = (n1 += 30);
}
```

Operators for classes

- ▶ Be careful when using references and when a value. In this case the “*operator+*” is called, but with a copy of the class Integer. As a result, the value of *n1.value* will NOT change (will remain 20).

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend bool operator += (Integer i, int val);
};
bool operator += (Integer i, int val)
{
    i.value = val;
    return true;
}
void main()
{
    Integer n1(20);
    bool res = (n1 += 30);
}
```

Operators for classes

- ▶ Since friend functions are allowed, the order of the parameters can be changed. In this case, the code compiles even if “`30 &= ...`” does not make any sense.

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend bool operator &= (int val, Integer i);
};
bool operator &= (int val, Integer i)
{
    i.value = val;
    return true;
}
void main()
{
    Integer n1(20);
    bool res = (30 &= n1);
}
```

Operators for classes

- ▶ A different case refers to postfix/prefix operators (++ and --)

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator++ ();
    bool operator++ (int value);
};
bool Integer::operator++ ()
{
    value++;
    return true;
}
bool Integer::operator++ (int val)
{
    value += 2;
    return false;
}
void main()
{
    Integer n1(20);
    bool res = (n1++);
}
```

Operators for classes

- In this case, the postfix form is being executed (`n1.value = 22, res = false`)

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator++ ();
    bool operator++ (int value);
};
bool Integer::operator++ ()
{
    value++;
    return true;
}
bool Integer::operator++ (int val)
{
    value += 2;
    return false;
}
void main()
{
    Integer n1(20);
    bool res = (n1++);
}
```

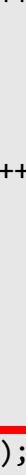
val=0, must be int type

Operators for classes

- In this case the prefix form is being executed (`n1.value = 21, res = true`)

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator++ ();
    bool operator++ (int value);
};
bool Integer::operator++ ()
{
    value++;
    return true;
}
bool Integer::operator++ (int val)
{
    value += 2;
    return false;
}
void main()
{
    Integer n1(20);
    bool res = (++n1);
}
```



Operators for classes

- ▶ Prefix/postfix operators can be “*friend*” functions. Normally the first parameter of the friend function has to be a reference type. After execution n1.value = 22, res = false

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend bool operator++ (Integer &i);
    friend bool operator++ (Integer &i,int value);
};
bool operator++ (Integer &i)
{
    i.value++;
    return true;
}
bool operator++ (Integer &i,int val)
{
    i.value += 2;
    return false;
}
void main()
{
    Integer n1(20);
    bool res = (n1++);
}
```

Operators for classes

- ▶ Postfix/prefix operators have a special meaning
 - ▶ PostFix - the value is returned first and then the operation is executed
 - ▶ Prefix - the operation is executed first and then the value is returned

```
App.cpp
void main()
{
    int x = 3;
    int y;
    [y = x++;  
    int z;
    z = ++x;
}
```

- ▶ In the first case y takes x's value and then the increment operation for x is being done. Meaning that y will be equal with 3 and x with 4.

```
mov    eax,dword ptr [x]
mov    dword ptr [y],eax
mov    ecx,dword ptr [x]
add    ecx,1
mov    dword ptr [x],ecx
```

Operators for classes

- ▶ Postfix/prefix operators have a special meaning
 - ▶ PostFix - the value is returned first and then the operation is executed
 - ▶ Prefix - the operation is executed first and then the value is returned

App.cpp

```
void main()
{
    int x = 3;
    int y;
    y = x++;
    int z;
    z = ++x;
}
```

- ▶ In the first case y takes x's value and then the increment operation for x is being done. Meaning that y will be equal with 3 and x with 4.
- ▶ In the second case, first the increment operation for x is done and then the assignation towards z, meaning that z will be equal with 5 and x also with

mov	eax,dword ptr [x]
add	eax,1
mov	dword ptr [x],eax
mov	ecx,dword ptr [x]
mov	dword ptr [z],ecx

Operators for classes

- ▶ Prefix/postfix operators can be modified to have the desired behavior (postfix, prefix) in the following way:

App.cpp

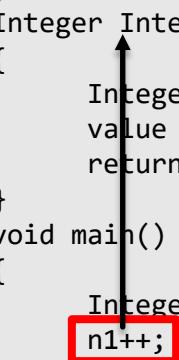
```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer& operator++ ();
    Integer operator++ (int value);
};
Integer& Integer::operator++ ()
{
    value += 1;
    return (*this);
}
Integer Integer::operator++ (int)
{
    Integer tempObject(value);
    value += 1;
    return (tempObject);
}
void main()
{
    Integer n1(20);
    n1++;
}
```

Operators for classes

- ▶ Prefix/postfix operators can be modified to have the desired behavior (postfix, prefix) in the following way:

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer& operator++ ();
    Integer operator++ (int value);
};
Integer& Integer::operator++ ()
{
    value += 1;
    return (*this);
}
Integer Integer::operator++ (int)
{
    Integer tempObject(value);
    value += 1;
    return (tempObject);
}
void main()
{
    Integer n1(20);
    n1++;
}
```

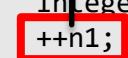
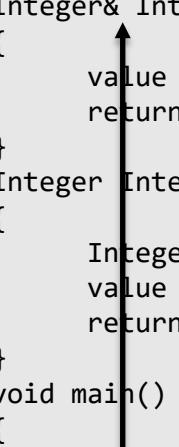


Operators for classes

- ▶ Prefix/postfix operators can be modified to have the desired behavior (postfix, prefix) in the following way:

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer& operator++ ();
    Integer operator++ (int value);
};
Integer& Integer::operator++ ()
{
    value += 1;
    return (*this);
}
Integer Integer::operator++ (int)
{
    Integer tempObject(value);
    value += 1;
    return (tempObject);
}
void main()
{
    Integer n1(20);
    ++n1;
}
```



Operators for classes

- ▶ A special operator is **new**. **new** has a special format (it has to return **void*** and has a **size_t** first parameter).
- ▶ The **new** operator cannot be used as a friend function.
- ▶ The **size_t** parameter represents the size of the object to be allocated.
- ▶ The **new** operator does not call the constructor, it is expected to allocate memory for the current object. In this case, after execution **GlobalValue = 100**. The constructor (if any) will be called automatically by the compiler after the memory has been allocated.

App.cpp

```
int GlobalValue = 0;
class Integer {
    int value;
public:
    Integer(int val) : value(val) {}
    void* operator new(size_t t);
};
void* Integer::operator new (size_t t) {
    return &GlobalValue;
}
void main() {
    Integer *n1 = new Integer(100);
}
```

Operators for classes

- ▶ If *new operator* is declared with multiple parameters, they can be called/used in the following way:

App.cpp

```
int GlobalValue = 0;

class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    void* operator new(size_t t,int value);
};

void* Integer::operator new (size_t t,int value)
{
    return &GlobalValue;
}

void main()
{
    Integer *n1 = new(100) Integer(123);
}
```

- ▶ The functions/methods that overload *new* with multiple parameters are also called *placement new*

Operators for classes

- ▶ The new [] operator has a similar behavior. It is used for allocating multiple objects. It has a similar format: it must return a **void*** and the first parameter is also a **size_t** (that represent the amount of memory needed for all of the elements in the vector).
- ▶ For the following example to work, a default constructor is required. After the execution, all elements from GlobalValue have their value equal to 1.

App.cpp

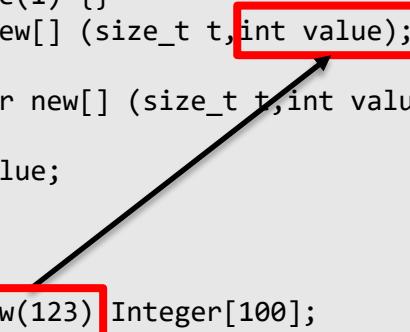
```
int GlobalValue[100];
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer() : value(1) {}
    void* operator new [] (size_t t);
};
void* Integer::operator new[] (size_t t) { return &GlobalValue[0]; }
void main()
{
    Integer *n1 = new Integer[100];
}
```

Operators for classes

- ▶ `new[]` operator can also have several parameters. The following example shows an example on how such construct can be used.

App.cpp

```
int GlobalValue[100];
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    Integer() : value(1) {}
    void* operator new[] (size_t t,int value);
};
void* Integer::operator new[] (size_t t,int value)
{
    return &GlobalValue;
}
void main()
{
    Integer *n1 = new(123) Integer[100];
}
```



Operators for classes

- ▶ Generally speaking, the normal behavior for operators that assure the allocation is the following:

Operator
void* operator new (size_t size)
void* operator new[] (size_t size)
void operator delete (void* object)
void operator delete[] (void* objects)

- ▶ It is recommended for the new and delete operators to throw exceptions

Operators for classes

- ▶ Another special operator is the `cast` operator
- ▶ This operator allows the transformation of an object from one type to another
- ▶ Being a casting operator, we do not have to provide the return type (it is considered the type we are casting to) → in the next example: *float*

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    operator float();
};
Integer::operator float()
{
    return float(value * 2);
}
void main()
{
    Integer n1(2);
    float f = (float)n1;
}
```

Operators for classes

- ▶ Cast operators are also used when such a conversion is explicitly required.
- ▶ As in the previous case, the value for f will be 4.0
- ▶ Cast operators cannot be used with friend specifier

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    operator float();
};
Integer::operator float()
{
    return float(value * 2);
}
void main()
{
    Integer n1(2);
    float f = n1;
}
```

Operators for classes

- ▶ Make sure to pay attention to all operators. In this case : $f = 4.2$

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    operator float();
};
Integer::operator float()
{
    return float(value * 2);
}
void main()
{
    Integer n1(2);
    float f = n1 + 0.2f;
}
```

Operators for classes

- ▶ However, in this case $f = 20.2$ due to the addition operator (**operator+**)

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    operator float();
    float operator + (float f);
};
Integer::operator float()
{
    return float(value * 2);
}
float Integer::operator+ (float f)
{
    return value * 10.0f + f;
}
void main()
{
    Integer n1(2);
    float f = n1+0.2f;
}
```

Operators for classes

- ▶ The indexing operators allow the usage of [] for a certain object.
- ▶ They have only one restriction and that is that they only have one parameter - but this parameter can be anything and the return value also can be of any kind. Also, the indexing operator cannot be a friend function/ outside the current object
- ▶ In this case, `ret=true` because byte 1 from `n1.value` is set

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator [](int index);
};
bool Integer::operator [](int index)
{
    return (value & (1 << index)) != 0;
}
void main()
{
    Integer n1(2);
    bool ret = n1[1];
}
```

Operators for classes

- ▶ The following example uses a different key (a *const char **) for the index operator []/

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator [](const char *name);
};
bool Integer::operator [](const char *name)
{
    if ((strcmp(name, "first") == 0) && ((value & 1) != 0))
        return true;
    if ((strcmp(name, "second") == 0) && ((value & 2) != 0))
        return true;
    return false;
}
void main()
{
    Integer n1(2);
    bool ret = n1["second"];
}
```

Operators for classes

- ▶ One can also overload the index operator [] (to be used with different keys). The following example uses *operator[]* with both *int* and *const char ** keys.

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator [](const char *name);
    bool operator [](int index);
};
bool Integer::operator [](int index)
{
    ...
}
bool Integer::operator [](const char *name)
{
    ...
}
void main()
{
    Integer n1(2);
    bool ret = n1["second"];
    bool v2 = n1[2];
}
```

Operators for classes

- ▶ The function call operator (*operator()*) works almost the same as the indexing operator.
- ▶ Like the indexing operator, the function call operator () can only be a member function within the class

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    bool operator ()(int index);
};
bool Integer::operator ()(int index)
{
    return (value & (1 << index)) != 0;
}
void main()
{
    Integer n1(2);
    bool ret = n1(1);
}
```

Operators for classes

- ▶ The main difference between index operator (`operator[]`) and function call operator (`operator()`) is that function call operator can have multiple parameters (or none).

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator ()(int start,int end);
};
int Integer::operator ()(int start,int end)
{
    return (value >> start) & ((1 << (end - start)) - 1);
}

void main()
{
    Integer n1(122);
    int res = n1(1,3);
}
```

Operators for classes

- In this example, the function call operator (*operator()*) is used without any parameter:

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator ()();
};

int Integer::operator }()
{
    return (value*2);
}

void main()
{
    Integer n1(122);
    int res = n1();
}
```

Operators for classes

- ▶ The member access operator (*operator →*) can also be overwritten.
- ▶ In this case, even though there are no restrictions imposed by the compiler, this operator has to return a pointer to an object.

App.cpp

```
class MyData
{
    float value;
public:
    void SetValue(float val) { value = val; }
};

class Integer
{
    MyData data;
public:
    MyData* operator -> ();
};

MyData* Integer::operator ->()
{
    return &data;
}

void main()
{
    Integer n1;
    n1->SetValue(100);
}
```

Operators for classes

- ▶ *operator→* has to be used with an object (NOT a pointer). The following example will not compile as *n2* (a pointer) does not have a data member called *SetValue*.

App.cpp

```
class MyData
{
    float value;
public:
    void SetValue(float val) { value = val; }
};

class Integer
{
    MyData data;
public:
    MyData* operator -> ();
};

MyData* Integer::operator ->()
{
    return &data;
}

void main()
{
    Integer n1;
    Integer *n2 = &n1;
    n2->SetValue(100);
}
```

Operators for classes

- ▶ However, if we convert the pointer to an object, the **operator→** will work.
- ▶ The “**→**” operator can be defined only in a class (it cannot be defined outside the class as a friend function)

App.cpp

```
class MyData
{
    float value;
public:
    void SetValue(float val) { value = val; }
};

class Integer
{
    MyData data;
public:
    MyData* operator -> ();
};

MyData* Integer::operator ->()
{
    return &data;
}

void main()
{
    Integer n1;
    Integer *n2 = &n1;
    (*n2)->SetValue(100);
}
```

Operators for classes

- ▶ Other operators that behave in the same way as *operator* → are:
 - ❖ . (A.B)
 - ❖ ->* (A->*B)
 - ❖ .* (A.*B)
 - ❖ * (*A)
 - ❖ & (&A)

Operators for classes

- ▶ The list operator “*operator*,” is used in case of lists
- ▶ For example, the following list is evaluated from left to right and without a specific operator, it returns the last value::

```
int x = (10,20,30,40)
```

- ▶ The evaluation is done as follows:
 - ❖ First evaluated is the expression “10,20” → which returns 20
 - ❖ The following expression which is evaluated is “20,30” (20 returned from the previous expression) which returns 30
 - ❖ And finally, it is evaluated “30,40” which will return 40

Operators for classes

- ▶ In this case, the “*operator*,” is called first, for n1 and 2.5f, which returns 75 ($30 * 2.5 = 75$)
- ▶ *res* local variable will have the value 75

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator , (float f);
};
int Integer::operator ,(float f)
{
    return (int)(value*f);
}

void main()
{
    Integer n1(30);
    int res = (n1, 2.5f);
}
```

Operators for classes

- In this case the “*operator*,” is called first for *n1* and *2.5f*, which returns the value $30*2.5=75$, then the default “,” operator for 75 and 10 is applied that returns 10.

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    int operator , (float f);
};
int Integer::operator ,(float f)
{
    return (int)(value*f);
}

void main()
{
    Integer n1(30);
    int res = (n1, 2.5f, 10);
}
```

Operators for classes

- ▶ It is recommended to use *friend* specifier to explain several combinations that can be found when using the list operator (*operator,*)

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend int operator , (Integer&, float f);
    friend int operator , (int value, Integer&);
};
int operator , (Integer& i, float f)
{
    return (int)(i.value*f);
}
int operator , (int value, Integer &i)
{
    return value + i.value;
}
void main()
{
    Integer n1(30);
    int res = (n1, 2.5f, n1);
}
```

Operators for classes

- ▶ It is recommended to use *friend* specifier to explain several combinations that can be found when using the list operator (*operator,*)

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend int operator , (Integer&, float f);
    friend int operator , (int value, Integer&);
};
int operator , (Integer& i, float f)
{
    return (int)(i.value*f);
}
int operator , (int value, Integer &i)
{
    return value + i.value;
}
void main()
{
    Integer n1(30);
    int res = (n1, 2.5f, n1);
}
```

(n1,2.5f)=75

Operators for classes

- ▶ It is recommended to use *friend* specifier to explain several combinations that can be found when using the list operator (*operator,*)

App.cpp

```
class Integer
{
    int value;
public:
    Integer(int val) : value(val) {}
    friend int operator , (Integer&, float f);
    friend int operator , (int value, Integer&);
};
int operator , (Integer& i, float f)
{
    return (int)(i.value*f);
}
int operator , (int value, Integer &i)
{
    return value + i.value;
}
void main()
{
    Integer n1(30);
    int res = (n1, 2.5f, n1);
}
```

(75,n1) = 105

Operators for classes

- When overloading an operator, some optimizations that compiler is doing (such as lazy evaluation) will be lost.

App.cpp

```
class Bool {  
    bool value;  
    const char * name;  
public:  
    Bool(bool val, const char * nm) : value(val), name(nm) {}  
    Bool operator|| (const Bool &i) {  
        bool res = value || i.value;  
        printf("Compute bool(%s and %s)=>%s\n", name, i.name, res ? "true" : "false");  
        Bool b(res, "temp");  
        return b;  
    };  
    operator bool() {  
        printf("Return bool for %s => %s\n", this->name,value?"true":"false");  
        return value;  
    }  
};  
int main() {  
    Bool n1(true,"n1");  
    Bool n2(false,"n2");  
    Bool n3(false,"n3");  
    bool res = ((bool)n1) || ((bool)n2) || ((bool)n3);  
    return 0;  
}
```

Output:

Return bool for n1 => true

Once **n1** is evaluated and it is **true**, the rest of the evaluation is skipped. Lazy evaluation for **||** operator is applied and the cast operators for **n2** and **n3** are **not** called anymore.

Operators for classes

- When overloading an operator, some optimizations that compiler is doing (such as lazy evaluation) will be lost.

App.cpp

```
class Bool {  
    bool value;  
    const char * name;  
public:  
    Bool(bool val, const char * nm) : value(val), name(nm) {}  
    Bool operator|| (const Bool &i) {  
        bool res = value || i.value;  
        printf("Compute bool(%s || %s)=>%s\n", name, i.name, res ? "true" : "false");  
        Bool b(res, "temp");  
        return b;  
    };  
    operator bool() {  
        printf("Return bool for %s => %s\n", this->name,value?"true":"false");  
        return value;  
    }  
};  
int main() {  
    Bool n1(true,"n1");  
    Bool n2(false,"n2");  
    Bool n3(false,"n3");  
    bool res = n1 || n2 || n3;  
    return 0;  
}
```

Output:

Compute bool(n1 || n2)=>true
Compute bool(temp || n3)=>true
Return bool for temp => true

However, in this case since we have used our own
operator|| the lazy evaluation will not be applied,
and the entire expression will be evaluated.

A decorative graphic in the top-left corner consists of several overlapping blue triangles of varying shades, creating a layered effect.

Operations with objects

Object operations

- ▶ Let's analyze the following code:

App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                      Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
};
void Set(Date d,int value)
{
    d.SetX(value);
}
void main()
{
    Date d(1);
    Set(d,123);
}
```

```
push    123
sub    esp,10h
       10h = 16 = sizeof(Data)

mov    eax,esp
mov    ecx,dword ptr [d.X]
mov    dword ptr [eax],ecx
mov    edx,dword ptr [d.Y]
dword ptr [eax+4],edx
mov    ecx,dword ptr [d.Z]
dword ptr [eax+8],ecx
mov    edx,dword ptr [d.T]
dword ptr [eax+0Ch],edx
call   Set
add    esp,14h
```

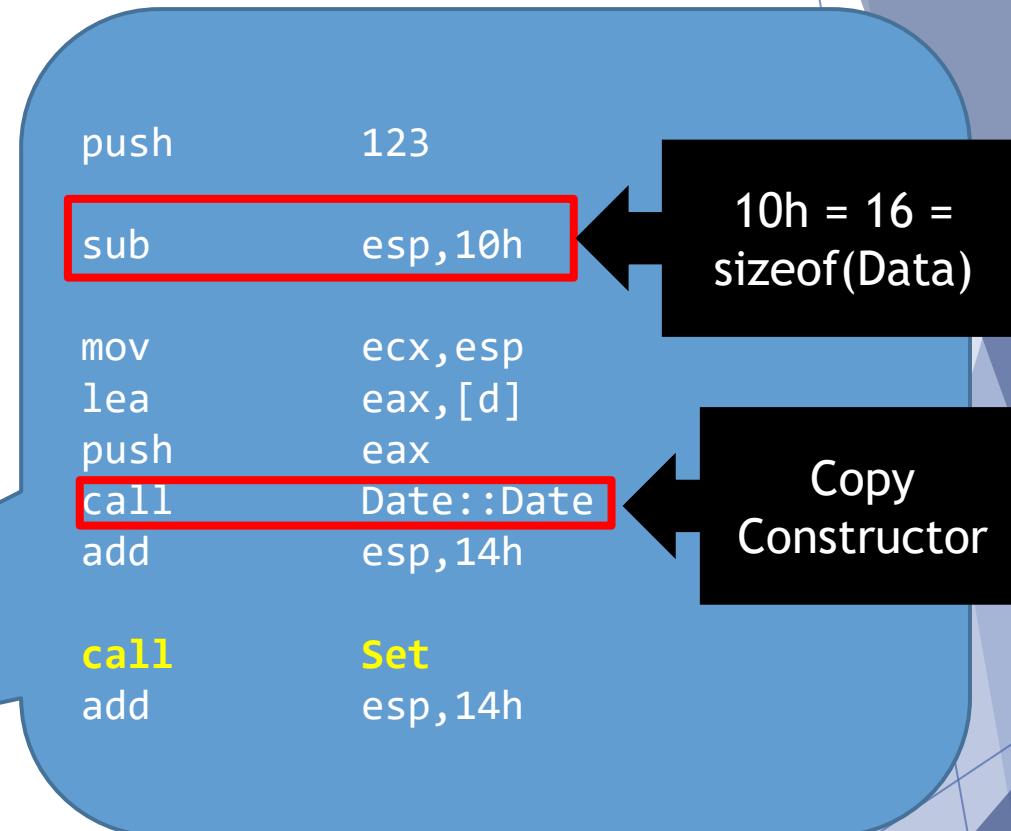
- ▶ In this case, as there is no copy-constructor the compiler copies the entire object “d” byte-by-byte similar to what *memcpy* function does.

Object operations

- ▶ Let's analyze the following code:

App.cpp

```
class Date
{
    int X, Y, Z, T;
public:
    Date(int value) : X(value), Y(value + 1),
                      Z(value + 2), T(value + 3) {}
    Date(const Date & d) : X(d.X), Y(d.Y),
                           Z(d.Z), T(d.T) {}
    void SetX(int value) { X = value; }
};
void Set(Date d, int value)
{
    d.SetX(value);
}
void main()
{
    Date d(1);
    Set(d, 123);
}
```



- ▶ However, if a copy-constructor exists, that method will be called to copy the object "d" into the stack.

Object operations

- ▶ Let's analyze the following code:

App.cpp

```
class Date
{
    int X, Y, Z, T;
public:
    Date(int value) : X(value), Y(value + 1),
                      Z(value + 2), T(value + 3) {}
    Date(Date && d) : X(d.X), Y(d.Y),
                      Z(d.Z), T(d.T) {}
    void SetX(int value) { X = value; }
};
void Set(Date d, int value)
{
    d.SetX(value);
}
void main()
{
    Date d(1);
    Set(d, 123);
}
```

Move
Constructor

error C2280: 'Date::Date(const Date &)': attempting to reference a deleted
function
note: compiler has generated 'Date::Date' here
note: 'Date::Date(const Date &)': function was implicitly deleted because 'Date'
has a user-defined move constructor

- ▶ If the move constructor is present, but not a copy constructor, the code will fail at compile time. Adding a copy-constructor will make this code work.

Object operations

- ▶ Let's analyze the following code:

App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                      Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
};
void Set(Date &d,int value)
{
    d.SetX(value);
}
void main()
{
    Date d(1);
    Set(d,123);
}
```

push 123
lea eax,[d]
push eax
call Set
add esp,8

- ▶ If we send the object via a reference or pointer, the copy-constructor is no longer required.

Object operations

When a parameter is given to a function, we have the following cases:

- ▶ If the parameter is *reference/pointer* - only its address is copied on the stack
- ▶ If the parameter is an *object*, all of that object is copied on the stack and it can be accessed as any other parameter that was copied on the stack (relative at [EBP+xxx]). The compiler uses the copy-constructor to copy an object into the stack. If no copy constructor is present, a *memcpy* - like code is generated (a code that copies byte-by-byte the entire content of the object into the stack).

Object operations

- ▶ Let's analyze the following code:

App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                      Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
};
Date Get(int value)
{
    Date d(value);
    return d;
}
void main()
{
    Date d(1);
    d = Get(100);
}
```

```
Date Get(int value)
{
    push    ebp
    mov     ebp,esp
    Date d(value);
    mov     eax,dword ptr [value]
    push    eax
    lea     ecx,[d]
    call    Date::Date
    return d;
    mov     eax,dword ptr [ebp+8]
    mov     ecx,dword ptr [d.X]
    mov     dword ptr [eax],ecx
    mov     edx,dword ptr [d.Y]
    mov     dword ptr [eax+4],edx
    mov     ecx,dword ptr [d.Z]
    mov     dword ptr [eax+8],ecx
    mov     edx,dword ptr [d.T]
    mov     dword ptr [eax+12],edx
    eax,dword ptr [ebp+8]
}

    mov     esp,ebp
    pop    ebp
    ret
```

- ▶ In this case, function *Get* returns an object (NOT a reference) of type *Date*.

Object operations

- ▶ Let's analyze the following code :

App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                      Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
};

Date Get(int value)
{
    Date d(value);
    return d;
}

void main()
{
    Date d(1);
    d = Get(100);
}
```

- ▶ In this case, function *Get* returns an object (NOT a reference) of type *Date*.

Object operations

- ▶ Let's analyze the following code :

App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                      Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
};
Date Get(int value)
{
    Date d(value);
    return d;
}
void main()
{
    Date d(1);
    d = Get(100);
}
```

App.pseudocode

```
class Date
{
    ...
};

Date* Get(Date *tempObject, int value)
{
    Date d(value);
    memcpy(tempObject,&d,sizeof(Date));
    return tempObject;
}
void main()
{
    Date d(1);
    unsigned char temp[sizeof(Date)]
    Date* tmpObj = Get(temp,100);
    memcpy(d,tmpObj,sizeof(Date))
}
```

Object operations

- ▶ Let's analyze the following code:

App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                      Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
    Date(const Date & obj) { X = obj.X; }
};
Date Get(int value)
{
    Date d(value);
    return d;
}
void main()
{
    Date d(1);
    d = Get(100);
}
```

App.pseudocode

```
class Date
{
    ...
};

Date* Get(Date *tempObject, int value)
{
    Date d(value);
    tempObject->Date(d);
    return tempObject;
}

void main()
{
    Date d(1);
    unsigned char temp[sizeof(Date)]
    Date* tmpObj = Get(temp,100);
    memcpy(d,tmpObj,sizeof(Date))
}
```

Object operations

- ▶ Let's analyze the following code:

App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                      Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
    Date(const Date & obj) { X = obj.X; }
    Date& operator = (const Date &d)
    {
        X = d.X;
        return (*this);
    }
    Date Get(int value)
    {
        Date d(value);
        return d;
    }
void main()
{
    Date d(1);
    d = Get(100);
}
```

App.pseudocode

```
class Date
{
    ...
};

Date* Get(Date *tempObject, int value)
{
    Date d(value);
    tempObject->Date(d);
    return tempObject;
}

void main()
{
    Date d(1);
    unsigned char temp[sizeof(Date)]
    Date* tmpObj = Get(temp,100);
    d.operator=(*tmpObj);
}
```

Object operations

- ▶ Let's analyze the following code:

App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                      Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
    Date(const Date & obj) { X = obj.X; }
    Date(const Date && obj) { X = obj.X; } // Line highlighted
    Date& operator = (const Date &d)
    {
        X = d.X;
        return (*this);
    }
};

Date Get(int value)
{
    Date d(value);
    return d;
}

void main()
{
    Date d(1);
    d = Get(100);
}
```

App.pseudocode

```
class Date
{
    ...
};

Date* Get(Date *tempObject, int value)
{
    Date d(value);
    tempObject->Date(d);
    return tempObject;
}

void main()
{
    Date d(1);
    unsigned char temp[sizeof(Date)]
    Date* tmpObj = Get(temp,100);
    d.operator=(*tmpObj);
}
```

If present, the move constructor is used when returning an object from a function !

Object operations

When dealing with temporary object, the compiler will always prefer:

1. Move constructor
2. Move assignment

Instead of copy constructor or simple assignment.

These methods (move constructor / move assignment) are preferred if they exist. If they are not specified, the copy-constructor and assignment operator will be used (if any).

**Move constructor and Move assignment are only used for temporary object.
For a regular object (reference) copy-constructor and assignment operator
are preferred (if they exists).**

Object operations

- ▶ Let's consider the following class:

App.cpp

```
class String {  
    char * text;  
    void CopyString(const char * string) {  
        text = new char[strlen(string) + 1];  
        memcpy(text, string, strlen(string) + 1);  
    }  
public:  
    String(const char * s) {  
        CopyString(s);  
        printf("CTOR => Obj:%p,Text:%p\n", this, text);  
    }  
    ~String() {  
        if (text != nullptr) {  
            printf("DTOR => Obj:%p,Text:%p\n", this, text);  
            delete text;  
        } else {  
            printf("DTOR => Obj:%p (nothing to delete)\n", this);  
        }  
    }  
}
```

Object operations

- ▶ Let's consider the following class:

App.cpp

```
class String {  
    char * text;  
    void CopyString(const char * string) { ... }  
public:  
    String(const char * s) { ... }  
    ~String() { ... }  
  
    String(const String & obj) {  
        CopyString(obj.text);  
        printf("COPY => Obj:%p,Text:%p from Obj:%p,Text:%p\n", this, text,&obj,obj.text);  
    }  
    String& operator = (const String &obj) {  
        if (text != nullptr) {  
            printf("Clear => Obj:%p,Text:%p\n", this, text);  
            delete text;  
            text = nullptr;  
        }  
        CopyString(obj.text);  
        printf("EQ(Copy) => Obj:%p,Text:%p from Obj:%p,Text:%p\n", this, text, &obj, obj.text);  
        return (*this);  
    }  
}
```

- ▶ We will also add a copy-constructor and an assignment operator to this class.

Object operations

- ▶ What will be the result of the following code ?

App.cpp

```
class String {
    char * text;
    void CopyString(const char * string) { ... }
public:
    String(const char * s) { ... }
    ~String() { ... }
    String(const String & obj) { ... }
    String& operator = (const String &obj) { ... }
}
String Get(const char * text)
{
    printf("Entering Get function\n");
    String s(text);
    printf("Exiting Get function\n");
    return s;
}
void main()
{
    printf("Entering main function\n");
    String s("");
    s = Get("C++ test");
    printf("Exiting main function\n");
}
```

Object operations

- ▶ What will be the result of the following code ?

App.cpp

```
class String {  
    char * text;  
    void CopyString(const char * string) { ... }  
public:  
    String(const char * s) { ... }  
    ~String() { ... }  
    String(const String & obj) { ... }  
    String& operator = (const String &obj) { ... }  
}  
String Get(const char * text)  
{  
    printf("Entering Get function\n");  
    String s(text);  
    printf("Exiting Get function\n");  
    return s;  
}  
void main()  
{  
    printf("Entering main function\n");  
    String s("");  
    s = Get("C++ test");  
    printf("Exiting main function\n");  
}
```

Entering main function
CTOR => Obj:010FFA68,Text:01295040
Entering Get function
CTOR => Obj:010FFA04,Text:01295070
Exiting Get function
COPY => Obj:010FFA24,Text:01294EF8 from Obj:010FFA04,Text:01295070
DTOR => Obj:010FFA04,Text:01295070
Clear => Obj:010FFA68,Text:01295040
EQ(Copy) => Obj:010FFA68,Text:01294F30 from Obj:010FFA24,Text:01294EF8
DTOR => Obj:010FFA24,Text:01294EF8
Exiting main function
DTOR => Obj:010FFA68,Text:01294F30

Translated

GET function

Entering main function
CTOR => main::s (Text:01295040)
Entering Get function
CTOR => Get::s (Text:01295070)
Exiting Get function
COPY => temp_obj_1(Text:01294EF8) from Get::s(Text:01295070)
DTOR => Get::s(Text:01295070)
Clear => main::s(Text:01295040)
EQ(Copy) => main::s(Text:01294F30) from temp_obj_1(Text:01294EF8)
DTOR => temp_obj_1(Text:01294EF8)
Exiting main function
DTOR => main::s(Text:01294F30)

Object operations

- ▶ What will be the result of the following code ?

App.cpp

```
class String {  
    char * text;  
    void CopyString(const char * string) { ... }  
public:  
    String(const char * s) { ... }  
    ~String() { ... }  
    String(const String & obj) { ... }  
    String& operator = (const String &obj)  
}  
String Get(const char * text)  
{  
    printf("Entering Get function\n");  
    String s(text);  
    printf("Exiting Get function\n");  
    return s;  
}  
void main()  
{  
    printf("Entering main function\n");  
    String s("");  
    s = Get("C++ test");  
    printf("Exiting main function\n");  
}
```

From the *String::text* point of view the following happen:

1. Allocate memory for main::s::text
2. Allocate memory for Get::s::text
3. Allocate memory for temp_obj_1::text
4. Copy string from Get::s::text to temp_obj_1::text
5. Free memory for Get::s::text
6. Free memory for main::s::text
7. Copy memory from temp_obj_1::text to main::s::text
8. Free memory for temp_obj_1::text
9. Free memory for main::s::text

"C++ test" is allocated 3 times (for Get::S, temp_obj_1 and main::s)

Translated

Entering main function
CTOR => main::s (Text:01295040)

GET function
Entering Get function
CTOR => Get::s (Text:01295070)
Exiting Get function
COPY => temp_obj_1(Text:01294EF8) from Get::s(Text:01295070)
DTOR => Get::s(Text:01295070)

Clear => main::s(Text:01295040)
EQ(Copy) => main::s(Text:01294F30) from temp_obj_1(Text:01294EF8)
DTOR => temp_obj_1(Text:01294EF8)

Exiting main function
DTOR => main::s(Text:01294F30)

Object operations

- ▶ Let's consider the following class:

App.cpp

```
class String {  
    char * text;  
    void CopyString(const char * string) { ... }  
public:  
    String(const char * s) { ... }  
    ~String() { ... }  
    String(const String & obj) { ... }  
    String& operator = (const String &obj) { ... }  
  
    String(String && obj) {  
        this->text = obj.text;  
        obj.text = nullptr;  
        printf("MOVE => Obj:%p,Text:%p from Obj:%p,Text:%p\n", this, text, &obj, obj.text);  
    }  
    String& operator = (String &&obj) {  
        this->text = obj.text;  
        printf("EQ(Move) => Obj:%p,Text:%p from Obj:%p,Text:%p\n", this, text, &obj, obj.text);  
        obj.text = nullptr;  
        return (*this);  
    }  
}
```

- ▶ Now, we will add a move constructor and a move assignment to the class as well.

Object operations

- ▶ What will be the result of the following code ?

App.cpp

```
class String {
    char * text;
    void CopyString(const char * string) { ... }
public:
    String(const char * s) { ... }
    ~String() { ... }
    String(const String & obj) { ... }
    String& operator = (const String &obj) { ... }
    String(String && obj) { ... }
    String& operator = (String &&obj) { ... }
}
String Get(const char * text)
{
    printf("Entering Get function\n");
    String s(text);
    printf("Exiting Get function\n");
    return s;
}
void main()
{
    printf("Entering main function\n");
    String s("");
    s = Get("C++ test");
    printf("Exiting main function\n");
}
```

Object operations

- ▶ What will be the result of the following code ?

App.cpp

```
class String {  
    char * text;  
    void CopyString(const char * string) { ... }  
public:  
    String(const char * s) { ... }  
    ~String() { ... }  
    String(const String & obj) { ... }  
    String& operator = (const String &obj) { ... }  
    String(String && obj) { ... }  
    String& operator = (String &&obj) { ... }  
}  
String Get(const char * text)  
{  
    printf("Entering Get function\n");  
    String s(text);  
    printf("Exiting Get function\n");  
    return s;  
}  
void main()  
{  
    printf("Entering main function\n");  
    String s("");  
    s = Get("C++ test");  
    printf("Exiting main function\n");  
}
```

GET function

```
Entering main function  
CTOR => Obj:00AFFBD0,Text:00DA60A0  
Entering Get function  
CTOR => Obj:00AFFB6C,Text:00DA60D0  
Exiting Get function  
MOVE => Obj:00AFFB8C,Text:00DA60D0 from Obj:00AFFB6C,Text:00000000  
DTOR => Obj:00AFFB6C (nothing to delete)  
EQ(Move) => Obj:00AFFBD0,Text:00DA60D0 from Obj:00AFFB8C,Text:00DA60D0  
DTOR => Obj:00AFFB8C (nothing to delete)  
Exiting main function  
DTOR => Obj:00AFFBD0,Text:00DA60D0
```

Translated

```
Entering main function  
CTOR => main::s(Text:00DA60A0)  
Entering Get function  
CTOR => Get::s(Text:00DA60D0)  
Exiting Get function  
MOVE => temp_obj_1(Text:00DA60D0) from Get::s(Text:00000000)  
DTOR => Get::s (nothing to delete)  
EQ(Move) => main::s,Text:00DA60D0 from temp_obj_1(Text:00DA60D0)  
DTOR => temp_obj_1 (nothing to delete)  
Exiting main function  
DTOR => main::s(Text:00DA60D0)
```

Object operations

- ▶ What will be the result of the following code ?

App.cpp

```
class String {  
    char * text;  
    void CopyString(const char * string) { ... }  
public:  
    String(const char * s) { ... }  
    ~String() { ... }  
    String(const String & obj) { ... }  
    String& operator = (const String &obj)  
    String(String && obj) { ... }  
    String& operator = (String &&obj) { ... }  
}  
String Get(const char * text)  
{  
    printf("Entering Get function\n");  
    String s(text);  
    printf("Exiting Get function\n");  
    return s;  
}  
void main()  
{  
    printf("Entering main function\n");  
    String s("");  
    s = Get("C++ test");  
    printf("Exiting main function\n");  
}
```

From the *String::text* point of view the following happen:

1. Allocate memory for main::s::text
2. Allocate memory for Get::s::text
3. Copy the pointer of Get::s::text to temp_obj_1::text
4. Make Get::s::text NULL (nullptr)
5. Copy the pointer of temp_obj_1::text to main::s::text
6. Make temp_obj_1::text NULL (nullptr)
7. Free memory for main::s::text

"C++ test" is allocated one time and then the pointer is moved until it reaches the destination object ("s" from main)

Translated

Entering main function
CTOR => main::s(Text:00DA60A0)

GET function

Entering Get function
CTOR => Get::s(Text:00DA60D0)
Exiting Get function
MOVE => temp_obj_1(Text:00DA60D0) from Get::s(Text:00000000)
DTOR => Get::s (nothing to delete)

EQ(Move) => main::s,Text:00DA60D0 from temp_obj_1(Text:00DA60D0)
DTOR => temp_obj_1 (nothing to delete)

Exiting main function
DTOR => main::s(Text:00DA60D0)

Object operations

- ▶ What is the problem with the following code ?

App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                      Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
    Date(const Date & obj) { X = obj.X; }
    Date& operator = (Date &d)
    {
        X = d.X;
        return (*this);
    }
    Date& Get(int value)
    {
        Date d(value);
        return d;
    }
void main()
{
    Date d(1);
    d = Get(100);
}
```

App.pseudocode

```
class Date
{
    ...
};

Date* Get(int value)
{
    Date d(value);
    return &d;
}

void main()
{
    Date d(1);
    Date* tmpObj = Get(100);
    d.operator=(*tmpObj);
}
```

Object operations

- ▶ What is the problem with the following code ?

App.cpp

```
class Date
{
    int X,Y,Z,T;
public:
    Date(int value) : X(value), Y(value + 1),
                      Z(value + 2), T(value+3) {}
    void SetX(int value) { X = value; }
    Date(const Date & obj) { X = obj.X; }
    void operator = (Date &d)
    {
        X = d.X;
    }
};

Date& Get(int value)
{
    Date d(value);
    return d;
}

void main()
{
    Date d(1);
    d = Get(100);
}
```

```
void operator = (Date &d)
{
    push    ebp
    mov     ebp,esp
    sub    esp,0CCh
    push    ebx
    push    esi
    push    edi
    push    ecx
    lea     edi,[ebp-0CCh]
    mov     ecx,33h
    mov     eax,0CCCCCCCCh
    rep stos    dword ptr es:[edi]
    pop     ecx
    mov     dword ptr [this],ecx

    X = d.X;
    mov     eax,dword ptr [this]
    mov     edx,dword ptr [d]
    mov     edx,dword ptr [ecx]
    mov     dword ptr [eax],edx

    }
    pop     edi
    pop     esi
    pop     ebx
    mov     esp,ebp
    pop     ebp
    ret
```

4

Q & A

OOP

Gavrilut Dragos
Course 5

Summary

- ▶ Inheritance
- ▶ Virtual methods
- ▶ How virtual methods are modeled by C++ compiler
- ▶ Covariance
- ▶ Abstract classes (Interfaces)
- ▶ Memory alignment in case of inheritance

A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented at various angles, creating a sense of depth and movement.

- ▶ Inheritance

Inheritance

- ▶ Inheritance is a process that transfer class proprieties (methods and members) from one class (often called the base class to another that inherits the base class - called derived class). The derive class may extend the base class by adding additional methods and/or members.
- ▶ Such an example will be the class Automobile, where we can define the following properties:
 - ▶ Number of doors
 - ▶ Number of wheels
 - ▶ Size
- ▶ From this class we can derive a particularization of the Automobile class (for example electrical machines) that besides the properties of the base class (doors, wheels, size, etc) has its own properties (battery lifetime).

Inheritance

- ▶ Inheritance in case of C++ classes can be simple or multiple:

- ▶ Simple Inheritance

Simple

```
class <class_name>: <access modifier> <base class> { ... }
```

- ▶ Multiple Inheritance

Multiple

```
class <class_name>: <access modifier> <base class 1> ,  
                  <access modifier> <base class 2> ,  
                  <access modifier> <base class 3> ,  
                  ...  
                  <access modifier> <base class n> ,  
{ ... }
```

- ▶ The access modifier is optional and can be one of the following: (public / private or protected).
- ▶ If it is not specified, the default access modifier is private.

Inheritance

- ▶ Class “Derived” inherits members and methods from class “Base”. That is why we can call methods SetX and SetY from an instance of “Derived” class.

App.cpp

```
class Base
{
public:
    int x;
    void SetX(int value);
};

class Derived : public Base
{
    int y;
public:
    void SetY(int value);
};

void main()
{
    Derived d;
    d.SetX(100);
    d.x = 10;
    d.SetY(200);
}
```

Inheritance

- ▶ The following code will not compile. Class “Derived” inherits class “Base”, but member “x” from class Base is private (this means that it can not be accessed in class “Derived”).

App.cpp

```
class Base
{
private:
    int x;
};
class Derived : public Base
{
    int y;
public:
    void SetY(int value);
    void SetX(int value);
};
void Derived::SetX(int value)
{
    x = value; // Red box highlights this line
}
void main()
{
    Derived d;
    d.SetX(100);
    d.SetY(200);
}
```

error C2248: 'Base::x': cannot access private member declared
in class 'Base'
note: see declaration of 'Base::x'
note: see declaration of 'Base'

Inheritance

- ▶ The solution for this case is to use the “***protected***” access modifier. A protected member is a member that can be accessed by classes that inherits current class, but it can not be accessed from outside the class.

App.cpp

```
class Base
{
protected:
    int x;
};

class Derived : public Base
{
    int y;
public:
    void SetY(int value);
    void SetX(int value);
};
void Derived::SetX(int value)
{
    x = value;
}
void main()
{
    Derived d;
    d.SetX(100);
    d.SetY(200);
}
```

Inheritance

- ▶ The code below will not compile. “x” is declared as ***protected*** - this means that it can be accessed in method ***SetX*** from a derived class, but it can not be accessed outside it’s scope (class).

App.cpp

```
class Base
{
protected:
    int x;
};
class Derived : public Base
{
    int y;
public:
    void SetY(int value);
    void SetX(int value);
};
void Derived::SetX(int value)
{
    x = value;
}
void main()
{
    Derived d;
    d.SetX(100);
    d.x = 100;
}
```

error C2248: 'Base::x': cannot access protected member declared
in class 'Base'
note: see declaration of 'Base::x'
note: see declaration of 'Base'

Inheritance

- ▶ The following table shows if a member with a specific access modifier can be accessed and in what conditions:

Access modifier	In the same class	In a derived class	Outside it's scope	Friend function in the base class	Friend function in the derived class
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	No	Yes	Yes
private	Yes	No	No	Yes	No

Inheritance

- ▶ The code below will not compile. “x” is a private member of “Base” therefore a friend function defined in “Derived” class can not access it.

App.cpp

```
class Base
{
private:
    int x;
};
class Derived : public Base
{
    int y;
public:
    void SetY(int value);
    void friend SetX(Derived &d);
};
void SetX(Derived &d)
{
    d.x = 100;
}
void main()
{
    Derived d;
    SetX(d);
}
```

Inheritance

- ▶ The solution is to change the access modifier of data member “x” from class “Base” from private to protected.

App.cpp

```
class Base
{
protected:
    int x;
};
class Derived : public Base
{
    int y;
public:
    void SetY(int value);
    void friend SetX(Derived &d);
};
void SetX(Derived &d)
{
    d.x = 100;
}
void main()
{
    Derived d;
    SetX(d);
}
```

Inheritance

- ▶ Be careful where you define the friend function. In the example below SetX friend function is declared in the “Derived” class. This means that it can access methods and data members from instances of “Derived” class and not other classes (e.g. Base class). The code will not compile.

App.cpp

```
class Base
{
private:
    int x;

};

class Derived : public Base
{
    int y;
public:
    void SetY(int value);
    void friend SetX(Base &d);
};

void SetX(Base &d)
{
    d.x = 100;
}

void main()
{
    Derived d;
}
```

Inheritance

- ▶ This code will work properly because the friend function is defined in class “Base”.

App.cpp

```
class Base
{
private:
    int x;
public:
    void friend SetX(Base &d);
};

class Derived : public Base
{
    int y;
public:
    void SetY(int value);

};

void SetX(Base &d)
{
    d.x = 100;
}

void main()
{
    Derived d;
}
```

Inheritance

- ▶ Access modifiers can also be applied to the inheritance relation.
- ▶ As a result, the members from the base class change their original access modifier in the derived class.

App.cpp

```
class Base
{
public:
    int x;
};

class Derived : public Base
{
    int y;
public:
    void SetY(int value) { ... }
};

void main()
{
    Derived d;
    d.x = 100;
}
```

- ❖ In this case, because “x” is public in the “Base” class, and the inheritance relation is also public, “x” will be public as well in the “Derived” class and will be accessible from outside the class scope.

Inheritance

- ▶ Access modifiers can also be applied to the inheritance relation.
- ▶ As a result, the members from the base class change their original access modifier in the derived class.

App.cpp

```
class Base
{
public:
    int x;
};

class Derived : private Base
{
    int y;
public:
    void SetY(int value) { ... }
};

void main()
{
    Derived d;
    d.x = 100;
}
```

- ❖ This code will not compile. “x” is indeed public in class “Base”, but since the inherit relation between class “Base” and class “Derived” is private, “x” will change its access modifier from public to private in class Derived and will not be accessible from outside its scope.
- ❖ However, if we are to create an instance of type “Base” we will be able to access “x” for that instance outside its scope.

Inheritance

- ▶ The rules that show how an access modifier is change if we change the access modifier of the inheritance relation are as follows:

Access modifier used for the inheritance relation →	public	private	protected
Access modifier used for a data member or method			
public	public	private	protected
private	private	private	private
protected	protected	private	protected

private > protected > public

Inheritance

- ▶ Let's consider the following case:

App.cpp

```
class A
{
public:
    A() { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};
class B: public A
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};
int main()
{
    B b;
    return 0;
}
```

Output

```
ctor: A is called !
ctor: B is called !
dtor: B is called !
dtor: A is called !
```

Inheritance

- ▶ Let's consider the following case:

App.cpp

```
class A
{
public:
    A() { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};
class B: public A
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};
int main()
{
    B b;
    return 0;
}
```

Output

```
push
mov
sub
ebp
ebp,esp
esp,44h

mov
mov
call
dword ptr [this],ecx
ecx,dword ptr [this]
A::A (0DB14B5h)

push
call
add
mov
offset string "ctor: B is called !\n"
_call (0DB14A6h)
esp,4
eax,dword ptr [this]

mov
pop
ret
esp,ebp
ebp
```

- ▶ The code in YELLOW reflects the execution of the base constructor.

Inheritance

- ▶ Let's consider the following case:

App.cpp

```
class A
{
public:
    A() { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};

class B
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};

class C: public B, public A
{
public:
    C() { printf("ctor: C is called !\n"); }
    ~C() { printf("dtor: C is called !\n"); }
};

int main()
{
    C c;
    return 0;
}
```

Output

```
ctor: B is called !
ctor: A is called !
ctor: C is called !
dtor: C is called !
dtor: A is called !
dtor: B is called !
```

- ▶ In case of multiple inheritance, the order of base classes is used when the constructor is called (in this case - first class B, then class A and finally class C)

Inheritance

- ▶ Let's consider the following case:

App.cpp

```
class A
{
public:
    A() { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};
class B
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};
class C: public B, public A
{
public:
};
int main() {
    C c;
    return 0;
}
```

Output

```
ctor: B is called !
ctor: A is called !
dtor: A is called !
dtor: B is called !
```

- ▶ If no constructor is defined in class C, but there are at least one constructor defined in one of the class from which C is derived from, the compiler will create a default constructor that calls the constructor of class B followed by the constructor of class A.

Inheritance

- ▶ Let's consider the following case:

App.cpp

```
class A
{
public:
    A(int x) { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};

class B
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};

class C: public B, public A
{
public:
};

int main() {
    C c;
    return 0;
}
```

error C2280: 'C::C(void)': attempting to reference a deleted function
note: compiler has generated 'C::C' here
note: 'C::C(void)': function was implicitly deleted because a base class
'A' has either no appropriate default constructor or overload resolution
was ambiguous
note: see declaration of 'A'

- ▶ This code will fail, as there is no explicit call to *A::A(int)* constructor.

Inheritance

- ▶ Let's consider the following case:

App.cpp

```
class A
{
public:
    A(int x) { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};

class B
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};

class C: public B, public A
{
public:
    C() : A(100) { printf("ctor: C is called !\n"); }
    ~C() { printf("dtor: C is called !\n"); }
};

int main()
{
    C c;
    return 0;
}
```

Output

```
ctor: B is called !
ctor: A is called !
ctor: C is called !
dtor: C is called !
dtor: A is called !
dtor: B is called !
```

- ▶ The solution is to explicitly call the constructor of A in the member initializer list for C::C()

Inheritance

- ▶ Let's consider the following case:

App.cpp

```
class A
{
public:
    A(int x) { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};

class B
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};

class C: public B, public A
{
public:
    C() : A(100), B() { printf("ctor: C is called !\n"); }
    ~C() { printf("dtor: C is called !\n"); }
};

int main()
{
    C c;
    return 0;
}
```

Output

```
ctor: B is called !
ctor: A is called !
ctor: C is called !
dtor: C is called !
dtor: A is called !
dtor: B is called !
```

Using this method **WILL NOT CHANGE** the order of the constructors (in this case, even if we call A(100) followed by B(), the compiler will still call **B::B()** first and then **A::A(int)**)

Inheritance

- ▶ Let's consider the following case:

App.cpp

```
class A {  
public:  
    A(int x) { printf("ctor: A is called !\n"); }  
    ~A() { printf("dtor: A is called !\n"); }  
};  
class B {  
public:  
    B() { printf("ctor: B is called !\n"); }  
    ~B() { printf("dtor: B is called !\n"); }  
};  
class C {  
public:  
    C(bool n) { printf("ctor: C is called !\n"); }  
    ~C() { printf("dtor: C is called !\n"); }  
};  
class D: public B, public A  
{  
    C c;  
public:  
    D(): c(true), A(100), B() { printf("ctor: D is called !\n"); }  
    ~D() { printf("dtor: D is called !\n"); }  
};  
int main()  
{  
    D d;  
    return 0;  
}
```

Output

```
ctor: B is called !  
ctor: A is called !  
ctor: C is called !  
ctor: D is called !  
dtor: D is called !  
dtor: C is called !  
dtor: A is called !  
dtor: B is called !
```

Inheritance

- ▶ Let's consider the following case:

App.cpp

```
struct A {
    A(int x) { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};

struct B {
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};

struct C {
    C(bool n) { printf("ctor: C is called !\n"); }
    ~C() { printf("dtor: C is called !\n"); }
};

struct D {
    D(bool n) { printf("ctor: D is called !\n"); }
    ~D() { printf("dtor: D is called !\n"); }
};

class E: public B, public A {
    C c;
    D d;
    int v1, v2;
public:
    E(): d(true), A(100), c(true), B(), v2(100), v1(20) { printf("ctor: E is called !\n"); }
    ~E() { printf("dtor: E is called !\n"); }
};

void main() {
    E e;
}
```

Output

```
ctor: B is called !
ctor: A is called !
ctor: C is called !
ctor: D is called !
-- v1 is initialized
-- v2 is initialized
ctor: E is called !
dtor: E is called !
dtor: D is called !
dtor: C is called !
dtor: A is called !
dtor: B is called !
```

Inheritance

- ▶ When inheriting from multiple classes, the general rule for calling constructors and destructors is as follows:
 1. First all of the constructors from the base classes are called in the order of their inheriting definition (left-to-right)
 2. All of the constructors from data members are called (again in their definition order - top-to-bottom)
 3. Then the constructor initialization value for data members (basic types, references, constants) are used in the order they are defined in that class.
 4. Finally, the code of the constructor of the class is called.
- ▶ Destructors are called in a reverse way (starting from point 4 to point 1).

Tested with:

- ▶ cl.exe: 19.16.27030.1
- ▶ Params: /permissive- /GS- /analyze- /W3 /Zc:wchar_t /ZI /Gm- /Od /sdl /Fd"Debug\vc141.pdb" /Zc:inline /fp:precise /D "WIN32" /D "_DEBUG" /D "_CONSOLE" /D "_UNICODE" /D "UNICODE" /errorReport:prompt /WX- /Zc:forScope /RTCu /arch:IA32 /Gd /Oy- /MDd /FC /Fa"Debug\" /nologo /Fo"Debug\" /Fp"Debug\TestCpp.pch" /diagnostics:classic

A decorative graphic in the top-left corner consists of several overlapping blue triangles of varying shades, creating a sense of depth and perspective.

► Virtual methods

Virtual methods

App.cpp

```
class A
{
public:
    int a1, a2, a3;
    void Set() { printf("A"); }
};

class B: public A
{
public:
    int b1, b2;
    void Set() { printf("B"); }
};

void main()
{
    B b;
    b.Set();
}
```

- ❖ This code prints “*B*” on the screen. From the inheritance point of view, both *A* and *B* class have the same method called *Set*
- ❖ In this case it is said that class *B* hides method *Set* from class *A*

Virtual methods

App.cpp

```
class A
{
public:
    int a1, a2, a3;
    void Set() { printf("A"); }
};

class B: public A
{
public:
    int b1, b2;
    void Set() { printf("B"); }
};

void main()
{
    B b;
    A* a = &b;
    a->Set();
}
```

- ❖ In this case, the code will print “A” on the screen, because we are using a pointer of type *A**
- ❖ However, in reality, “a” pointer points to an object of type *B* → so the expected result should be that the product will print “*B*” and not “*A*”
- ❖ So ... what can we do to change this behavior ?

Virtual methods

App.cpp

```
class A
{
public:
    int a1, a2, a3;
    virtual void Set() { printf("A"); }
};

class B: public A
{
public:
    int b1, b2;
    void Set() { printf("B"); }
};

void main()
{
    B b;
    A* a = &b;
    a->Set();
}
```

- ❖ The solution is to use “**virtual**” keyword in front of a method - definition
- ❖ If we do this, the program will print “**B**”
- ❖ In this case, it is said that class B **overrides** method Set from class A
- ❖ **Using virtual keyword makes a method to be part of the instance !**

Virtual methods

Virtual methods can be used for:

- ▶ Polymorphism
- ▶ Memory deallocation (virtual destructor)
- ▶ Anti-debugging techniques

Virtual methods

Polymorphism = the ability to access instances of different classes through the same interface. In particular to C++, this translates into the ability to automatically convert (cast) a pointer to a certain class to its base class.

App-1.cpp

```
class Figure {
    public: virtual void Draw() { printf("Figure"); }
};

class Circle: public Figure {
    public: void Draw() { printf("Circle"); }
};

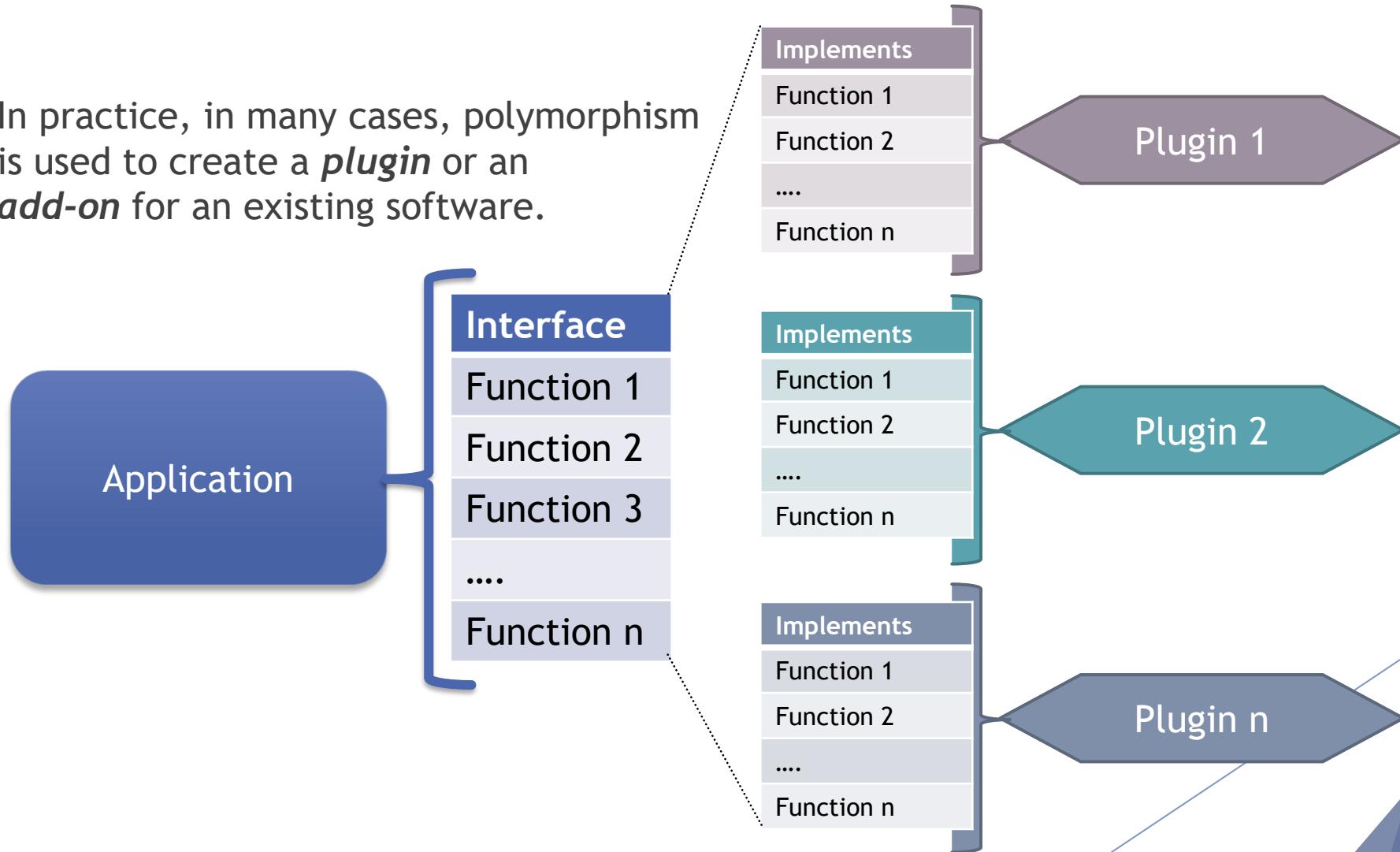
class Square: public Figure {
    public: void Draw() { printf("Square"); }
};

void main()
{
    Figure *f[2];
    f[0] = new Circle();
    f[1] = new Square();
    for (int index = 0;index<2;index++)
        f[index]->Draw();
}
```

- ❖ After the execution this code will print on the screen “**Circle**” and “**Square**”.
- ❖ If we haven’t uses **virtual** specifier, the program would have printed “**Figure**” twice !

Virtual methods

In practice, in many cases, polymorphism is used to create a *plugin* or an *add-on* for an existing software.



Virtual methods

In particular for C++ language, *virtual* specifier can be used as a specifier for destructors.

Let's analyze the following case:

App-1.cpp

```
class Figure {
    public: virtual void Draw() { printf("Figure"); }
    public: ~Figure() { printf("Delete Figure\n"); }
};
class Circle: public Figure {
    public: void Draw() { printf("Circle"); }
    public: ~Circle() { printf("Delete Circle"); }
};
class Square: public Figure {
    public: void Draw() { printf("Square"); }
    public: ~Square() { printf("Delete Square"); }
};
void main() {
    Figure *f[2];
    f[0] = new Circle();
    f[1] = new Square();
    for (int index = 0; index < 2; index++)
        delete (f[index]);
}
```

- ❖ After this code gets executed, the following texts will be printed on the screen:
“Delete Figure”
“Delete Figure”.
- ❖ What would happen if both *Circle* and *Square* classes allocate some memory ?

Virtual methods

In particular for C++ language, *virtual* specifier can be used as a specifier for destructors.

Let's analyze the following case:

App-1.cpp

```
class Figure {
    public: virtual void Draw() { printf("Figure"); }
    public: virtual ~Figure() { printf("Delete Figure\n"); }
};
class Circle: public Figure {
    public: void Draw() { printf("Circle"); }
    public: ~Circle() { printf("Delete Circle"); }
};
class Square: public Figure {
    public: void Draw() { printf("Square"); }
    public: ~Square() { printf("Delete Square"); }
};
void main() {
    Figure *f[2];
    f[0] = new Circle();
    f[1] = new Square();
    for (int index = 0; index < 2; index++)
        delete (f[index]);
}
```

- ❖ The solution is to declare de destructor as *virtual*. As a result, the destructor for actual class will be called, followed by the destructor of the base class
- ❖ The following text will be printed:
**Delete Circle
Delete Figure
Delete Square
Delete Figure**

Virtual methods

Let's analyze the following case:

App-1.cpp

```
class A
{
public:
    virtual bool Odd(int x) { return x % 2 == 0; }
};

class B : public A
{
public:
    virtual bool Odd(char x) { return x % 3 == 0; }
};

int main() {
    A* a = new B();
    printf("%d\n", a->Odd(3));
    return 0;
}
```

- ▶ *Odd* is a virtual function - however, class B **does not override** it (as it uses *char* as the first parameter instead of *int*). As a result, class B will have 2 *Odd* methods and a->Odd will call the one with an *int* parameter. Upon execution, value **false** (0) is written to the screen.

Virtual methods

Let's analyze the following case:

App-1.cpp

```
class A
{
public:
    virtual bool Odd(int x) { return x % 2 == 0; }
};

class B : public A
{
public:
    virtual bool Odd(char x) { return x % 3 == 0; }
};

int main() {
    A* a = new B();
    printf("%d\n", a->Odd(3));
    return 0;
}
```

To override a virtual function, one must use
the SAME method signature !

- ▶ *Odd* is a virtual function - however, class B does not *override* it (as it uses *char* as the first parameter instead of *int*). As a result, class B will have 2 *Odd* methods and a->Odd will call the one with an *int* parameter. Upon execution, value *false* (0) is written to the screen.

Virtual methods

Let's analyze the following case:

App-1.cpp

```
class A
{
public:
    virtual bool Odd(int x) { return x % 2 == 0; }
};

class B : public A
{
public:
    virtual bool Odd(char x) override { return x % 3 == 0; }
};

int main() {
    A* a = new B();
    printf("%d\n", a->Odd(3));
    return 0;
}
```

error C3668: 'A::Odd': method with override specifier
'override' did not override any base class methods

- ▶ Assuming that , in reality, the intent was to override *Odd* method, then one way of making sure that this kind of mistakes will not happen is to use the **override** keyword (added with C++11 standard). As a result, this code will not compile as it is expected that method Odd to have the same signature !!!

Virtual methods

Let's analyze the following case:

App-1.cpp

```
class A
{
public:
    virtual bool Odd(int x) { return x % 2 == 0; }
};

class B : public A
{
public:
    virtual bool Odd(int x) override { return x % 3 == 0; }
};

int main() {
    A* a = new B();
    printf("%d\n", a->Odd(3));
    return 0;
}
```

- ▶ Now the code compiles and prints “1” (true) on the screen.

Virtual methods

Let's consider the following code:

App-1.cpp

```
struct A {  
    virtual bool Odd(int x) = 0;  
};  
struct B : public A {  
    virtual bool Odd(int x) { return x % 2 == 0; }  
};  
struct C : public B {  
    virtual bool Odd(int x) { return x % 3 == 0; }  
};  
int main() {  
    A* a = new C();  
    printf("%d\n", a->Odd(3));  
    return 0;  
}
```

- ▶ This program runs and prints value 1 (True) → even if 3 is not an odd number.
- ▶ The reason why this could happen is that method *Odd* was overridden in class C (keep in mind that we have used *struct* in this example to show that the behavior is identical to the one from *class*).
- ▶ What can we do if we want to make sure that *Odd* method from class B **can not** be overridden ?

Virtual methods

Let's consider the following code:

App-1.cpp

```
struct A {  
    virtual bool Odd(int x) = 0;  
};  
struct B : public A {  
    virtual bool Odd(int x) final { return x % 2 == 0; }  
};  
struct C : public B {  
    virtual bool Odd(int x) { return x % 3 == 0; }  
};  
int main() {  
    A* a = new C();  
    printf("%d\n", a->Odd(3));  
    return 0;  
}
```

error C3248: 'B::Odd': function declared as 'final'
cannot be overridden by 'C::Odd'

- ▶ The solution is to use the specifier *final* after the declaration of a virtual function. This tells the compiler that other classes that inherit current class can not **override** that method.

Virtual methods

Let's consider the following code:

App-1.cpp

```
struct A {  
    virtual bool Odd(int x) = 0;  
};  
struct B : public A {  
    virtual bool Odd(int x) override final { return x % 2 == 0; }  
};  
struct C : public B {  
};  
int main() {  
    A* a = new C();  
    printf("%d\n", a->Odd(3));  
    return 0;  
}
```

- ▶ It is possible to use both ***override*** and ***final*** specifiers when declaring a method.
- ▶ In this case their meaning is:
 - ▶ ***override*** → The purpose of this method is to override the existing method from the base class (in this case, it overrides *A::Odd*)
 - ▶ ***final*** → Other classes that might inherit class B can not override this method.

Virtual methods

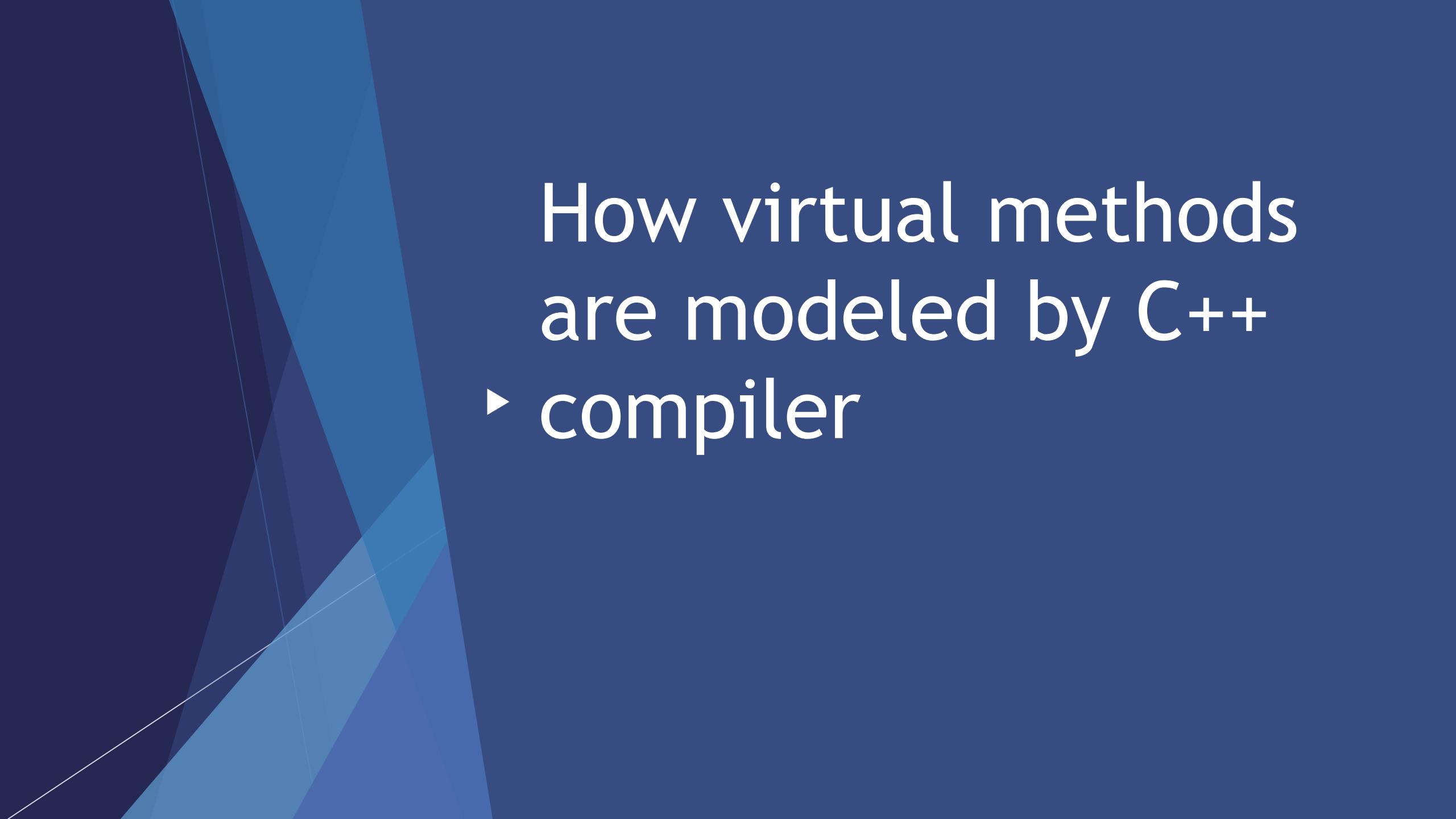
Let's consider the following code:

App-1.cpp

```
struct A {  
    virtual bool Odd(int x) = 0;  
};  
struct B final : public A  
{  
    virtual bool Odd(int x) override { return x % 2 == 0; }  
};  
struct C : public B {  
};  
int main() {  
    A* a = new C();  
    printf("%d\n", a->Odd(3));  
    return 0;  
}
```

error C3246: 'C': cannot inherit from 'B' as it has been declared as 'final'

- ▶ **final** specifier can also be used directly in the class/struct definition. In this case , it's meaning is that inheritance from class B is NOT possible.
- ▶ This code will not compile !

A decorative graphic in the top-left corner consists of several overlapping blue triangles of varying shades, creating a geometric pattern.

How virtual methods are modeled by C++ compiler

How virtual methods are modeled by C++ compiler

- ❖ Let's analyze the following two programs. Their only difference is the usage of *virtual* in case of APP-2.

App-1.cpp

```
class A
{
public:
    int a1, a2, a3;
    void Set() { printf("A"); }
};
void main()
{
    printf("%d", sizeof(A));
}
```

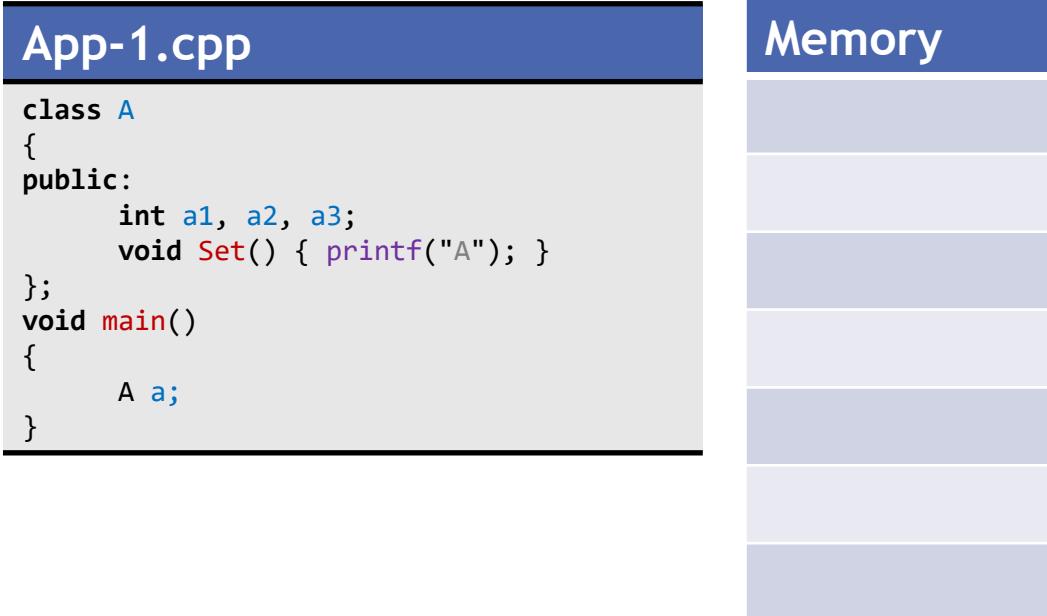
App-2.cpp

```
class A
{
public:
    int a1, a2, a3;
    virtual void Set() { printf("A"); }
};
void main()
{
    printf("%d", sizeof(A));
}
```

- ❖ When executed, APP-1 will print “12” and App-2 will print “16” (for x86 architecture). If we run the same App-2 on x64 it will print “24”
- ❖ Why ?

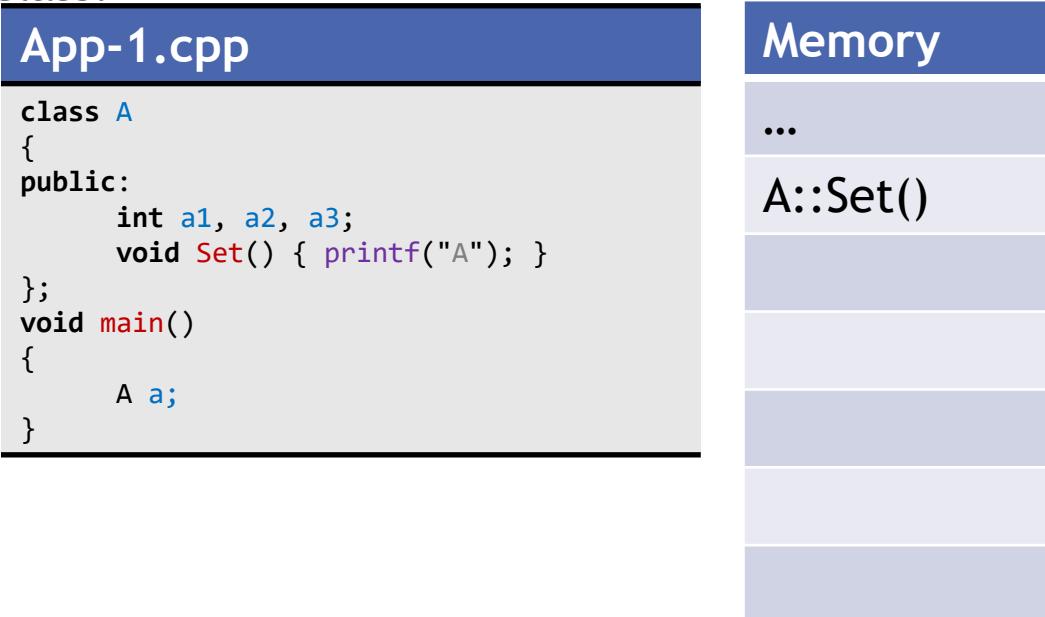
How virtual methods are modeled by C++ compiler

- ❖ Using the *virtual* keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called *vptr* and if added is the first pointer in the class.



How virtual methods are modeled by C++ compiler

- ❖ Using the *virtual* keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called *vptr* and if added is the first pointer in the class.



How virtual methods are modeled by C++ compiler

- ❖ Using the *virtual* keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called *vptr* and if added is the first pointer in the class.

App-1.cpp

```
class A
{
public:
    int a1, a2, a3;
    void Set() { printf("A"); }
};
void main()
{
    A a;
```

Memory

```
...
A::Set()
...
main()
```

How virtual methods are modeled by C++ compiler

- ❖ Using the *virtual* keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called *vptr* and if added is the first pointer in the class.

App-1.cpp

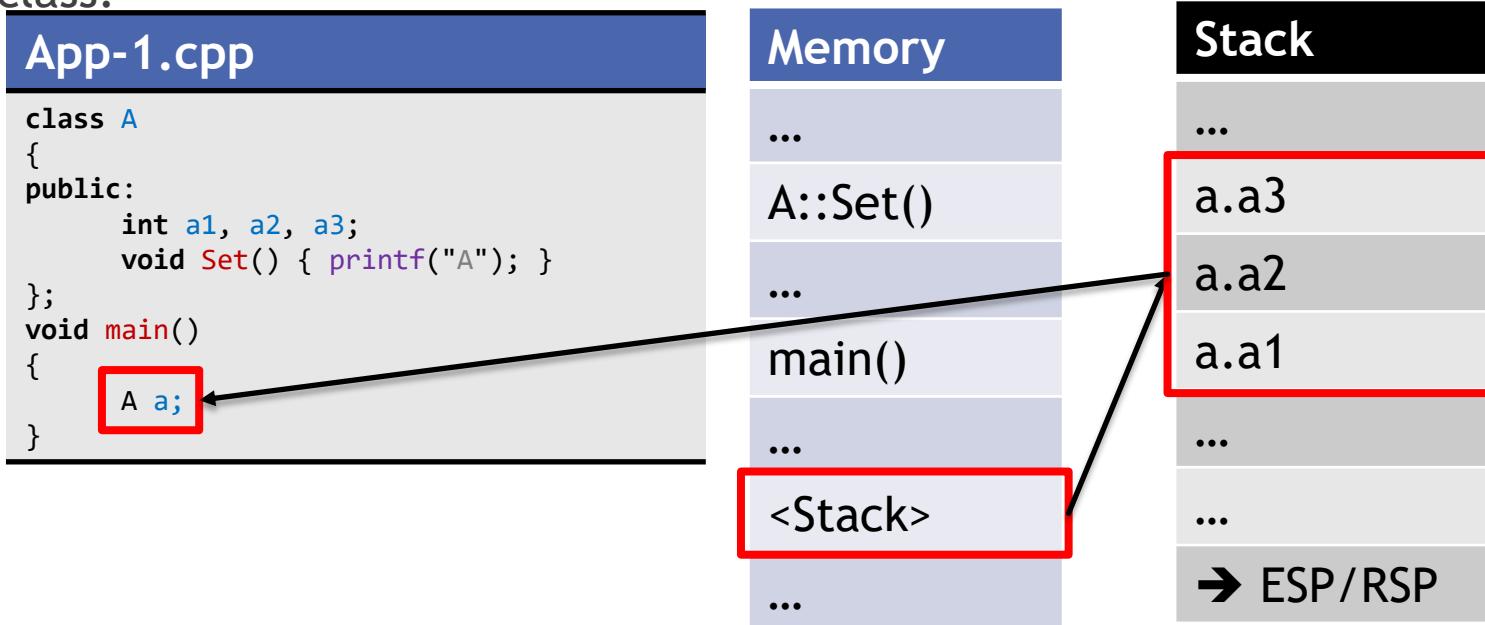
```
class A
{
public:
    int a1, a2, a3;
    void Set() { printf("A"); }
};
void main()
{
    A a;
}
```

Memory

```
...
A::Set()
...
main()
...
<Stack>
...
```

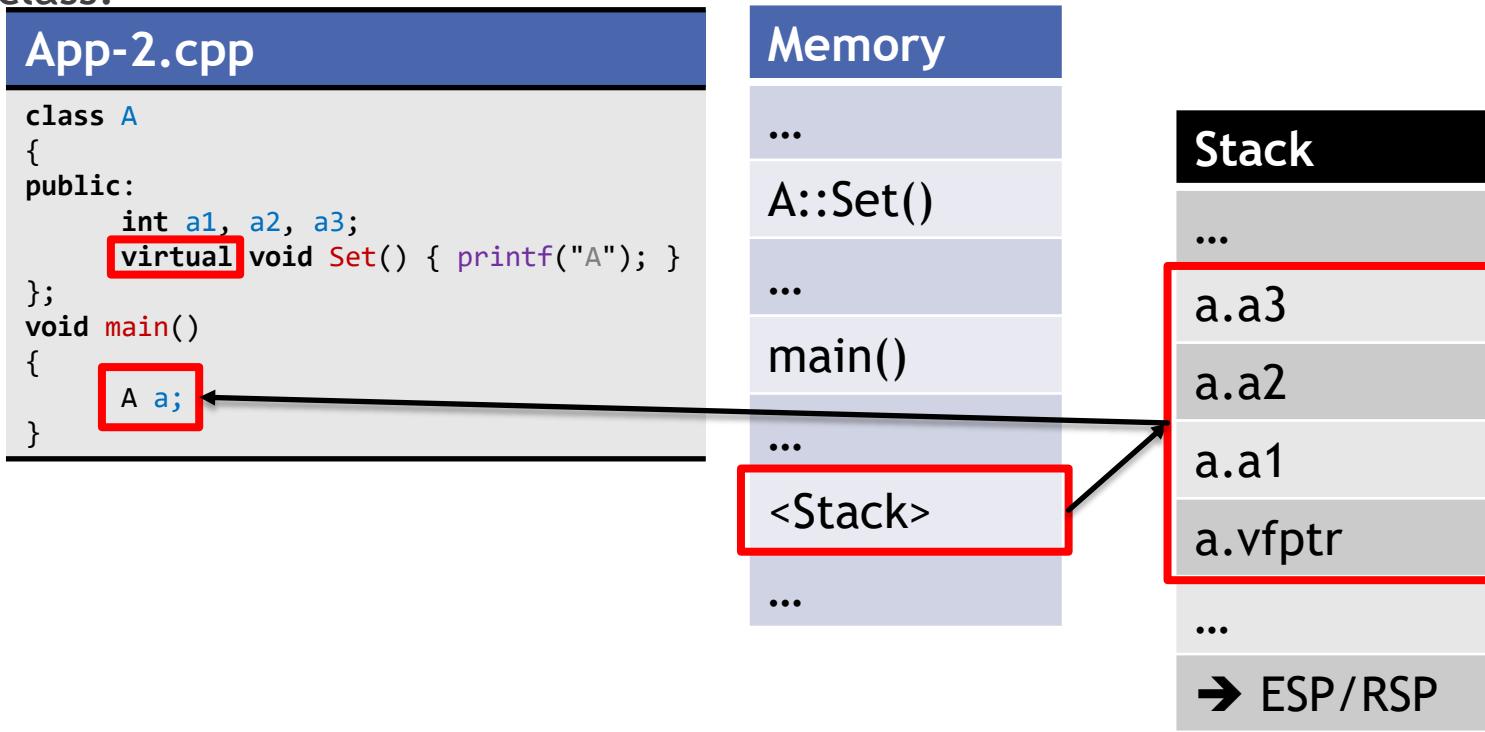
How virtual methods are modeled by C++ compiler

- ❖ Using the ***virtual*** keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called ***vptr*** and if added is the first pointer in the class.



How virtual methods are modeled by C++ compiler

- Using the ***virtual*** keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called ***vfptr*** and if added is the first pointer in the class.

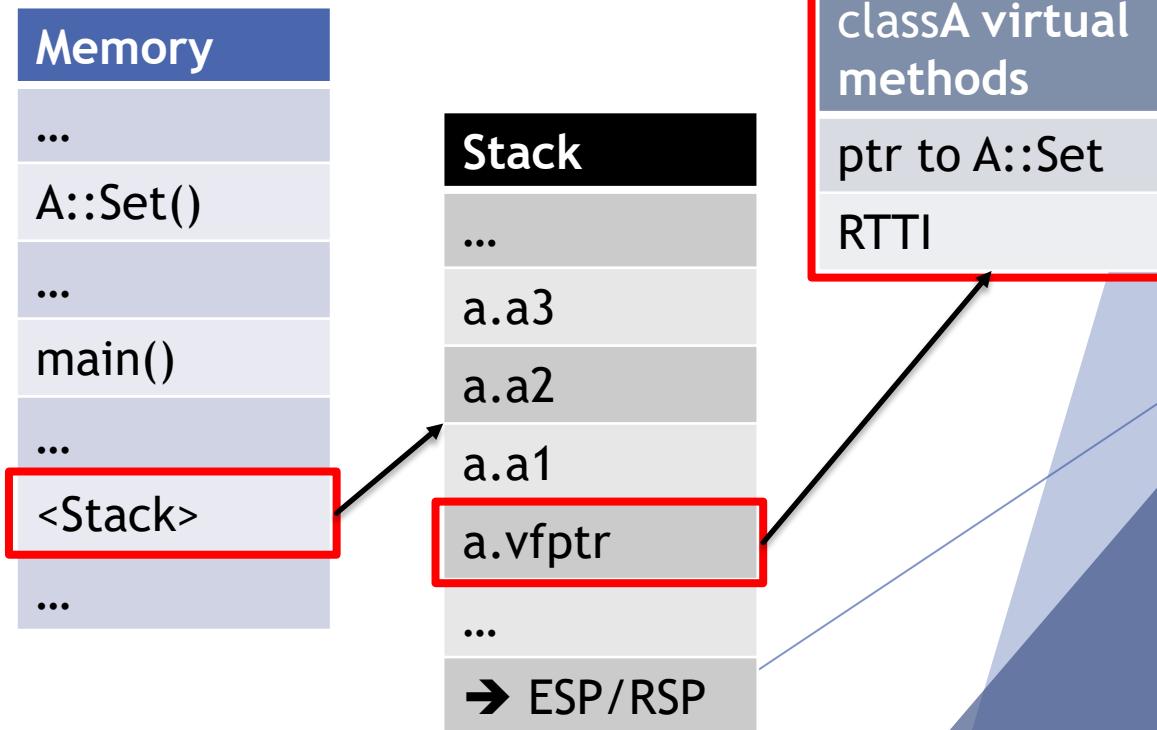


How virtual methods are modeled by C++ compiler

- Using the ***virtual*** keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called ***vfptr*** and if added is the first pointer in the class.

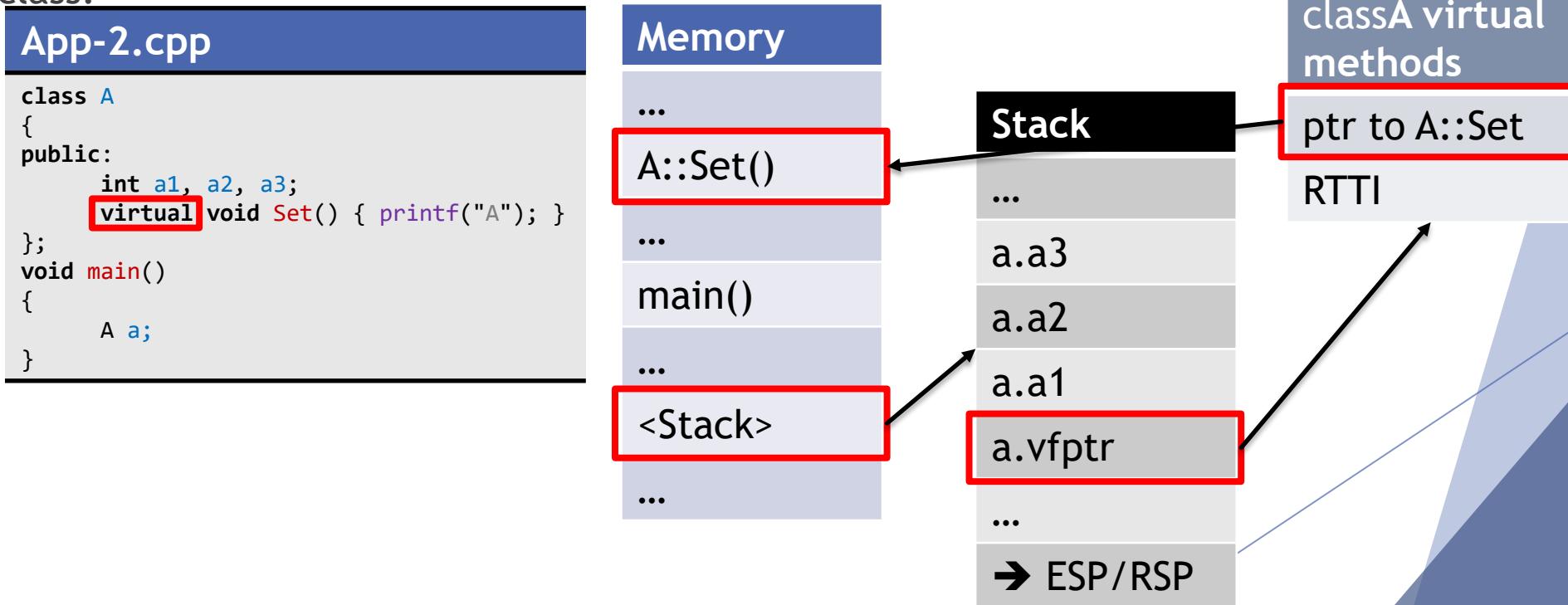
App-2.cpp

```
class A
{
public:
    int a1, a2, a3;
    virtual void Set() { printf("A"); }
};
void main()
{
    A a;
}
```



How virtual methods are modeled by C++ compiler

- Using the ***virtual*** keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called ***vfptr*** and if added is the first pointer in the class.



How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to be certain that `vfptr` pointer is set correctly. As such, any constructor is modified to include the code that sets up the `vfptr` pointer. If no constructor is present, the default one will be created automatically.

App.cpp

```
class A
{
public:
    int x, y;
    int Calcul() { return x+y; }
};
void main()
{
    A a;
    a.x = 1;
    a.y = 2;
}
```

Disasm

```
A a;
a.x = 1;
mov     dword ptr [ebp-12],1
a.y = 2;
mov     dword ptr [ebp-8],2
```

- In this case, there no default constructor defined and no need for the compiler to provide one automatically (e.g. virtual methods, const or reference data members, etc).

How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to make certain that `vptr` pointer is set correctly. As such, any constructor is modified to include the code that sets up the `vptr` pointer. If no constructor is present, the default one will be created automatically.

App.cpp

Disasm

```
class A
{
public:
    int x, y;
    int Calcul() { return x+y; }
    A() { x = y = 0; }
};
void main()
{
    A a;
    a.x = 1;
    a.y = 2;
}
```

```
A a;
lea    ecx,[ebp-16]
call   A::A
a.x = 1;
mov    dword ptr [ebp-16],1
a.y = 2;
mov    dword ptr [ebp-12],2
```

- In this case, there is a constructor that will be called when “a” is created.

How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to make certain that **vfptr** pointer is set correctly. As such, any constructor is modified to include the code that sets up the **vfptr** pointer. If no constructor is present, the default one will be created automatically

App.cpp	Disasm (A::A)
<pre>class A { public: int x, y; int Calcul() { return x+y; } A() { x = y = 0; } }; void main() { A a; a.x = 1; a.y = 2; }</pre>	<pre>push ebp mov ebp,esp mov dword ptr [ebp-8],ecx // EBP-8=this mov eax,dword ptr [ebp-8] mov dword ptr [eax+4],0 // this->y = 0 mov ecx,dword ptr [ebp-8] mov dword ptr [ecx],0 // this->x = 0 mov eax,dword ptr [ebp-8] pop ebp ret</pre>

- In this case, there is a default constructor and the code from the default constructor will be called when object “a” is created.

How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to make certain that **vfptr** pointer is set correctly. As such, any constructor is modified to include the code that sets up the **vfptr** pointer. If no constructor is present, the default one will be created automatically

App.cpp	Disasm
<pre>class A { public: int x, y; virtual int Calcul() {return x+y;} }; void main() { A a; a.x = 1; a.y = 2; }</pre>	<pre>A a; lea ecx,[ebp-20] call A::A a.x = 1; mov dword ptr [ebp-16],1 a.y = 2; mov dword ptr [ebp-12],2</pre>

- In this case, even if no constructor is defined, the compiler will automatically create one to initialize the **vfptr** pointer (this is required because **Calcul** is a virtual method).

How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to make certain that `vfptr` pointer is set correctly. As such, any constructor is modified to include the code that sets up the `vfptr` pointer. If no constructor is present, the default one will be created automatically

App.cpp	Disasm
<pre>class A { public: int x, y; virtual int Calcul() {return x+y;} }; void main() { A a; a.x = 1; a.y = 2; }</pre>	<p>Disasm</p> <pre>A a; lea ecx,[ebp-20] call A::A d.a.x = 1; ebp-16],1 ebp-12],2</pre> <p>Disasm</p> <pre>push ebp mov ebp,esp dword ptr [ebp-8],ecx eax,dword ptr [ebp-8] dword ptr [eax], A-virtual-fnc-list eax,dword ptr [ebp-8] mov esp,ebp pop ebp ret</pre>

Memory address where a list of pointers to virtual functions is (in this case only one method: *Calcul*)

How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to make certain that `vfptr` pointer is set correctly. As such, any constructor is modified to include the code that sets up the `vfptr` pointer. If no constructor is present, the default one will be created automatically

App.cpp	Disasm
<pre>class A { public: int x, y; virtual int Calcul() {return x+y;} A() { x = y = 0; } }; void main() { A a; a.x = 1; a.y = 2; }</pre>	<pre>A a; lea ecx,[ebp-20] call A::A a.x = 1; mov dword ptr [ebp-16],1 a.y = 2; mov dword ptr [ebp-12],2</pre>

- If a constructor exists, it will be modified (in a similar manner to the change that is done for const/references data members).

How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to make certain that **vfptr** pointer is set correctly. As such, any constructor is modified to include the code that sets up the **vfptr** pointer. If no constructor is present, the default one will be created automatically.

App.cpp

```
class A
{
public:
    int x, y;
    virtual int Calcul() {return x+y;}
    A() { x = y = 0; }
};
void main()
{
    A a;
    a.x = 1;
    a.y = 2;
}
```

Disasm A::A

```
push    ebp
mov     ebp,esp
mov     dword ptr [ebp-8],ecx
mov    eax,dword ptr [ebp-8]
mov    dword ptr [eax],addr virt fnc
mov    eax,dword ptr [ebp-8]
mov    dword ptr [eax+8],0
mov    ecx,dword ptr [ebp-8]
mov    dword ptr [ecx+4],0
mov    eax,dword ptr [ebp-8]
mov     esp,ebp
pop     ebp
ret
```

- The code colored in **blue** is the code added by the compiler to initialize the **vfptr** pointer.

How virtual methods are modeled by C++ compiler

- ❖ The code added by the compiler to initialize the *vfptr* pointer will be added for every defined constructor.

App.cpp

```
class A
{
public:
    int x, y;
    virtual int Calcul() {return x+y;}
    A() { x = y = 0; }
    A(const A& a) { x = a.x; y = a.y; }
};
void main()
{
    A a;
    A a2 = a;
}
```

In this case the code for *vfptr* initialization will be added for both the default constructor and the copy constructor.

How virtual methods are modeled by C++ compiler

- ❖ However, in case of the *assignment operator* the compiler will not add any special code to initialize the *vptr* pointer.

App.cpp

```
class A
{
public:
    int x, y;
    virtual int Calcul() {return x+y;}
    A() { x = y = 0; }
    A& operator = (A &a) { x = a.x; y = a.y; return *this;}
};
void main()
{
    A a;
    A a2;
    a2 = a;
}
```

How virtual methods are modeled by C++ compiler

- ❖ A virtual method is called using its reference from the *vfptr* table only if the object is a pointer.

App.cpp	Disasm
<pre>class A { public: int x, y; virtual int Calcul() {return x+y;} A() { x = y = 0; } }; void main() { A a; a.x = 1; a.y = 2; a.Calcul(); }</pre>	<pre>A a; lea ecx,[a] call A::A a.x = 1; mov dword ptr [ebp-10h],1 a.y = 2; mov dword ptr [ebp-0Ch],2 a.Calcul(); lea ecx,[a] call A::Calcul</pre>

- ❖ In this case, even if *Calcul* method is *virtual* as it called directly with an object , the compiler will not generate code that will find out its address from the *vfptr* table (it will use the method *Calcul* exact address).

How virtual methods are modeled by C++ compiler

- ❖ A virtual method is called using its reference from the *vfptr* table only if the object is a pointer.

App.cpp	Disasm
<pre>class A { public: int x, y; virtual int Calcul() {return x+y;} A() { x = y = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>A a; lea ecx,[a] call A::A a.x = 1; mov dword ptr [ebp-10h],1 a.y = 2; mov dword ptr [ebp-0Ch],2 A* a2 = &a; lea eax,[a] mov dword ptr [a2],eax a2->Calcul(); mov eax,dword ptr [a2] mov edx,dword ptr [eax] mov ecx,dword ptr [a2] mov eax,dword ptr [edx] call eax</pre> <p>❖ In this case <i>vfptr</i> is used to find out <i>Calcul</i> method address.</p>

EAX = address of a2

EDX = address of VFPTTR

EAX = address of first function from VFPTTR

How virtual methods are modeled by C++ compiler

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { };</pre>

How virtual methods are modeled by C++ compiler

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); }; </pre>

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
    int x;
    virtual int Calcul() {return 0;}
    A() { x = 0; }
};
void main()
{
    A a;
    a.x = 1;
    a.y = 2;
    A* a2 = &a;
    a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {
    int (*Calcul) ();
};

class A {
public:
    int x;
}
```

How virtual methods are modeled by C++ compiler

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); }; class A { public: A_VirtualFunctions *vfPtr; int x; }</pre>

How virtual methods are modeled by C++ compiler

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); }; class A { public: A_VirtualFunctions *vfPtr; int x; int A_Calcul() { return 0; } }</pre>

How virtual methods are modeled by C++ compiler

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); }; class A { public: A_VirtualFunctions *vfPtr; int x; int A_Calcul() { return 0; } A_VirtualFunctions Global_A_vfPtr; Global_A_vfPtr.Calcul = &A::A_Calcul;</pre>

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
    int x;
    virtual int Calcul() {return 0;}
    A() { x = 0; }
};

void main()
{
    A a;
    a.x = 1;
    a.y = 2;
    A* a2 = &a;
    a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {
    int (*Calcul) ();
};

class A {
public:
    A_VirtualFunctions *vfPtr;
    int x;
    int A Calcul() { return 0; }
    A() {
        x = 0;
    }
};

A_VirtualFunctions Global_A_vfPtr;
Global_A_vfPtr.Calcul = &A::A_Calcul;
```

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
    int x;
    virtual int Calcul() {return 0;}
    A() { x = 0; }
};

void main()
{
    A a;
    a.x = 1;
    a.y = 2;
    A* a2 = &a;
    a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {
    int (*Calcul) ();
};

class A {
public:
    A_VirtualFunctions *vfPtr;
    int x;
    int A_Calcul() { return 0; }
    A() {
        vfPtr = &Global_A_vfPtr;
        x = 0;
    }
};

A_VirtualFunctions Global_A_vfPtr;
Global_A_vfPtr.Calcul = &A::A_Calcul;
```

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
    int x;
    virtual int Calcul() {return 0;}
    A() { x = 0; }
};

void main()
{
    A a;
    a.x = 1;
    a.y = 2;
    A* a2 = &a;
    a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {
    int (*Calcul) ();
};

class A {
public:
    A_VirtualFunctions *vfPtr;
    int x;
    int A_Calcul() { return 0; }
    A() {
        vfPtr = &Global_A_vfPtr;
        x = 0;
    }
};

A_VirtualFunctions Global_A_vfPtr;
Global_A_vfPtr.Calcul = &A::A_Calcul;
```

```
void main()
{
    A a;
    a.x = 1;
    a.y = 2;
    A* a2 = &a;
}
```

How virtual methods are modeled by C++ compiler

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); }; class A { public: A_VirtualFunctions *vfPtr; int x; int A_Calcul() { return 0; } A() { vfPtr = &Global_A_vfPtr; x = 0; } }; A_VirtualFunctions Global_A_vfPtr; Global_A_vfPtr.Calcul = &A::A_Calcul; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->vfPtr->Calcul(); }</pre>

How virtual methods are modeled by C++ compiler

- ❖ Keep in mind the **vfptr** is just a pointer. As such, it can be changed during execution

App.cpp

```
class A
{
public:
    int x;
    virtual void Print() { printf("A"); }
};
class B
{
public:
    int x;
    virtual void Print() { printf("B"); }
};
void main()
{
    A a;
    B b;
    A* a2 = &a;
    a.Print();
    a2->Print();
}
```

- ❖ This code will print “**AA**” on the screen. First time when method **Print** is called directly (“*a.Print()*”), *second time when method Print is called using the vfptr pointer (“*a2->Print()*”)*

How virtual methods are modeled by C++ compiler

- ❖ Keep in mind the **vfptr** is just a pointer. As such, it can be changed during execution

App.cpp

```
class A
{
public:
    int x;
    virtual void Print() { printf("A"); }
};
class B
{
public:
    int x;
    virtual void Print() { printf("B"); }
};
void main()
{
    A a;
    B b;
    memcpy(&a, &b, sizeof(void*));
    A* a2 = &a;
    a.Print();
    a2->Print();
}
```

- ❖ This code will however print “**AB**”. Using **memcpy** function allow us to overwrite the actual **vfptr**-ul of object “*a*” with the one from object “*b*”. As method *Print* has the same signature in both classes (*A* and *B*) the result will be “*AB*”

How virtual methods are modeled by C++ compiler

- ❖ Keep in mind the **vfptr** is just a pointer. As such, it can be changed during execution

App.cpp

```
class A
{
public:
    int x;
    virtual void Print() { printf("A"); }
};
class B
{
public:
    int x;
    virtual void Print() { printf("B"); }
};
void main()
{
    A a;
    B b;
    memcpy(&a, &b, sizeof(void*));
    A* a2 = &a;
    A a3 = (*a2);
    A *a4 = &a3;
    a4->Print();
}
```

- ❖ Every constructor called will set the **vfptr** to its correct value. In this case , “**A a3=(*a2)**” will call the copy constructor for class **A** and will set the **vfptr** for local variable **a3** correctly.
- ❖ As a result, this code will print “**A**” on the screen , even if “**a2**” has the **vfptr** of “**b**”

How virtual methods are modeled by C++ compiler

- ❖ A virtual function can be overwritten in the derived class.

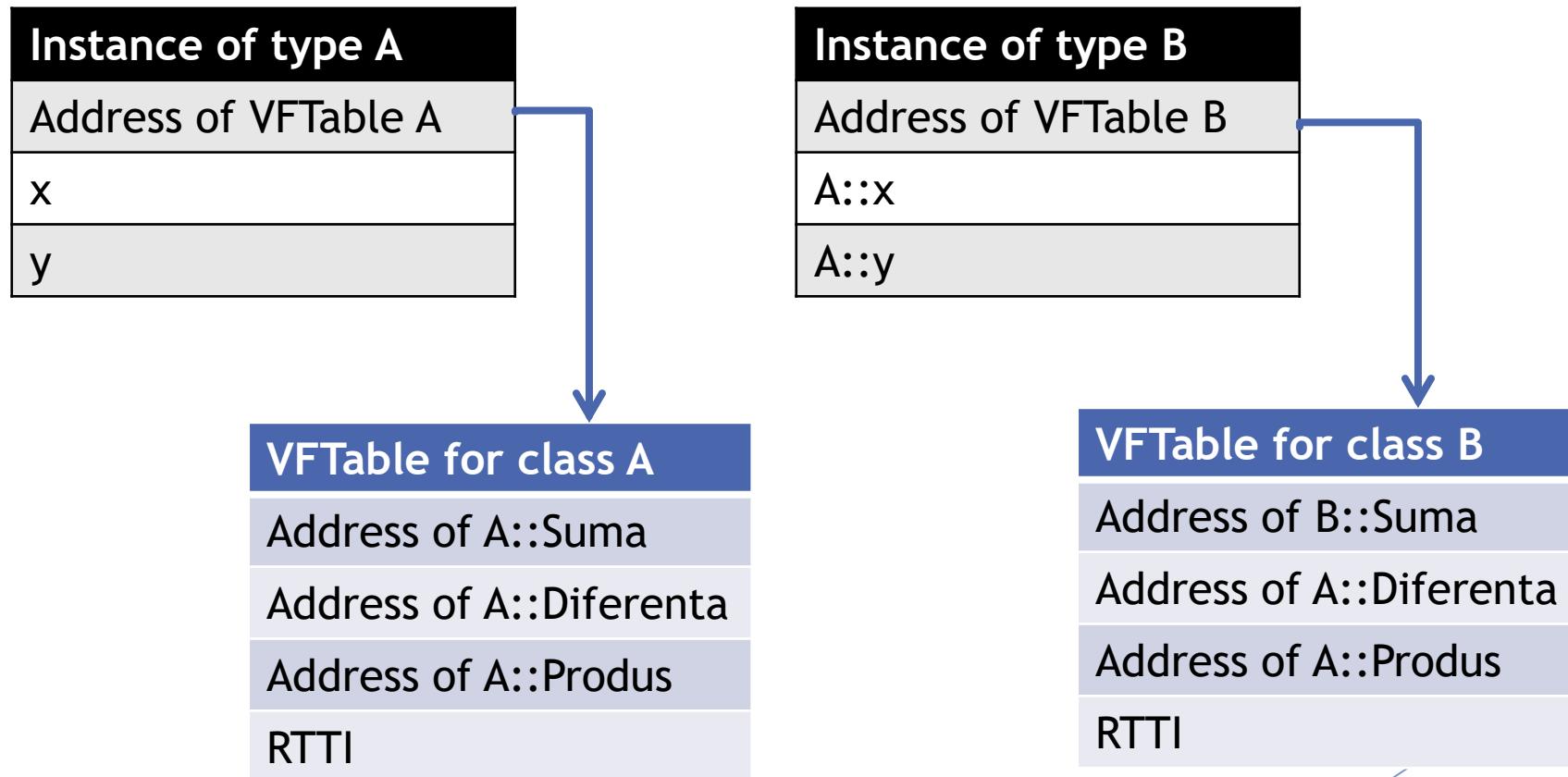
App.cpp

```
class A
{
public:
    int x, y;
    virtual int Suma() { return x + y; }
    virtual int Diferenta() { return x - y; }
    virtual int Produs() { return x*y; }
};
class B : public A
{
public:
    int Suma() { return 1; }
};

void main()
{
    B b;
    b.x = 1;
    b.y = 2;
    A* a;
    a = &b;
    int x = a->Suma();
}
```

- ❖ In this case, “**x**” will be 1 as “*a*” is in fact an object of type “*b*” that has overwrite method “*Suma*”
- ❖ For the rest of the methods (*Diferenta* and *Produs*) the behavior will be identical to the one from the base class (A).

How virtual methods are modeled by C++ compiler



How virtual methods are modeled by C++ compiler

- ❖ A derived class can also add other (new) virtual methods .

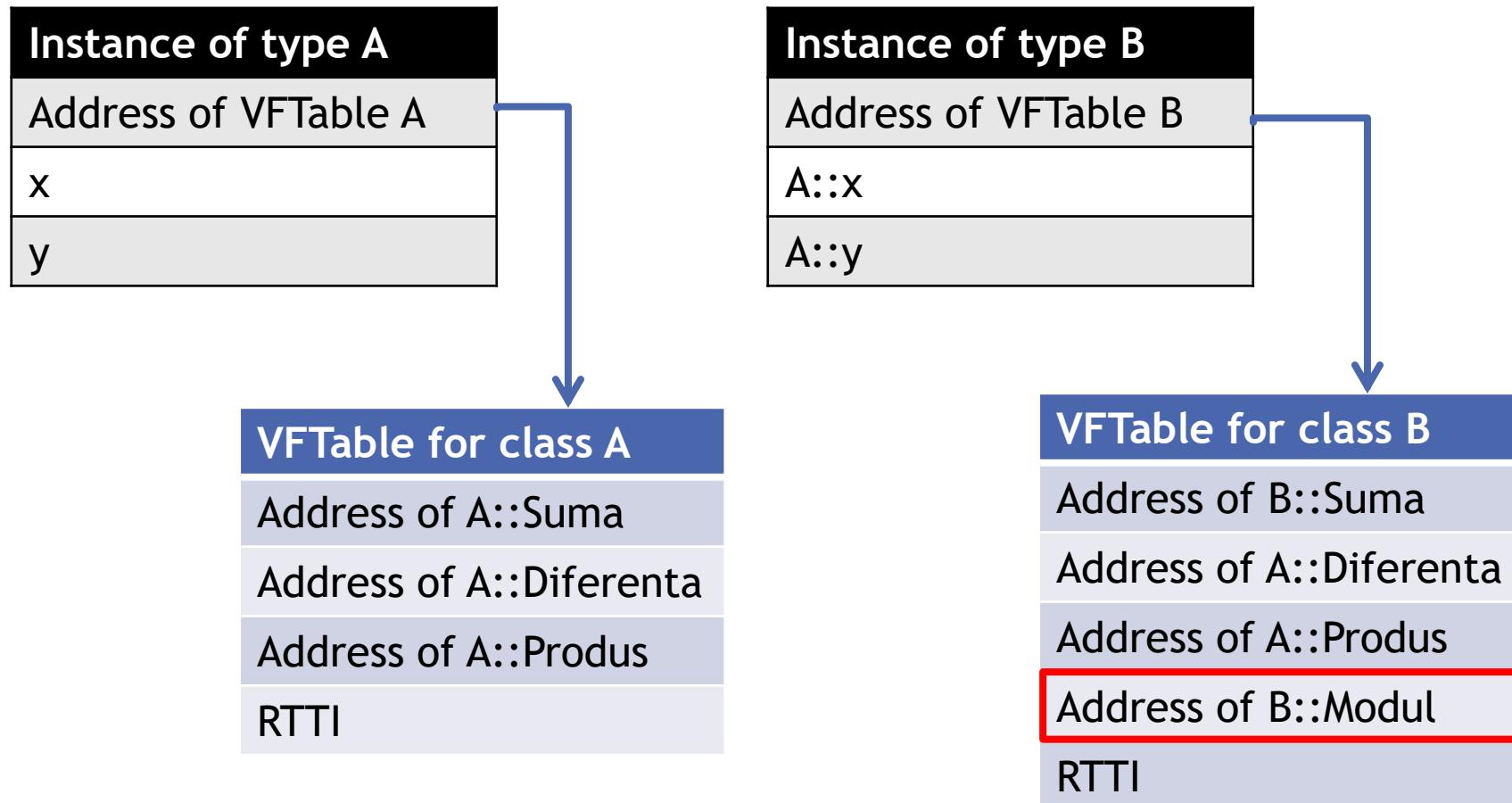
App.cpp

```
class A
{
public:
    int x, y;
    virtual int Suma() { return x + y; }
    virtual int Diferenta() { return x - y; }
    virtual int Produs() { return x*y; }
};
class B : public A
{
public:
    int Suma() { return 1; }
    virtual int Modul() { return 0; }
};

void main()
{}
```

- ❖ In this case, class *B* also have a new virtual method called “*Module*”) that is not present on class *A*.
- ❖ This means that any class that will be derived from *B* will have this method as well.

How virtual methods are modeled by C++ compiler



How virtual methods are modeled by C++ compiler

- When a class is derived from two(or more) classes that have virtual functions, the compiler creates multiple **vfptr** pointers (one for each base class).

App.cpp

```
class A {  
public:  
    int a1;  
    virtual int Suma() { return 1; }  
    virtual int Diferenta() { return 2; }  
};  
class B {  
public:  
    int b1,b2;  
    virtual int Inmultire() { return 3; }  
    virtual int Impartire() { return 4; }  
};  
class C : public A, public B {  
public:  
    int x, y;  
};  
void main() {  
    C c;  
    C *cptr = &c;  
    cptr->Impartire();  
    cptr->Diferenta();  
}
```

Disasm

```
cptr->Impartire();  
mov      ecx,dword ptr [cptr]  
add      ecx,8 //this for type B  
mov      eax,dword ptr [cptr]  
mov      edx,dword ptr [eax+8]  
mov      eax,dword ptr [edx+4]  
call    eax  
  
cptr->Diferenta();  
mov      eax,dword ptr [cptr]  
mov      edx,dword ptr [eax]  
mov      ecx,dword ptr [cptr]  
mov      eax,dword ptr [edx+4]  
call    eax
```

How virtual methods are modeled by C++ compiler

- When a class is derived from two(or more) classes that have virtual functions, the compiler creates multiple **vfptr** pointers (one for each base class).

App.cpp

```
class A {  
public:  
    int a1;  
    virtual int Suma() { return 1; }  
    virtual int Diferenta() { return 2; }  
};  
class B {  
public:  
    int b1,b2;  
    virtual int Inmultire() { return 3; }  
    virtual int Impartire() { return 4; }  
};  
class C : public A, public B {  
public:  
    int x, y;  
};  
void main() {  
    C c;  
    C *cptr = &c;  
    cptr->Impartire();  
    cptr->Diferenta();  
}
```

Offset	Field
+ 0	A::vfptr
+ 4	A::a1
+ 8	B::vfptr
+ 12	B::b1
+ 16	B::b2
+ 20	C::x
+ 24	C::y

VFTable for class A

Address of A::Suma
Address of A::Diferenta
RTTI

VFTable for class B

Address of B::Inmultire
Address of B::Impartire
RTTI

How virtual methods are modeled by C++ compiler

- The same memory alignment is used for classes derived out of class **C** (e.g. in this example, class **D**)

```
App.cpp

class A {
public:
    int a1;
    virtual int Suma() { return 1; }
    virtual int Diferenta() { return 2; }
};
class B {
public:
    int b1,b2;
    virtual int Inmultire() { return 3; }
    virtual int Impartire() { return 4; }
};
class C : public A, public B {
public:
    int x, y;
};
class D : public C {
public:
    int d1;
};
```

Offset	Field
+ 0	A::vfptr
+ 4	A::a1
+ 8	B::vfptr
+ 12	B::b1
+ 16	B::b2
+ 20	C::x
+ 24	C::y
+ 28	D::d1

VFTable for class A

Address of A::Suma
Address of A::Diferenta
RTTI

VFTable for class B

Address of B::Inmultire
Address of B::Impartire
RTTI

A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented diagonally, creating a sense of depth and movement.

► Covariance

Covariance

- ❖ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int a1, a2;
    virtual A* clone() { return new A(); }
};
class B : public A
{
public:
    int b1, b2;
    virtual A* clone() { return new B(); }
};
void main()
{
    B *b = new B();
    B *ptrB;
    ptrB = b->clone();
}
```

- ❖ This code will not compile. However, in reality “*b->clone()*” returns an object of *type B* so it should work.

error C2440: '=': cannot convert from 'A *' to 'B *'
note: Cast from base to derived requires dynamic_cast
or static_cast

Covariance

- ❖ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int a1, a2;
    virtual A* clone() { return new A(); }
};
class B : public A
{
public:
    int b1, b2;
    virtual A* clone() { return new B(); }
};
void main()
{
    B *b = new B();
    B *ptrB;
    ptrB = b->clone();
}
```

- ❖ We have two solutions for this problem:

Covariance

- ❖ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int a1, a2;
    virtual A* clone() { return new A(); }
};
class B : public A
{
public:
    int b1, b2;
    virtual A* clone() { return new B(); }
};
void main()
{
    B *b = new B();
    B *ptrB;
    ptrB = (B*) b->clone();
```

- ❖ We have two solutions for this problem:

1. Use an explicit cast and convert the pointer from *A** to *B**

Covariance

- ❖ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int a1, a2;
    virtual A* clone() { return new A(); }
};
class B : public A
{
public:
    int b1, b2;
    virtual B* clone() { return new B(); }
};
void main()
{
    B *b = new B();
    B *ptrB;
    ptrB = b->clone();
}
```

- ❖ We have two solutions for this problem:

1. Use an explicit cast and convert the pointer from *A** to *B**
2. Use **covariance**. This means that we can modify the return type of the method *clone* in class *B* to return a *B* pointer* instead of an *A* pointer*.

Covariance

- ❖ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int a1, a2;
    virtual A* clone() { return new A(); }
};

class B : public A
{
public:
    int b1, b2;
    virtual B* clone() { return new B(); }
};

void main()
{
    B *b = new B();
    B *ptrB;
    ptrB = b->clone();
    A *a = (A*)b;
    ptrB = (B*)a->clone();
}
```

- ❖ We have two solutions for this problem:

1. Use an explicit cast and convert the pointer from *A** to *B**
2. Use **covariance**. This means that we can modify the return type of the method *clone* in class *B* to return a *B* pointer* instead of an *A* pointer*.

Covariance is related to the pointer type. In this case, even if the compiler calls “*B::clone*”, the expected value is *A** (specific to a *A* pointer* that is “*aA::clone*”)

Covariance

- ❖ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int a1, a2;
    virtual A* clone() { return new A(); }
};

class B : public A
{
public:
    int b1, b2;
    virtual B* clone() { return new B(); }
};

void main()
{
    B *b = new B();
    B *ptrB;
    ptrB = b->clone();
    A *a = (A*)b;
    ptrB = a->clone();
}
```

- ❖ We have two solutions for this problem:

1. Use an explicit cast and convert the pointer from *A** to *B**
2. Use **covariance**. This means that we can modify the return type of the method *clone* in class *B* to return a *B* pointer* instead of an *A* pointer*.

That is why this code will NOT compile, as the result for *a->clone* is *A** and not *B**. During execution, "*B::clone*" will be called, nevertheless.

Covariance

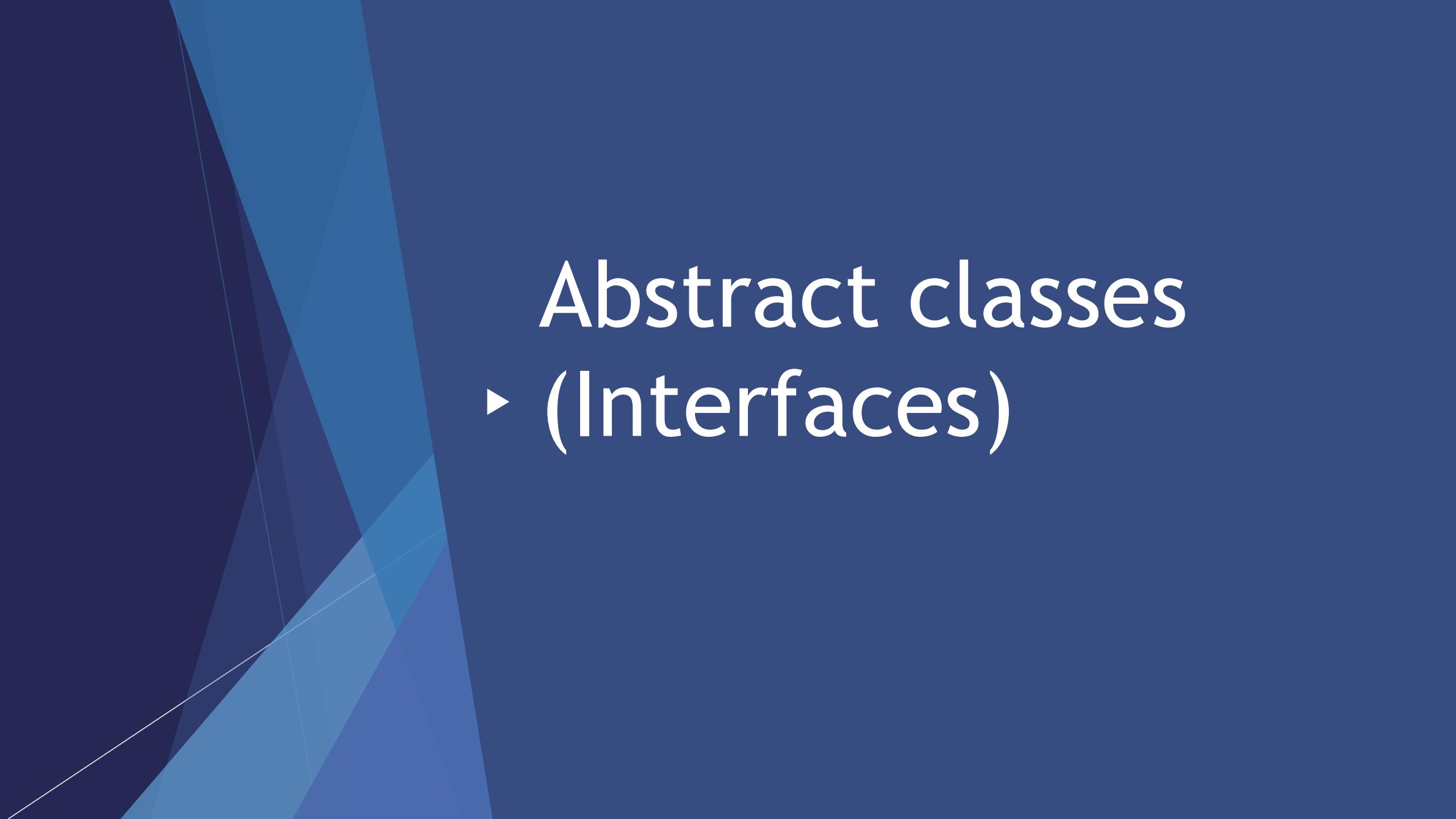
- ❖ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int a1, a2;
    virtual A* clone() { return new A(); }
};
class B : public A
{
public:
    int b1, b2;
    virtual int* clone() { return new int(); }
};
void main()
{
    ...
}
```

error C2555: 'B::clone': overriding
virtual function return type differs
and is not covariant from
'A::clone'

- ❖ This code will not compile. The return type for virtual functions can be changed, but only to a type that is derived from the return type of the virtual method described in the base class. In this case, `int*` is not derived from `A*`

The background features a dark blue gradient with a subtle geometric pattern of lighter blue and white triangles on the left side.

Abstract classes ► (Interfaces)

Abstract classes (Interfaces)

- ▶ In C++ we can define a virtual method without a body (it is called a **pure virtual method** and it is defined by adding “=0” at the end of its definition).
- ▶ If a class contains a pure virtual method, that class is an abstract class (a class that can not be instantiated). In other languages this concept is similar to the concept of an interface.
- ▶ Having a pure virtual method forces the one that implements a derived class to implement that method as well if he/she would like to create an instance from the newly created class.

App.cpp

```
class A
{
public:
    int a1, a2, a3;
    virtual void Set() = 0;
};
```

```
void main()
{
    A a;
```

❖ The code will not compile as “A” is an abstract class.

```
error C2259: 'A': cannot instantiate abstract class
note: due to following members:
note: 'void A::Set(void)': is abstract
note: see declaration of 'A::Set'
```

Abstract classes (Interfaces)

- ▶ In C++ we can define a virtual method without a body (it is called a **pure virtual method** and it is defined by adding “=0” at the end of its definition).

App.cpp

```
class A
{
public:
    int a1, a2, a3;
    virtual void Set() = 0;
};
class B: A
{
public:
    int a1, a2, a3;
    void Set(){...};
}
void main()
{
    B b;
```

- ❖ This code will compile because class *B* has an implementation for method *Set*
- ❖ In order to be able to create an instance of a class, all of its pure virtual methods (defined in that class or obtained via inheritance) **MUST** be implemented !

Abstract classes (Interfaces)

- ▶ In C++ we can define a virtual method without a body (it is called a **pure virtual method** and it is defined by adding “=0” at the end of its definition).

App.cpp

```
class A
{
public:
    int a1, a2, a3;
    virtual void Set() = 0;
};
class B: A
{
public:
    int a1, a2, a3;
    void Set(){...};
}
void main()
{
    B b;
    A* a;
}
```

- ❖ This code will however compile. It is possible (and recommended whenever working with polymorphism) to create a pointer towards an abstract class (in this case an *A** pointer).

Abstract classes (Interfaces)

- ▶ Other languages (such as Java or C#) have a similar concept called *interface* (primarily used in these languages to avoid multiple inheritance).
- ▶ *interfaces* are however different from an abstract class. An interface **CAN NOT** have data members, or methods that are not pure virtual. An abstract class is a class that has at least one pure virtual method. An abstract class can have methods, constructors, destructor or data members.
- ▶ In C++ it is often easier to use *struct* instead of class to describe an interface due to the fact that the default access modifier is *public*
- ▶ Cl.exe (Microsoft) has a keyword (*__interface*) that works like an interface (allows you to create one). However, this is not part of the standard.



Memory alignment in case of ► inheritance

Memory alignment in case of inheritance

```
class A
{
public:
    int a1,a2,a3;
};
```

`sizeof(A) = 12`

```
class B: public A
{
public:
    int b1,b2
};
```

`sizeof(B) = 20`

Offset	Field	C1	C2
+ 0	A::a1	A	B
+ 4	A::a2		
+ 8	A::a3		
+ 12	B::b1		
+ 16	B::b2		

Memory alignment in case of inheritance

```
class A
{
public:
    int a1,a2,a3;
};
```

```
sizeof(A) = 12
```

```
class B: public A
{
public:
    int b1,b2
};
```

```
sizeof(B) = 20
```

Offset	Field	C1	C2
+ 0	A::a1		
+ 4	A::a2		
+ 8	A::a3		
+ 12	B::b1		
+ 16	B::b2		

A
B

Memory alignment in case of inheritance

```
class A
{
public:
    int a1,a2,a3;
};
```

`sizeof(A) = 12`

```
class B: public A
{
public:
    int b1,b2
};
```

`sizeof(B) = 20`

Offset	Field	C1	C2
+ 0	A::a1		
+ 4	A::a2		
+ 8	A::a3		
+ 12	B::b1	A	B
+ 16	B::b2		

Memory alignment in case of inheritance

```
class A  
{  
public:  
    int a1,a2,a3;  
};
```

`sizeof(A) = 12`

```
class B:  
{  
public:  
    int b1,b2;  
};
```

`sizeof(B) = 8`

```
class C:public A,B  
{  
public:  
    int c1,c2;  
};
```

`sizeof(C) = 28`

Offset	Field	C1	C2	C3
+ 0	A::a1	A		
+ 4	A::a2			
+ 8	A::a3			
+ 12	B::b1		B	
+ 16	B::b2			
+20	C::c1			C
+24	C::c2			

Memory alignment in case of inheritance

```
class A  
{  
public:  
    int a1,a2,a3;  
};
```

`sizeof(A) = 12`

```
class B:  
{  
public:  
    int b1,b2;  
};
```

`sizeof(B) = 8`

When building a derived class in memory, if the inheritance is does not contain the virtual specifier, will be done using the left-to-right rule for any base classes.

```
class C:public B,A  
{  
public:  
    int c1,c2;  
};
```

`sizeof(C) = 28`

Offset	Field	C1	C2	C3
+ 0	B::b1			
+ 4	B::b2	B		
+ 8	A::a1			
+ 12	A::a2		A	
+ 16	A::a3			C
+20	C::c1			
+24	C::c2			

Memory alignment in case of inheritance

warning C4584: 'C' : base-class 'A' is already a base-class of 'B'.

- ❖ Multiple inheritance can create ambiguous situations. For example, in this case the fields from class A are copied twice in class C.

App.cpp

```
class A
{
public:
    int a1, a2, a3;
};
class B: public A
{
public:
    int b1, b2;
};
class C : public A, public B
{
public:
    int c1, c2;
};
void main()
{}
```

Offset	Field	C1	C2	C3
+0	A::a1	A		
+4	A::a2			
+8	A::a3			
+12	B::A::a1	B::A		
+16	B::A::a2			
+20	B::A::a3			
+24	B::b1			
+28	B::b2			
+32	C::c1			
+36	C::c2			

Memory alignment in case of inheritance

- ❖ Multiple inheritance can create ambiguous situations. For example, in this case the fields from class **A** are copied twice in class **C**.

App.cpp

```
class A {  
public:  
    int a1, a2, a3;  
};  
class B : public A {  
public:  
    int b1, b2;  
};  
class C : public A, public B {  
public:  
    int c1, c2;  
};  
void main()  
{  
    C c:  
    c.a1 = 10;  
}
```

This is an ambiguous case. “**c.a1 = 10**” can refer to the member “**a1**” from the direct inheritance of class **A**, or the member “**a1**” from the direct inheritance of class **B** that in terms inherits class **A**.

This code will NOT compile !!!

```
warning C4584: 'C': base-class 'A' is already a base-class of 'B'  
note: see declaration of 'A'  
note: see declaration of 'B'  
-----  
error C2385: ambiguous access of 'a1'  
note: could be the 'a1' in base 'A'  
note: or could be the 'a1' in base 'A'
```

Memory alignment in case of inheritance

- ❖ Multiple inheritance can create ambiguous situations. For example, in this case the fields from class **A** are copied twice in class **C**.

App.cpp

```
class A {  
public:  
    int a1, a2, a3;  
};  
class B : public A {  
public:  
    int b1, b2;  
};  
class C : public A, public B {  
public:  
    int c1, c2;  
};  
void main()  
{  
    C c;  
    c.A::a1 = 10;  
    c.B::A::a1 = 20;  
}
```

- ❖ The solution is to describe any field/data member using its full scope. For example:
 - “**c.A::a1**” means data member “a1” from the direct inheritance of “A” in class “C”
 - “**c.B::A::a1**” means data member “a1” from the inheritance of “A” in class “B” that is directly inherit by class “C”
- ❖ What can we do if we want to have only one copy of the fields from class “A” in our object ?
- ❖ This problem is also known as the “*Diamond Problem*”

Memory alignment in case of inheritance

- ❖ Multiple inheritance can create ambiguous situations. For example, in this case the fields from class **A** are copied twice in class **C**.

App.cpp

```
class A {
public:
    int a1, a2, a3;
};
class B : public virtual A {
public:
    int b1, b2;
};
class C : public virtual A, public B {
public:
    int c1, c2;
};
void main()
{
    C c;
    c.a1 = 10;
    c.a2 = 20;
}
```

- ❖ One solution to this problem is to use the **virtual** specifier when deriving from a class. In this case, class “A” is inherited virtually (meaning that its fields must be added once).
- ❖ For this code to work, both “C” and “B” class need to inherit class “A” using **virtual** keyword.

Memory alignment in case of inheritance

- ❖ Just like in the case of virtual methods, if no constructor is present, one will be created by the compiler. However, this constructor is a little bit different than the others (as it has one parameter of type bool).

App.cpp	Disasm
<pre>class A { public: int a1, a2, a3; }; class B : public virtual A { public: int b1, b2; }; class C : public virtual A, public B { public: int c1, c2; }; void main() { C c; c.a1 = 10; c.b1 = 20; }</pre>	<p>C c;</p> <pre>push 1<-->ecx,[c] TRUE lea ecx,[c] call C::C c.a1 = 10; mov eax,dword ptr [c] mov ecx,dword ptr [eax+4] mov dword ptr [c+ecx],10 c.b1 = 20; mov dword ptr [c+20],20</pre>

Memory alignment in case of inheritance

- ❖ The first parameter, tells the constructor if a special table with indexes needs to be created or not !

App.cpp

```
class A {  
public:  
    int a1, a2, a3;  
};  
class B : public virtual A {  
public:  
    int b1, b2;  
};  
class C : public virtual A, public B {  
public:  
    int c1, c2;  
};  
void main()  
{  
    C c;  
    c.a1 = 10;  
    c.b1 = 20;  
}
```

Disasm C::C

```
push    ebp  
mov     ebp,esp  
mov     dword ptr [this],ecx  
cmp     dword ptr [ebp+8],0  
je      DONT_SET_VAR_PTR  
mov     eax,dword ptr [this]  
mov     dword ptr [eax],addr_index  
DONT_SET_VAR_PTR:  
push    0  
mov     ecx,dword ptr [this]  
call    B::B  
mov     eax,dword ptr [this]  
mov     esp,ebp  
pop    ebp  
ret     4
```

Memory alignment in case of inheritance

- Once the constructor is called, an object that has virtual inheritance will look as follows:

Offset	Field	C1	C2
+ 0	Ptr Class C Variable Offsets Table		
+ 4	B::b1		
+ 8	B::b2	B	
+ 12	C::c1		C
+ 16	C::c2		
+ 20	A::a1		
+ 24	A::a2	A	
+ 28	A::a3		

Offset	Offset relative to C
+ 0	0
+ 4	Virtual A 20

Memory alignment in case of inheritance

- ❖ Accessing a data member / field that benefits from the *virtual* inheritance, is done in 3 steps (not in one) in the following way:

App.cpp

```
class A { ... }
class B: public virtual A { ... }
class C : public virtual A, public B { ... }
void main()
{
    C c;
    c.a1 = 10;
}
```

Disasm

```
c.a1 = 10;
mov     eax,dword ptr [c]
mov     ecx,dword ptr [eax+4]
mov     dword ptr [c+ecx],10
```

Memory alignment in case of inheritance

- In the first step, EAX register gets the pointer to the table where offsets of data member/fields from A class are stored

App.cpp			
Offset	Field	C1	C2
+ 0	Ptr Class C Variable Offsets Table		
+ 4	B::b1		
+ 8	B::b2		
+ 12	C::c1		
+ 16	C::c2		
+ 20	A::a1		
+ 24	A::a2		
+ 28	A::a3		

A B C

Disasm			
c.a1 = 10;			
mov		eax,dword ptr [c]	
mov		ecx,dword ptr [eax+4]	
mov		dword ptr [c+ecx],10	

Memory alignment in case of inheritance

- ❖ Second step - ECX gets the value from the second index in that table (+4), more exactly value 20 (that reflects the offset of "A" from the beginning of "C")

App.cpp			
Offset	Field	C1	C2
+ 0	Ptr Class C Variable Offsets Table		
+ 4	B::b1		
+ 8	B::b2		
+ 12	C::c1		
+ 16	C::c2		
+ 20	A::a1		
+ 24	A::a2		
+ 28	A::a3		

B C
 A

Disasm	
c.a1 = 10;	
mov eax,dword ptr [c]	
mov ecx,dword ptr [eax+4]	
mov dword ptr [c+ecx],10	

Offset	Offset relative to C
+ 0	0
+ 4	Virtual A 20

Memory alignment in case of inheritance

- ❖ Last step, we use “ECX” register as an offset to access A::a1 from the beginning of local variable “c”.

App.cpp			
Offset	Field	C1	C2
+ 0	Ptr Class C Variable Offsets Table		
+ 4	B::b1		
+ 8	B::b2		
+ 12	C::c1		
+ 16	C::c2		
+ 20	A::a1		
+ 24	A::a2		
+ 28	A::a3		

Diagram illustrating memory layout:

- Class A has fields A::a1, A::a2, and A::a3.
- Class B has fields B::b1 and B::b2.
- Class C has fields C::c1 and C::c2.
- The pointer at offset +0 points to the Class C Variable Offsets Table.
- Offsets +4 to +16 are relative to the start of Class C.
- Offset +20 is relative to the start of Class A.

Disasm

```
c.a1 = 10;
mov    eax,dword ptr [c]
mov    ecx,dword ptr [eax+4]
mov    dword ptr [c+ecx],10
```

Offset	Offset relative to C
+ 0	0
+ 4	Virtual A

Memory alignment in case of inheritance

- Fields/Data members that are obtained via virtual inheritance are usually added at the end of the class alignment.

App.cpp		Offset	Field
<pre>class A { ... } class B: public virtual A { ... } class C : public virtual A, public B { ... }</pre>		+ 0	ptr class C virtual members offsets
		+ 4	C::B::b1
		+ 8	C::B::b2
		+ 12	C::c1
		+ 16	C::c2
		+ 20	A::a1 (virtual A from C)
		+ 24	A::a2 (virtual A from C)
		+ 28	A::a3 (virtual A from C)

Offset	Offset relative to C
+ 0	0
+ 4	Virtual A 20

Memory alignment in case of inheritance

- If we use virtual inheritance when deriving “C” from “B” (in addition to the usage of virtual inheritance for class “A”) we will obtain the following alignment:

App.cpp		Offset	Field
<pre>class A { ... } class B: public virtual A { ... } class C : public virtual A, public virtual B { ... }</pre>		+ 0	ptr class C virtual members offsets
		+ 4	C::c1
		+ 8	C::c2
		+ 12	A::a1 (virtual A from C)
		+ 16	A::a2 (virtual A from C)
		+ 20	A::a3 (virtual A from C)
		+ 24	ptr class B virtual members offsets
		+ 28	B::b1 (virtual B from C)
		+ 32	B::b2 (virtual B from C)

Offset	Offset relative to C
+ 0	0
+ 4	Virtual A
+ 8	Virtual B

Memory alignment in case of inheritance

- In case of the index table for class "B", the offset "-12" refers to the position of "A" class (also obtained via virtual inheritance) relative to B with respect to C class (24 (offset of B) - 12 = 12 (offset of A))

App.cpp

```
class A
{ ... }
class B : public virtual A
{ ... }
class C : public virtual A,
           public virtual B
{ ... }
```

Offset Field

+ 0 ptr class C virtual members offsets

+ 4 C::c1

+ 8 C::c2

+ 12 A::a1 (virtual A from C)

+ 16 A::a2 (virtual A from C)

+ 20 A::a3 (virtual A from C)

+ 24 ptr class B virtual members offsets

+ 28 B::b1 (virtual B from C)

+ 32 B::b2 (virtual B from C)

Offset	Offset relative la B
+ 0	0
+ 4	Virtual A -12

Memory alignment in case of inheritance

- ❖ If we make only the inheritance of B from C to be virtual, the memory alignment is as follows:

App.cpp

```
class A
{ ... }
class B: public A
{ ... }
class C : public A,
           public virtual B
{ ... }
```

Offset	Offset relative la C
+ 0	-12
+ 4	Virtual B

Offset	Field
+ 0	A::a1
+ 4	A::a1
+ 8	A::a3
+ 12	ptr class C virtual members offsets
+ 16	C::c1
+ 20	C::c2
+ 24	B::A::a1
+ 28	B::A::a2
+ 32	B::A::a3
+ 36	B::b1
+ 40	B::b2

Memory alignment in case of inheritance

- If we make only the inheritance of B from C to be virtual, the memory alignment is as follows:

App.cpp

```
class A
{ ... }
class B: public A
{ ... }
class C : public A,
           public virtual B
{ ... }
```

Offset	Field
+ 0	A::a1
+ 4	A::a1
+ 8	A::a3
+ 12	ptr class C virtual members offsets
+ 16	C::c1
+ 32	B::A::a3
+ 36	B::b1
+ 40	B::b2

Offset	Offset relative la C
+ 0	-12
+ 4	Virtual B

First index (+0 offset, value -12) represents the offset of object C relative to the table of indexes).

It is usually 0 (as this table is the first entry), however in this case it is a negative value.

Memory alignment in case of inheritance

- If we make only the inheritance of B from C to be virtual, the memory alignment is as follows:

App.cpp

```
class A
{ ... }
class B: public A
{ ... }
class C : public A,
           public virtual B
{ ... }
```

Offset	Field
+ 0	A::a1
+ 4	A::a1
+ 8	A::a3
+ 12	ptr class C virtual members offsets
+ 16	C::c1
+ 20	C::c2

Offset	Offset relative la C
+ 0	-12
+ 4	Virtual B

The second offset (+4, value +12) reflects the position of B relative to the offset of the index table ($12+12=24$).

+ 36 B::b1

+ 40 B::b2

Q & A

OOP

Gavrilut Dragos
Course 6

Summary

- ▶ Casts
- ▶ Macros
- ▶ Macros vs Inline
- ▶ Literals
- ▶ Templates
- ▶ Function templates
- ▶ Class templates
- ▶ Template specialization
- ▶ Compile-time assertion checking

The background features a dark blue gradient with a subtle geometric pattern of lighter blue triangles and lines.

► Casts

Casts

- ▶ Assuming that a class A is derived from a class B, than it is possible for an object of type A to be converted to an object of type B.
- ▶ This is a normal behavior (obviously A contains every member defined in B).
- ▶ The conversion rules are as follows:
 - ❖ It will always be possible to convert a class to any of the classes that it inherits
 - ❖ It is not possible to convert from a base class to one of the classes that inherits it without an explicit cast
 - ❖ If the cast operator is overwritten, none of the above rules apply.

Casts

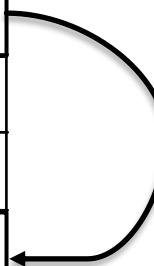
- ▶ Let's consider the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };
void main(void) {
    C* c = new C()
    B* b = &c;
}
```

- ▶ Assuming the pointer “c” points to the offset 300000, what do we need to do to obtain a pointer to an object of type B* ?

Offset	Field/Var
100000	c*
100004	b*
300000	c.a1
300004	c.a2
300008	c.a3
300012	c.b1
300016	c.b2
300020	c.c3
300024	c.c4



Casts

- ▶ Let's consider the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };

void main(void) {
    C* c = new C();
    B* b = &c;
}
```

- ▶ Assuming the pointer “c” points to the offset 300000, what do we need to do to obtain a pointer to an object of type B* ?
- ▶ The simplest way is to add (+12) to the offset where “c” variable points. This will position the new pointer to a location of a B type variable.

Offset	Field/Var
100000	c*
100004	b*
300000	c.a1
300004	c.a2
300008	c.a3
300012	c.b1
300016	c.b2
300020	c.c3
300024	c.c4

+12

B

Casts

```
class A
{
public:
    int a1,a2,a3;
};
```

```
class B:public A
{
public:
    int b1,b2;
};
```

```
void main(void)
{
    B b;
    A* a = &b;
}
```

```
;B b;
lea    ecx,[b]
call   B::B
;A* a = &b;
lea    eax,[b]
mov   dword ptr [a],eax
```

Casts

```
class A
{
public:
    int a1,a2,a3;
};
```

```
class B
{
public:
    int b1,b2;
};
```

```
class C:public A,B
{
public:
    int c1,c2;
};
```

```
void main(void)
{
    C c;
    A* a;
    B* b;

    a = &c;
    b = &c;
}
```

Disasm

```
a = &c;
    lea    eax,[c]
    mov    dword ptr [a],eax

b = &c;
    lea    eax,[c]
    test   eax,eax
    je     NULL_CAST
    lea    ecx,[c]
    add    ecx,0Ch
    mov    dword ptr [ebp-104h],ecx
    jmp    GOOD_CAST

NULL_CAST:
    mov    dword ptr [ebp-104h],0

GOOD_CAST:
    mov    edx,dword ptr [ebp-104h]
    mov    dword ptr [b],edx
```

Casts

```
class A
{
public:
    int a1,a2,a3;
};
```

```
class B
{
public:
    int b1,b2;
};
```

```
class C:public A,B
{
public:
    int c1,c2;
};
```

```
void main(void)
{
    C c;
    A* a;
    B* b;

    a = &c;
    b = &c;
}
```

Disasm

```
a = &c;           eax,[c]
                 mov    dword ptr [a],eax
b = &c;           lea    eax,[c]
                 test   eax,eax
                 je     NULL_CAST
                 lea    ecx,[c]
                 add    ecx,0Ch (0Ch = 12)
                 mov    dword ptr [ebp-104h],ecx
                 jmp    GOOD_CAST
NULL_CAST:      mov    dword ptr [ebp-104h],0
GOOD_CAST:      mov    edx,dword ptr [ebp-104h]
                 mov    dword ptr [b],edx
```

12 =
sizeof(A)

Casts

```
class A
{
public:
    int a1,a2,a3;
};
```

```
class B
{
public:
    int b1,b2;
};
```

```
class C:public A,B
{
public:
    int c1,c2;
};
```

```
void main(void)
{
    C c;
    A* a;
    B* b;
    C* c2;
    a = &c;
    b = &c;
    c2 = (C*)b;
}
```

Disasm

```
c2 = (C*)b;
    cmp    dword ptr [b],0
    je     NULL_CAST
    mov    eax,dword ptr [b]
    sub    eax,0Ch (0Ch = 12)
    mov    dword ptr [ebp-110h],eax
    jmp    GOOD_CAST

NULL_CAST:
    mov    dword ptr [ebp-110h],0

GOOD_CAST:
    mov    ecx,dword ptr [ebp-110h]
    mov    dword ptr [c2],ecx
```

12 =
sizeof(A)

Casts

- ▶ Let's analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : private A, private B { public: int c1, c2; };

void main(void)
{
    C c;
    A* a = &c; ← error C2243: 'type cast': conversion from
} 'C *' to 'A *' exists, but is inaccessible
```

- ▶ This type of cast can be performed only if the inheritance is public.
- ▶ In this example, there is a clear translation from “A*” to “C*”, but, as A is inherit using **private** access modifier, this cast is not allowed.

Upcast / downcast

- ▶ Whenever we discuss about conversion between classes that share an inheritance relationship, there are two notions that need to be mentioned: upcast & downcast
- ▶ Let's assume that we have the following classes:
- ▶ We have the following terminology:

App.cpp

```
void main() {  
    C object;  
    A * base = &object;  
}
```

UPCAST: From child to parent ⇔ convert to base
• Always possible and safe

and

App.cpp

```
void main() {  
    A object;  
    C * child = (C*)(&object);  
}
```

DOWNCAST: From parent to child
• Requires explicit cast or dynamic cast
• Unsafe !!!

App.cpp (C inherits B inherits A)

```
class A { public: int a1, a2, a3; };  
class B : public A { public: int b1, b2; };  
class C : public B { public: int c1, c2; };
```

Upcast / downcast

- ▶ Upcasting produces an interesting secondary effect called *object slicing*

App.cpp

```
struct A {  
    int a1, a2, a3;  
    A(int value) : a1(value), a2(value), a3(value) {}  
};  
struct B : public A {  
    int b1, b2;  
    B(int value): A(value*value), b1(value), b2(value) {}  
};  
void DoSomething(A object) {  
    printf("a1=%d, a2=%d, a3=%d\n", object.a1, object.a2, object.a3);  
}  
void main(void) {  
    B b_object(5);  
    DoSomething(b_object);  
}
```

DoSomething has one parameter of type A
(NOT a reference to an A !!!)

DoSomething is called with a parameter of type B

- ▶ This code will compile and will print on the screen: a1=25,a2=25,a3=25. Since an instance of B can always be casted to its base class, you can *slice* from an object of type B values v1 and v2 and you will obtain an object of type A that can be sent to function *DoSomething*.

Upcast / downcast

- ▶ Upcasting produces an interesting result

App.cpp

```
struct A {  
    int a1, a2, a3;  
    A(int value) : a1(value), a2(value), a3(value) {}  
};  
struct B : public A {  
    int b1, b2;  
    B(int value): A(value*value), b1(value*value) {}  
};  
void DoSomething(A object) {  
    printf("a1=%d, a2=%d, a3=%d\n", object.a1, object.a2, object.a3);  
}  
void main(void) {  
    B b_object(5);  
    DoSomething(b_object);  
}
```

```
B b_object(5);  
push    5  
lea     ecx,[b_object]  
call   B::B (0C71181h)  
DoSomething(b_object);  
mov     eax,dword ptr [b_object]  
mov     dword ptr [ebp-60h],eax  
mov     ecx,dword ptr [ebp-10h]  
mov     dword ptr [ebp-5Ch],ecx  
mov     edx,dword ptr [ebp-0Ch]  
mov     dword ptr [ebp-58h],edx  
sub    esp,0Ch  
mov     eax,esp  
mov     ecx,dword ptr [ebp-60h]  
mov     dword ptr [eax],ecx  
mov     edx,dword ptr [ebp-5Ch]  
mov     dword ptr [eax+4],edx  
mov     ecx,dword ptr [ebp-58h]  
mov     dword ptr [eax+8],ecx  
call   DoSomething (0C712A3h)  
add    esp,0Ch
```

Only the fields
a1,a2 and a3
from b_object
are copied on
the stack

- ▶ This code will compile and work correctly. Since an instance of B can always be converted to an object of type A values v1 and v2 can be passed to a function that expects an object of type A that can be send to function *DoSomething*.

Upcast / downcast

- ▶ Upcasting produces an interesting secondary effect called *object slicing*

App.cpp

```
struct A {  
    int a1, a2, a3;  
    A(int value) : a1(value), a2(value), a3(value) {}  
    A(const A& obj) : A(obj.a1) {}  
};  
struct B : public A {  
    int b1, b2;  
    B(int value): A(value*value), b1(value), b2(0) {}  
};  
void DoSomething(A object) {  
    printf("a1=%d, a2=%d, a3=%d\n", object.a1, object.a2, object.a3);  
}  
void main(void) {  
    B b_object(5);  
    DoSomething(b_object);  
}
```

```
B b_object(5);  
push    5  
lea     ecx,[b_object]  
call   B::B (0C71181h)  
DoSomething(b_object);  
sub    esp,0Ch  
mov    ecx,esp  
lea     eax,[b_object]  
push    eax  
call   A::A (0A4137Fh)  
call   DoSomething (0A412A3)  
add    esp,0Ch
```

Copy-ctor for
class A

- ▶ However, if we add a copy-constructor to class A, it would be used.

Upcast / downcast

- ▶ Upcasting produces an interesting secondary effect called *object slicing*

App.cpp

```
struct A {  
    int a1, a2, a3;  
    A(int value) : a1(value), a2(value), a3(value) {}  
    A(const A& obj) : A(obj.a1) {}  
};  
struct B : public A {  
    int b1, b2;  
    B(int value): A(value*value), b1(value), b2(value) {}  
};  
void main(void) {  
    B b_object(5);  
    A a_object = b_object;  
}
```

```
B b_object(5);  
push      5  
lea       ecx,[b_object]  
call     B::B (0C71181h)  
A a_object = b_object;  
lea       eax,[b_object]  
push      eax  
lea       ecx,[a_object]  
call     A::A (081137Fh)
```

Copy-ctor for
class A

- ▶ However, if we add a copy-constructor to class A, it would be used. This way of creating a object from another object that inherits it, is another usage of *object slicing*.

Casts

- ▶ Let's analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };

void main(void)
{
    C c;
    B* b = &c;
}
```

- ▶ What happens when we convert (cast) &c to B* ?

Casts

- ▶ Let's analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };

void main(void)
{
    C c;
    B* b = &c;
}
```

- ▶ What happens when

le
test
je
lea
add
mov
jmp

NULL CAST:

mov

DONE:

mov

mov

eax,[c]
eax, eax
NULL_CAST
ecx,[c]
ecx,0Ch
dword ptr [ebp-0F8h],ecx
DONE
dword ptr [ebp-0F8h],0
edx,dword ptr [ebp-0F8h]
dword ptr [b],edx

Casts

- ▶ Let's analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };

void main(void)
{
    C c;
    B* b = &c;
}
```

- ▶ But - what if we **DO NOT WANT** to change the address “*b*” points to when the cast is performed ? What if we want “*b*” to point to the exact same address as “*&c*” ?

Casts

- ▶ Let's analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };

void main(void)
{
    C c;
    B* b = (B*)((void *)(&c));
}
```

lea eax,[c]
mov dword ptr [b],eax

- ▶ But - what if we **DO NOT WANT** to change the address “*b*” points to when the cast is performed ? What if we want “*b*” to point to the exact same address as “*&c*” ?
- ▶ One solution is to use a double cast (first cast “*&c*” to a *void**, and then cast that *void** to a *B**)

Casts

- ▶ Let's analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : public A, public B { public: int c1, c2; };

void main(void)
{
    C c;
    B* b = reinterpret_cast<B*> (&c);
}
```

- ▶ But - what if we **DO NOT WANT** to change the address “*b*” points to when the cast is performed ? What if we want “*b*” to point to the exact same address as “*&c*” ?
- ▶ Another solution is to use *reinterpret_cast* keyword from C++ to do this !

Casts

- ▶ C++ has several ways / keywords that can be used to specify how a cast should behave:
 - ▶ *static_cast*
 - ▶ *reinterpret_cast*
 - ▶ *dynamic_cast*
 - ▶ *const_cast*
- ▶ All of them have the following syntax:

Syntax

```
static_cast <type to cast to> (expression)
const_cast <type to cast to> (expression)
dynamic_cast <type to cast to> (expression)
reinterpret_cast <type to cast to> (expression)
```

reinterpret_cast

- ▶ *reinterpret_cast* is the simplest cast mechanism that translates in changing the type of a pointer, while maintaining the same address where it points to.

Syntax

```
reinterpret_cast <tip*> (ptr) ⇔ ((tip*)((void*) ptr))
```

- ▶ This is the fastest cast possible (it guarantees very few assembly instructions that will be used for pointer type translation).
- ▶ However, it has a downside as it allows casts between pointers of incompatible types. The result may obtain a pointer with a data that has weird memory alignment, invalid pointers, etc.

reinterpret_cast

- ▶ Let's analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : private A, private B { public: int c1, c2; };

void main(void)
{
    C c;
    A* a = &c; ← error C2243: 'type cast': conversion from
    'C *' to 'A *' exists, but is inaccessible
}
```

- ▶ This code will not compile. While there is a valid conversion from a pointer of type **C*** to a pointer of type **A*** due to the inheritance, as class **A** is inherit in a private way, the conversion (cast) is not possible.

reinterpret_cast

- ▶ Let's analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };
class C : private A, private B { public: int c1, c2; };

void main(void)
{
    C c;
    A* a = reinterpret_cast<A*> (&c);
}
```

- ▶ This code will compile. It is important to understand the *reinterpret_cast* is not bounded by class type or access modifier type.
- ▶ However, the only reason this code works is that “A” class fields (a1,a2 and a3) are the first fields in an instance of type *C* (if we want to obtain a pointer to B in this manner, we will only obtain a B* pointer that overlaps over the offset of local variable *c*).

reinterpret_cast

- ▶ `reinterpret_cast` can not be used with constant values or with variable of basic type. The following expression will not compile.

C++

```
int x = reinterpret_cast <int>(1000);  
  
int y = 100;  
int x = reinterpret_cast <int>(y);
```

error C2440: 'reinterpret_cast': cannot convert from 'int' to 'int'
note: Conversion is a valid standard conversion, which can be
performed implicitly or by use of static_cast, C-style cast or
function-style cast

- ▶ `reinterpret_cast` can however, copy the value of a pointer to a variable (even if that variable is not large enough to hold the entire pointer value).

C++

```
int x = reinterpret_cast<int>("test");
```

ASM

```
mov dword ptr [x],205858h
```

```
char x = reinterpret_cast<char>("test");
```

```
mov    eax, 205858h  
mov    byte ptr [x],al
```

warning C4302:
'reinterpret_cast':
truncation from 'const
char *' to 'char'

- ▶ In this case 0x205858 is actually the address/offset where the string "test" is located in memory.

reinterpret_cast

- The same logic goes for float data types (float / double). Just like in the case of int, they can not be casted in this way.

Cod C++

```
float x = reinterpret_cast<float>(1.2f);  
double x = reinterpret_cast<double>(1.2);
```

error C2440: 'reinterpret_cast': cannot convert from 'float' to
'float'
note: Conversion is a valid standard conversion, which can be
performed implicitly or by use of static_cast, C-style cast or
function-style cast

- reinterpret_cast can however be used with references to change their values. This includes references of a different type than the actual value.

Cod C++

```
int number = 10;  
reinterpret_cast<int &>(number) = 20;  
  
int number = 10;  
reinterpret_cast<char &>(number) = 20;
```

ASM

```
mov dword ptr [number],20  
  
mov byte ptr [number],20
```

reinterpret_cast

- ▶ `reinterpret_cast` can also be used for direct memory access to the actual code (in other words, it can be used to convert a pointer to a function, to a pointer to a data type).

App.cpp

```
int addition(int x, int y)
{
    return x + y;
}

void main(void)
{
    char* a = reinterpret_cast<char*> (addition);
}
```

- ▶ In this example, “a” points to the actual machine code that was generated for the ***addition*** instruction.
- ▶ This is widely used by obfuscator or pieces of code that modify the behavior of a program during the runtime.

static_cast

- ▶ **static_cast** implies a conversion where:
 - ▶ Values can be truncated upon assignment
 - ▶ In case of inheritance, the address where a pointer points to can be changed
- ▶ It can also be used with initialization lists
- ▶ It **does not** check in any way the type of the object (RTTI field from vfptr pointer)
- ▶ It is in particular useful if we want to identify a function with a specific set of parameters (especially if we have multiple instances of that functions/method and calling it might produce an ambiguity).

static_cast

- ▶ static_cast can be used with constants. In this case, the conversion and assignment are often done in the pre-compiling phase.

C++	ASM
int x = static_cast<int>(1000);	mov dword ptr [x],1000
char x = static_cast<char>(1000);	mov byte ptr [x],232 ($232=1000 \% 256$)
char x = static_cast<char>(3.75);	mov byte ptr [x],3 ($3 = \text{int}(3.75)$)
char x = static_cast<char>"test");	error C2440: 'static_cast': cannot convert from 'const char [5]' to 'char'
const char * x = static_cast<const char *>(9);	error C2440: 'static_cast': cannot convert from 'int' to 'const char *'

static_cast

- ▶ Let's analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B { public: int b1, b2; };

void main(void)
{
    B b;
    A* a = static_cast<A *>(&b); // Error line
}
```

error C2440: 'static_cast': cannot convert from 'B *' to 'A *'
note: Types pointed to are unrelated; conversion requires
reinterpret_cast, C-style cast or function-style cast

- ▶ This code will not compile, as there is NO possible conversion from a pointer of type B to a pointer of type A

static_cast

- ▶ Let's analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B
{
    public:
        int b1, b2;
        operator A* () { return new A(); }
};
void main(void)
{
    B b;
    A* a = static_cast<A *>(&b); ← error C2440: 'static_cast': cannot convert from 'B *' to 'A *'
}
```

error C2440: 'static_cast': cannot convert from 'B *' to 'A *'
note: Types pointed to are unrelated; conversion requires
reinterpret_cast, C-style cast or function-style cast

- ▶ This code will not compile, even if we add a **cast operator**. The reason is the same, we try to convert a **B*** to an **A*** (the cast operator requires an object, not a pointer !!!).

static_cast

- ▶ Let's analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B
{
    public:
        int b1, b2;
        operator A* () { return new A(); }
};
void main(void)
{
    B b;
    A* a = static_cast<A *>(b);
}
```

- ▶ Now the code works and a pointer of type `A*` is obtained by calling the *cast operator overload* from class B.

static_cast

- ▶ Let's analyze the following program:

App.cpp

```
int Add(int x, char y)
{
    return x + y;
}
int Add(char x, int y)
{
    return x + y;
}
void main(void)
{
    int suma = Add(100, 200);
}
```

error C2666: 'Add': 2 overloads have similar conversions
note: could be 'int Add(char,int)'
note: or 'int Add(int,char)'
note: while trying to match the argument list '(int, int)'

- ▶ This code will not compile due to the ambiguity between two functions.

static_cast

- ▶ Let's analyze the following program:

App.cpp

```
int Add(int x, char y)
{
    return x + y;
}
int Add(char x, int y)
{
    return x + y;
}
void main(void)
{
    int suma = static_cast<int(*) (int, char)> (Add)(100, 200);
}
```

- ▶ Now the code compiles. By using the *static_cast* we can tell the compiler which one of the two *Add* function it should use !

`dynamic_cast`

- ▶ `dynamic_cast` works by safely convert a pointer/reference of some type into a pointer/reference of a different type, provided there is an inheritance relationship between those two types.
- ▶ In this case , the RTTI from the vptr pointer is used and the evaluation is done at the runtime. This means that casting using `dynamic_cast` is generally slower than other type of casts. However, if the cast is done, we are certain that we have obtained a pointer to a valid pointer/reference
- ▶ This also implies that RTTI field must be present → `dynamic_cast` works on classes that have at least one virtual method.
- ▶ `dynamic_cast` also works if there is a clear translation/cast between classes

dynamic_cast

- ▶ Let's analyze the following program:

App.cpp

```
class A { public: int a1, a2, a3; };
class B
{
    public:
        int b1, b2;
};
class C : public A, public B { public: int c1, c2; };
void main(void)
{
    C c;
    B *b = dynamic_cast<B*>(&c);
}
```

- ▶ This code will compile - but keep in mind that this is a safe (clear) translation as we will *always* be able to obtain an object of type B from an object of type C (that inherits B).

dynamic_cast

- ▶ Let's analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B
{
    public:
        int b1, b2;
};
class C : public A, public B { public: int c1, c2; };
void main(void)
{
    B b;
    C *c = dynamic_cast<C*>(&b);
```

error C2683: 'dynamic_cast': 'B' is not a polymorphic type
note: see declaration of 'B'

- ▶ This code will not compile. While there is a possible conversion from B^* to C^* , it's not a safe one. The only case such a conversion will be safe, is if that B^* was obtained from a “C” object and then re-casted back to a “ C^* ”. As this can be validated only for classes that have virtual methods (*polymorphic types*), the compiler will issue an error.

dynamic_cast

- ▶ Let's analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B
{
    public:
        int b1, b2;
        virtual void f() {};
};
class C : public A, public B { public: int c1, c2; };
void main(void)
{
    B b;
    C *c = dynamic_cast<C*>(&b);
}
```

- ▶ This code will compile.
- ▶ However, “c” will be *nullptr* (as we can not obtain a valid **C** object from a **B** object). This is very useful as it allows the programmer to check the result of the cast and have different actions based on it.

dynamic_cast

- ▶ Let's analyze the following code:

App.cpp

```
class A { public: int a1, a2, a3; };
class B
{
    public:
        int b1, b2;
        virtual void f() {};
};
class C : public A, public B { public: int c1, c2; };
void main(void)
{
    C c;
    B *b = (B*)&c;
    C *c2 = dynamic_cast<C*>(b);
}
```

- ▶ Now, the code compiles and the value of “c2” point to object “c”.
- ▶ In reality, *b** is a pointer to the B part of object “c” (this means that we can safely convert it to a *C**)

dynamic_cast

- ▶ Let's analyze the following code:

App.cpp

```
class A {
    public: int a1, a2, a3;
            virtual void f2() {};
};

class B {
    public: int b1, b2;
            virtual void f() {};
};

class C : public A, public B { public: int c1, c2; };

void main(void) {
    C c;
    A *a = (A*)&c;
    B *b = dynamic_cast<B*>(a);
}
```

- ▶ This code also compiles.
- ▶ While there is no relationship between *A* and *B* , since local variable “*a*” points to an object of type *C* , that contains a “*B*” component, this conversion is possible.

const_cast

- ▶ **const_cast** is used to change “const” characteristics for an object. It can be used only on data of the same type (it will not convert from a type to another), it will just remove the “const” characteristic of one object.
- ▶ It is also important to mention that this type of cast can produce ***undefined behavior*** depending on the way it is used !!!

const_cast

- ▶ Let's analyze the following code:

App.cpp

```
void main(void)
{
    int x = 100;
    const int * ptr = &x;

    int * non_const_pointer = const_cast <int *>(ptr);

    *non_const_pointer = 200;
    printf("%d", x);
}
```

- ▶ This code will compile and will convert *ptr* const pointer to a non-constant pointer that can be used to change the value of “x”
- ▶ As a result, the program will print “200” on the screen.
- ▶ It is important that the variable/memory zone where “ptr” pointer points to is NOT located on a READ-ONLY memory page (or simple put, it should be a constant pointer obtained from a non-constant variable).

const_cast

- ▶ Let's analyze the following code:

App.cpp

```
void main(void)
{
    const char * txt = "C++ exam";

    char * non_const_pointer = const_cast <char *>(txt);

    non_const_pointer[0] = 'c';
    printf("%s", non_const_pointer);
}
```

- ▶ This code will compile but it will most likely produce a run-time error.
- ▶ In this case, “C++ exam” is located in a section of the executable that is constant (does not have the write permission for the memory page where it is located). As a result, even if we convert the const pointer to a non-const pointer, when we try to modify it, it will crash !

const_cast

- ▶ Let's analyze the following code:

App.cpp

```
void main(void)
{
    const int x = 100;
    x = 200;
}
```

- ▶ This code will not compile, as “x” is defined as *const*, and *const* variables can not be changed !

const_cast

- ▶ Let's analyze the following code:

App.cpp

```
void main(void)
{
    const int x = 100;
    *(const_cast<int*>(&x)) = 200;

}
```

- ▶ Now , this code will compile (using the **const_cast** we can remove the const attribute of “x”) allowing one to change “x” value.
- ▶ The behavior of the program is however , undefined → in this context **undefined** means that the actual value of “x” may not be 200 !!!
- ▶ A runtime error will not happen because “x” is actually located on the stack (this means that it is located on a memory page with the WRITE flag set).

const_cast

- ▶ Let's analyze the following code:

App.cpp

```
void main(void)
{
    const int x = 100;
    *(const_cast<int*>(&x)) = 200;

    printf("%d", x);
}
```

```
const int x = 100;
mov     dword ptr [x],64h
*(const_cast<int*>(&x)) = 200;
mov     dword ptr [x],0C8h
printf("%d", x);
mov     esi,esp
push   64h
push   2058A8h ; "%d"
call   printf
add    esp,8
```

- ▶ Caz 1: Debug mode, no optimizations
 - ▶ Since “x” is defined as constant, the compiler assumes that whenever “x” is used it can replace its value with the constant. This means, that the following instruction : “printf(“%d”, x);” is actually translated by the compiler into: “printf(“%d”, 100);”

const_cast

- ▶ Let's analyze the following code:

App.cpp

```
void main(void)
{
    const int x = 100;
    *(const_cast<int*>(&x)) = 200;

    printf("%d", *(const_cast<int*>(&x)));
}
```

```
const int x = 100;
        dword ptr [x],64h
*(const_cast<int*>(&x)) = 200;
        dword ptr [x],0C8h
printf("%d", x);
        esi,esp
mov    push
push    call
add
```

- ▶ In this case, we force the compiler to use the value of “x” through a **const_cast** → as a result value 200 is printed.
- ▶ However, keep in mind that this is not a normal usage of “x”

const_cast

- ▶ Let's analyze the following code:

App.cpp

```
class Test
{
public:
    const int x;
    Test(int value) : x(value) { }
    void Set(int value) { *(const_cast<int*>(&x)) = value; }
};
void main(void)
{
    Test t(100);
    printf("%d ", t.x);
    t.Set(200);
    printf("%d ", t.x);
}
```

- ▶ This is however, a different case. Even if “x” is a constant, every instance of class Test can have a different value for “x”. This means, that the compiler will not use the value used to construct “x”, but rather the content of memory that corresponds to “x”. As a results, this code will print first 100 and then 200.

A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented diagonally, creating a sense of depth and movement.

► Macros

Macros

- ▶ Macros are methods that can be used to modify (pre-process) the code in C/C++ before compiling
- ▶ They are defined using the following syntax:
`#define <macro> <value>`
- ▶ Once defined a macro can be removed from the definition list using the following syntax:
`#undef macro`
- ▶ Macros work before the compile time in a *preprocessor* phase.
- ▶ One simple way of understanding them is to think that you are within an editor and you are applying a *replace* command (search for a text and replace it with another text).
- ▶ In reality macros are far more complex (in terms of how the replacement is done, and what kind of things can be replaced by them).

Macros

- ▶ Examples (simple macros)

App.cpp

```
#define BUFFER_SIZE      1024  
  
char Buffer[BUFFER_SIZE];
```

- ▶ After the *preprocessor* phase the code will look like this:

App.cpp

```
char Buffer[1024];
```

Macros

- ▶ Macros work sequentially (are applied immediately after they are defined). Also, during the *preprocessor* phase, there is no notion of identifiers, so there can be a variable and a macro that have the same name.

App.cpp

```
void main(void)
{
    int value = 100;
    int temp;
    temp = value;
#define value 200
    temp = value;
}
```

- ▶ After *preprocessor* phase the code will look as follows:

App.cpp

```
void main(void)
{
    int value = 100;
    int temp;
    temp = value;
    temp = 200;
}
```

Macros

- ▶ Macros can be written on more than one line. To do this, the special character ‘\’ is used at the end of each line.
Important → after this character there shouldn’t be any other characters (except EOL).

App.cpp

```
#define PRINT\
    if (value > 100) printf("Greater!"); \
    else printf("Smaller!");

void main(void)
{
    int value = 100;
    PRINT;
}
```

- ▶ After *preprocessor* phase the code will look as follows:

App.cpp

```
void main(void)
{
    int value = 100;
    if (value > 100) printf("Greater!");
    else printf("Smaller!");
}
```

Macros

- ▶ Macros can be defined using another macro. In this case the usage of **#undef** and **#define** can change the value of a macro during *preprocessor* phase .

App.cpp

```
#define BUFFER_SIZE      VALUE
#define VALUE             1024
char Temp[BUFFER_SIZE];

#undef VALUE

#define VALUE            2048
char Temp2[BUFFER_SIZE];
```

- ▶ After *preprocessor* phase the code will look as follows:

App.cpp

```
char Temp[1024];
char Temp2[2048];
```

Macros

- ▶ Macros can be defined to look like a function.

App.cpp

```
#define MAX(x,y) ((x)>(y)?(x) : (y))

void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    int v3 = MAX(v1, v2);
}
```

- ▶ After *preprocessor* phase the code will look as follows:

App.cpp

```
void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    int v3 = ((v1)>(v2) ? (v1) : (v2));
}
```

Macros

- ▶ Macros that are defined to look like a function can have a variable number of parameters if the specifier ‘...’ is used.

App.cpp

```
#define PRINT(format,...) \
{ \
    printf("\nPrint values:"); \
    printf(format, __VA_ARGS__); \
}
void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    PRINT("%d,%d", v1, v2);
}
```

- ▶ After *preprocessor* phase the code will look as follows:

App.cpp

```
void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    printf("\nPrint values:");
    printf("%d,%d", v1, v2);
}
```

Macros

- ▶ Macros can use the special character ‘#’ to change a parameter into its corresponding string:

App.cpp

```
#define CHECK(condition) { \
    if (!(condition)) { printf("The condition '%s' wasn't evaluated correctly", #condition); \
} \
void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    CHECK(v1 > v2);
}
```

- ▶ After *preprocessor* phase the code will look as follows:

App.cpp

```
void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    if (!(v1>v2)) { printf("The condition '%s' wasn't evaluated correctly", "v1 > v2"); }
}
```

Macros

- ▶ Macros can use the special character '#' to change a parameter into its corresponding string:

App.cpp

```
#define CHECK(condition) \
    if (!(condition)) { printf("The condition '%s' wasn't evaluated correctly", #condition); \
}

void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    CHECK(v1 > v2);
}
```

- ▶ After *preprocessor* phase the code will look as follows:

App.cpp

```
void main(void)
{
    int v1, v2;
    v1 = 100;
    v2 = 200;
    if (!(v1>v2)) { printf("The condition '%s' wasn't evaluated correctly", "v1 > v2", ); }
}
```

Macros

- ▶ Macros can use this sequence of characters “##” to concatenate parameters:

App.cpp

```
#define SUM(type) \
    type add_##type(type v1, type v2) { return v1 + v2; }

SUM(int);
SUM(double);
SUM(char);
void main(void)
{
    int x = add_int(10, 20);
```

- ▶ After *preprocessor* phase the code will look as follows:

App.cpp

```
int add_int(int v1, int v2) { return v1 + v2; }
double add_double(double v1, double v2) { return v1 + v2; }
char add_char(char v1, char v2) { return v1 + v2; }

void main(void)
{
    int x = add_int(10, 20);
```

Macros

- ▶ Macros don't support overloading (if there are two macros with the same name, the last one will replace the other, a warning will be issued in the compiler and then the code will be compiled).
- ▶ The following code won't compile because SUM needs 3 parameters (only the second macros is used)

App.cpp

```
#define SUM(a,b) a+b
#define SUM(a,b,c) a+b+c
void main(void)
{
    int x = SUM(1, 2);
```

warning C4005: 'SUM': macro redefinition
note: see previous definition of 'SUM'

warning C4003: not enough arguments for
function-like macro invocation 'SUM'
error C2059: syntax error: ';' ←

- ▶ The following code is compiled and works properly

App.cpp

```
void main(void)
{
    #define SUM(a,b) a+b
    int x = SUM(1, 2);
    #define SUM(a,b,c) a+b+c
    x = SUM(1, 2, 3);
}
```

Macros

- ▶ Use round brackets! Macros don't analyse the expression; they just do a text replace → the results may be different than expected.

Incorrect

```
#define DIV(x,y) x/y
void main(void)
{
    int x = DIV(10 + 10, 5 + 5);
}
```

Correct

```
#define DIV(x,y) ((x)/(y))
void main(void)
{
    int x = DIV(10 + 10, 5 + 5);
}
```

- ▶ After precompilation the code will look as follows:

Incorrect

```
void main(void)
{
    int x = 10 + 10 / 5 + 5;
}
```

Correct

```
void main(void)
{
    int x = ((10 + 10) / (5 + 5));
}
```

Macros

- ▶ Be careful when they are used in a loop. Whenever possible use '{' and '}' to make a complex instruction easier to understand!

Incorrect

```
#define PRINT(x,y) \
    printf("X=%d",x);printf("Y=%d",y);
void main(void)
{
    int x = 10, y = 20;
    if (x > y)
        PRINT (x, y);
}
```

Correct

```
#define PRINT(x,y) \
    { printf("X=%d",x);printf("Y=%d",y); }
void main(void)
{
    int x = 10, y = 20;
    if (x > y)
        PRINT (x, y);
}
```

- ▶ After *preprocessor* phase the code will look as follows:

Incorrect

```
void main(void)
{
    int x = 10, y = 20;
    if (x > y)
        printf("X=%d",x);printf("Y=%d",y);
}
```

Correct

```
void main(void)
{
    int x = 10, y = 20;
    if (x > y)
        {printf("X=%d",x);printf("Y=%d",y);}
}
```

Macros

- ▶ Pay attention to operators and functions that modify the parameter of the macro. Because of the substitution some calls will be made more than one time.

Incorrect (res will be 3)

```
#define set_min(result,x,y) \
    result = ((x)>(y)?(x):(y)); \
void main(void)
{
    int x = 2, y = 1;
    int res;
    set_min(res, x++, y);
}
```

Correct

```
#define set_min(result,x,y) { \
    int t_1 = (x); \
    int t_2 = (y); \
    result = ((t_1)>(t_2)?(t_1):(t_2)); \
}
void main(void)
{
    int x = 2, y = 1;
    int res;
    set_min(res, x++, y);
}
```

- ▶ After *preprocessor* phase the code will look as follows:

Incorrect

```
void main(void)
{
    int x = 2, y = 1;
    int res;
    result = ((x++)>(y)?(x++):(y));
}
```

Correct

```
void main(void)
{
    int x = 2, y = 1;
    int res;
    { int t_1 = (x); int t_2 = (y);
    result = ((t_1)>(t_2)?(t_1):(t_2)); }
}
```

Macros

- ▶ There is a list of macros that are predefined for any C/C++ compiler:

Macro	Value
<code>__FILE__</code>	The current file in which the macro will be substituted
<code>__LINE__</code>	The line in the current file in which the macro will be substituted
<code>__DATE__</code>	The date when the code was compiled
<code>__TIME__</code>	The time when the code was compiled
<code>__STDC_VERSION__</code>	Compiler version(Cx98, Cx03,Cx13,...)
<code>__cplusplus</code>	It is defined if a C++ compiler is used
<code>__COUNTER__</code>	A unique identifier (0..n) used for indexing

- ▶ Besides these, each compiler defines it's own specific macros(for compiler version, linkage options, etc)

Macros

- ▶ The usage of `__COUNTER__` is very helpful if we want to create unique indexes in our program:

App.cpp

```
void main(void)
{
    int x = __COUNTER__;
    int y = __COUNTER__;
    int z = __COUNTER__;
    int t = __COUNTER__;
}
```

- ▶ After precompilation the code will look as follows:

App.cpp

```
void main(void)
{
    int x = 0;
    int y = 1;
    int z = 2;
    int t = 3;
}
```

A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented at various angles, creating a sense of depth and motion.

- ▶ Macros vs Inline

Macro vs Inline

- ▶ Let's analyze the following code:

App.cpp

```
int Max(int x, int y)
{
    return x > y ? x : y;
}
int main()
{
    int x, y;
    x = rand();
    y = rand();
    printf("%d", Max(x, y));
    return 0;
}
```

- ▶ OBSERVATION: This code is compiled without any compiler optimizations !

Macro vs Inline

- ▶ Let's analyze the following code :

App.cpp

```
int Max(int x, int y)
{
    return x > y ? x : y;
}
int main()
{
    int x, y;
    x = rand();
    y = rand();
    printf("%d", Max(x, y));
    return 0;
}
```



```
mov eax,dword ptr [y]
push eax
mov ecx,dword ptr [x]
push ecx
call Max
add esp,8
push eax
push offset string "%d"
call printf
```

- ▶ In this case , **Max** function will be called, with “x” and “y” being pushed on the stack.

Macro vs Inline

- ▶ So ... how can we optimize this program:

App.cpp

```
int Max(int x, int y)
{
    return x > y ? x : y;
}
int main()
{
    int x, y;
    x = rand();
    y = rand();
    printf("%d", Max(x, y));
    return 0;
}
```

Macro vs Inline

- ▶ So ... how can we optimize this program :

App.cpp

```
int Max(int x, int y)
{
    return x > y ? x : y;
}
int main()
{
    int x, y;
    x = rand();
    y = rand();
    printf("%d", Max(x, y));
    return 0;
}
```

App.cpp (with macros)

```
#define Max(x,y) x > y ? x : y

int main()
{
    int x, y;
    x = rand();
    y = rand();
    printf("%d", Max (x, y));
    return 0;
}
```

- ▶ If we replace **Max** function with **Max** macro, then the replacement is done in the pre-execution phase. As a result, the piece of code that pushes variable on the stack and **Max** function assembly stubs are no longer required.

Macro vs Inline

- ▶ So ... how can we optimize this program :

App.cpp

```
int Max(int x, int y)
{
    return x > y ? x : y;
}
int main()
{
    int x, y;
    x = rand();
    y = rand();
    printf("%d", Max(x, y));
    return 0;
}
```

App.cpp

```
#define Max(x, y) (x > y ? x : y)

int main()
{
    int x, y;
    x = rand();
    y = rand();
    printf("%d", Max (x, y));
    return 0;
}
```

```
mov     eax,dword ptr [x]
cmp     eax,dword ptr [y]
jle     X_IS_SMALLER
mov     mov
jmp     X_IS_SMALLER:
        mov     edx,dword ptr [y]
        mov     dword ptr [ebp-4Ch],edx
END_IF:
        mov     eax,dword ptr [ebp-4Ch]
        eax
        offset string "%d"
printf
```

- ▶ In this case, the replacement modifies the code from printf

Macro vs Inline

- ▶ So ... how can we optimize this program :

App.cpp

```
int Max(int x, int y)
{
    return x > y ? x : y;
}
int main()
{
    int x, y;
    x = rand();
    y = rand();
    printf("%d", Max(x, y));
    return 0;
}
```

App.cpp (inline)

```
inline int Max(int x, int y)
{
    return x > y ? x : y;
}
int main()
{
    int x, y;
    x = rand();
    y = rand();
    printf("%d", Max (x, y));
    return 0;
}
```

- ▶ Another solution is to use *inline* specifier. This specifier tells the compiler that we recommend to insert the code of *Max* function directly in the code of caller (e.g. in this case *main* function).

Macro vs Inline

- ▶ So ... how can we optimize this program :

App.cpp

```
int Max(int x, int y)
{
    return x > y ? x : y;
}
int main()
{
    int x, y;
    x = rand();
    y = rand();
    printf("%d", Max(x, y));
    return 0;
}
```

App.cpp (inline)

```
inline int Max(int x, int y)
{
    return x > y ? x : y;
}
int main()
{
    int x, y;
    x = rand();
    y = rand();
    printf("%d", Max (x, y));
    return 0;
}
```

Debug mode
(no optimizations)

```
mov    eax, dword ptr [y]
push   eax
mov    ecx, dword ptr [x]
push   ecx
Max
call   esp,8
add    eax
push   offset string "%d"
printf
```

- ▶ Another solution is to use *inline* specifier. This specifier tells the compiler that we recommend to insert the code of **Max** function directly in the code of caller (e.g. in this case **main** function).

Macro vs Inline

Release mode
(with optimizations)

```
cmp      edi,eax // edi = x
cmove
push
push
call
      eax
      offset string "%d"
printf
      return x > y ? x
}
int main()
{
    int x, y;
    x = rand();
```

cmove = Compare and Move if Greater

program :

App.cpp (inline)

```
inline int Max(int x, int y)
{
    return x > y ? x : y;
}
int main()
{
    int x, y;
    x = rand();
    y = rand();
printf("%d", Max (x, y));
    return 0;
}
```

Debug mode
(no optimizations)

```
mov
push
mov
push
call
add
push
push
call
      eax,dword ptr [y]
      eax
      ecx,dword ptr [x]
      ecx
Max
      esp,8
      eax
      offset string "%d"
printf
```

- ▶ Another solution is to use *inline* specifier. This specifier tells the compiler that we recommend to insert the code of **Max** function directly in the code of caller (e.g. in this case **main** function).
- ▶ If we use *optimizations* “x” and “y” are used as registers and the whole process is optimized.

Macro vs Inline

- ▶ Not all functions can be used with *inline* specifier

App.cpp

```
inline int Iterativ(int x,int limit)
{
    int sum = 0;
    for (; limit <= x; limit++, sum += limit);
    return sum;
}

int main()
{
    int x;
    x = rand();
    printf("%d", Iterativ(x,x-5));
    return 0;
}
```

- ▶ If this code is compiled using optimization, the compiler will integrate **Iterativ** function in main (mainly because **Iterativ** function does not contain any recursive functionalities - in this case).

Macro vs Inline

- ▶ Not all functions can be used with *inline* specifier

App.cpp

```
inline int Recursiv(int x,int limit)
{
    if (x == limit)
        return limit;
    else
        return x + Recursiv(x-1,limit);}

int main()
{
    int x;
    x = rand();
    printf("%d", Recursiv(x,x-5));
    return 0;
}
```

- ▶ If we run this code with *optimization* the compiler is not able to *inline* the code from **Recursiv** function:

- ▶ /permissive- /GS /GL /analyze- /W3 /Gy /Zc:wchar_t /Zi /Gm- /O2 /sdl /Fd"Release\vc141.pdb" /Zc:inline /fp:precise /D "WIN32" /D "NDEBUG" /D "_CONSOLE" /D "_UNICODE" /D "UNICODE" /errorReport:prompt /WX- /Zc:forScope /Gd /Oy- /Oi /MD /FC /Fa"Release\" /EHsc /nologo /Fo"Release\" /Fp"Release\TestCpp.pch" /diagnostics:classic
- ▶ cl.exe → 19.16.27030.1

Macro vs Inline

- ▶ Using MACROS guarantees inline replacement. On the other hand, debugging is really tricky, especially if the MACRO is complex in nature (multiple lines).
- ▶ Using **inline** specifier DOES NOT GUARANTEES that inline replacement is going to happen (it should be seen more as a suggestion for the compiler). On the other hand, debugging is easier and without any bit issues.
- ▶ In case of *inline* , this is used on in release mode (or to be more exact, if the optimizations are enabled (-O1, -O2, ...) at the compiler level.

Macro vs Inline

- ▶ Class methods (if defined in the definition of the class) are usually considered inline as well.

App.cpp

```
class Test
{
    int x; // private
public:
    void SetX(int value)
    {
        x = value;
    }
    int GetX() { return x; }
};
```

- ▶ There is one exception - when the class is exported (in this case the code from the header file is usually compiled and linked in the component that imports the library). In many cases, the optimization translates into direct access the data member.

Volatile specifier

- ▶ Let's analyze the following code:

App.cpp

```
int main()
{
    int x = rand();
    printf("%d", x);
    return 0;
}
```

call
push
push
call
rand
eax
offset string "%d"
printf

- ▶ If we compile the code in the **release** mode (or more exactly, if optimizations are in place) the compiler might decide that local variable “x” is not necessary (e.g. we can translate the entire code into “**printf(“%d”,rand())**”)
- ▶ The solution - if we want to avoid this behavior is to use **volatile** specifier.

App.cpp

```
int main()
{
    volatile int x = rand();
    printf("%d", x);
    return 0;
}
```

call
mov
rand
x, eax

mov
push
push
call
eax, x
eax
offset string "%d"
printf

The background features a dark blue gradient with a subtle geometric pattern of lighter blue triangles and lines.

► Literals

Literals

- ▶ Literals are a way of providing a special meaning to numbers by preceding them with a predefined string.
- ▶ Literals can be defined in the following way:

```
<return type> operator"_<literal_name> ( parameter_type )
```

- ▶ Where:
 - <return-type> can be any kind of type
 - <literal_name> has to be a name (form out of characters, underline, ...)
 - parameter_type has to be one of the following: const char *, unsigned long long int, long double, char, wchar_t, char8_t, char16_t, char32_t)
 - The underline is not required. However, compilers issue an warning if not used !

Literals

- ▶ Let's analyze the following code:

App.cpp

```
int operator"" _baza_3(const char * x)
{
    int value = 0;
    while ((*x) != 0)
    {
        value = value * 3 + (*x) - '0';
        x++;
    }
    return value;
}
void main(void)
{
    int x = 121102_baza_3;
}
```

- ▶ What we did in this case was to explain the compiler how to interpret if a number is follower by `_baza_3`

Literals

- ▶ Let's analyze the following code:

App.cpp

```
int operator"" _baza_3(const char * x)
{
    int value = 0;
    while ((*x) != 0)
    {
        value = value * 3 + (*x) - '0';
        x++;
    }
    return value;
}
void main(void)
{
    int x = 121102_baza_3;
}
```

push
call
add
mov

offset string "121102"
operator "" _baza_3
esp,4
dword ptr [x],eax

- ▶ Since we define the literal `_baza_3` as one that works with strings, the compiler will translate 121102 (a number) into “121102” (a string) that can be send to the literal conversion function.

Literals

- ▶ Let's analyze the following code:

App.cpp

```
int operator"" Mega(unsigned long long int x)
{
    return ((int)x)*1024*1024;
}
void main(void)
{
    int x = 3Mega;
}
```

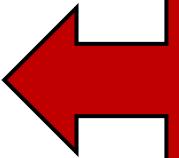
- ▶ This code will compile and run just like the previous one.
- ▶ However, the compiler will throw an warning: “warning C4455: ‘operator ““Mega”: literal suffix identifiers that do not start with an underscore are reserved’”

Literals

- ▶ Let's analyze the following code:

App.cpp

```
int operator"" _Mega(int x)
{
    return ((int)x)*1024*1024;
}
void main(void)
{
    int x = 3_Mega;
}
```



```
error C3684: 'operator ""_Mega': declaration of literal
operator has an invalid parameter list
error C3688: invalid literal suffix '_Mega'; literal operator
or literal operator template 'operator ""_Mega' not found
note: Literal operator must have a parameter list of the form
'unsigned long long' or 'const char *'
```

- ▶ This code will not compile, as we did not use a valid parameter for the defined literal.

Literals

- ▶ Let's analyze the following code:

App.cpp

```
int operator"" _to_int(unsigned long long int x)
{
    return (int)x;
}

void main(void)
{
    unsigned long long int a;
    int sum = a_to_int; ← error C2065: 'a_to_int': undeclared identifier
}
```

- ▶ This code will not compile, as the compiler will consider *a_to_int* a new identifier that was not defined yet.
- ▶ This translates, that we can use this with constant numbers or constant string (but not with variables).

Literals

- ▶ Let's analyze the following code:

App.cpp

```
void operator"" _print_dec(unsigned long long int x)
{
    printf("Decimal: %d\n", (int)x);
}
void operator"" _print_hex(unsigned long long int x)
{
    printf("Hex: 0x%X\n", (unsigned int)x);
}
void main(void)
{
    150_print_dec;
    150_print_hex;
}
```

- ▶ A literal does not have to return a value. In this case we used literals to print the value of a number (as a decimal value and as a hexadecimal value).
- ▶ This code will print 150 and 0x96 on the screen.

Literals

- ▶ Let's analyze the following code:

App.cpp

```
char* operator"" _reverse(const char *sir, size_t sz)
{
    char * txt = new char[sz + 1];
    txt[sz] = 0;
    for (size_t poz = 0; poz < sz; poz++)
        txt[sz - poz - 1] = sir[poz];
    return txt;
}

void main(void)
{
    char * text = "c++ test today"_reverse;
    printf("%s\n", text);
    delete[]text;
}
```

push
push
call
add
mov

0Eh ⇔ the length of “c++ test today”
offset string “c++ test today”
operator “” _reverse
esp,8
dword ptr [text],eax

- ▶ In particular for literals that have a `const char *` parameter (a pointer), the size of the data the pointer points to can also be send if a secondary parameter is added. The secondary parameter **must be a `size_t`.**

A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented diagonally, creating a sense of depth and movement.

- ▶ Templates

Templates

- ▶ Macros bring great power to the code written in C/C++
- ▶ Templates can be considered to be derived from the notion of macros → adapted for functions and classes
- ▶ The goal is to define a model of a function or class in which the types of data we work with can be modified at the precompilation stage (similar to macros)
- ▶ For this there is a key word “***template***”
- ▶ Just like macros, the usage of templates generates extra code at compilation (code that is specific for the data types for which the templates are made for). However, the code is faster and efficient.

Templates

- ▶ The templates defined in C++ are of two types:
 - ❖ For classes
 - ❖ For functions
- ▶ Templates work just like a macro - through substitution
- ▶ The difference is that the template substitution for classes isn't done where the first instance of that class is used, but separately, so that other instances that use the same type of data to be able to use the same substituted template
- ▶ **Important:** Because the substitution is made during precompilation, the templates must be stored in the files that are exported from a library (files.h) otherwise the compiler(in case it is tried the creation of a class from a template exported from a library) will not be able to do this.

Templates

- ▶ Templates for a function are defined as follows:

App.cpp

```
template <class T>
Return_type function_name(parameters)

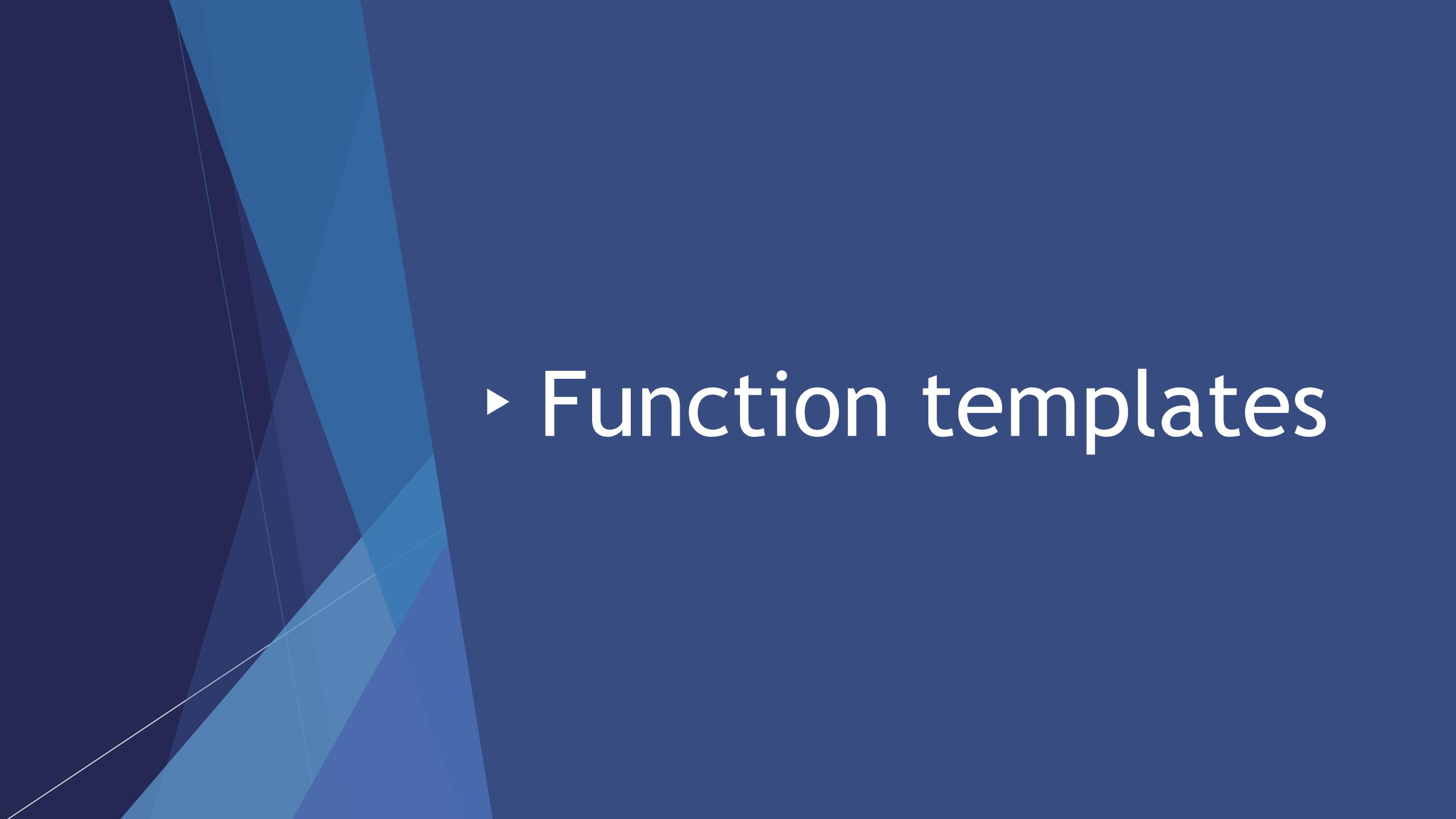
or

template <typename T>
Return_type function_name(parameters)
```

- ▶ Atleast one of the “Return_type” or “parameters” must contain a member of type T.

App.cpp

```
template <class T>
T Sum(T value_1,T value_2)
{
    return value_1 + value_2;
}
```

The background features a large, abstract graphic on the left side composed of various shades of blue and dark blue triangles and trapezoids, creating a layered, geometric pattern.

► Function templates

Templates

- ▶ A simple example:

App.cpp

```
template <class T>
T Sum(T value_1, T value_2)
{
    return value_1 + value_2;
}

void main(void)
{
    double x = Sum(1.25, 2.5);
}
```

- ▶ The code compiles correctly → x receives the value 3.75

Templates

- ▶ A simple example:

App.cpp

```
template <class T>
T Sum(T value_1, T value_2)
{
    return value_1 + value_2;
}

void main(void)
{
    double x = Sum(1, 2.5);
}
```

error C2672: 'Sum': no matching overloaded function found
error C2782: 'T Sum(T,T)': template parameter 'T' is ambiguous
note: see declaration of 'Sum'
note: could be 'double'
note: or 'int'
error C2784: 'T Sum(T,T)': could not deduce template argument for
'T' from 'double'

- ▶ The code will not compile - 1 is converted to int, 2.5 to double, but the Sum function should receive two parameters of the same type.
- ▶ The compiler can't decide which type to use → ambiguous code

Templates

- ▶ A simple example:

App.cpp

```
template <class T>
T Sum(T value_1, T value_2)
{
    return value_1 + value_2;
}

void main(void)
{
    int x = Sum(1, 2);
    double d = Sum(1.5, 2.4);
}
```

- ▶ In this case → x receives the value 3 and the called function will substitute class T with int, and d receives 3.9 (in this case the substitution will be made with double).
- ▶ In the compiled code there will be two Sum functions(one with parameters of type *int* and the other one with parameters of type *double*)

Templates

- ▶ Be careful how you put the parameters!!!

App.cpp

```
template <class T>
T Sum(int x, int y)
{
    return (T)(x + y);
}

void main(void)
{
    int x = Sum(1, 2);
    double d = Sum(1.5, 2.4);
}
```

error C2672: 'Sum': no matching overloaded function found
error C2783: 'T Sum(int,int)': could not deduce template argument for 'T'
note: see declaration of 'Sum'
error C2672: 'Sum': no matching overloaded function found
error C2783: 'T Sum(int,int)': could not deduce template argument for 'T'
note: see declaration of 'Sum'

- ▶ The code above can't be compiled → not because the return value is different, but because the compiler can't deduce what type to use for class T (what return value is obtained)

Templates

- ▶ Be careful how you put the parameters!!!

App.cpp

```
template <class T>
T Sum(int x, int y)
{
    return (T)(x + y);
}

void main(void)
{
    int x = Sum<int>(1, 2);
    double d = Sum<double>(1.5, 2.4);
}
```

- ▶ In this case the code can be compiled (we use <> to specify the type we want to use in the template).
- ▶ X will be 3 and d will also be 3 (1.5 and 2.4 are converted to int)

Templates

- ▶ Templates can be made for more types

App.cpp

```
template <class T1, class T2, class T3>
T1 Sum(T2 x, T3 y)
{
    return (T1)(x + y);
}

void main(void)
{
    int x = Sum<int>(1, 2);
    double d = Sum<int,double,double>(1.5, 2.4);
}
```

- ▶ In the first case (`Sum<int>`) the compiler converts to `Sum<int,int,int>` (which will match the parameters)

Templates

- ▶ Templates can be made for more types

App.cpp

```
template <class T1, class T2, class T3>
T1 Sum(T2 x, T3 y)
{
    return (T1)(x + y);
}

void main(void)
{
    int x = Sum<int,char>(1, 10.5);
    double d = Sum<int,double,double>(1.5, 2.4);
}
```

- ▶ In the first case (`Sum<int>`) the compiler converts to `Sum<int,int,int>` (which will match the parameters)
- ▶ The compiler tries to deduce the type based on the parameters, if it is not specified. The return type must be specified because the compiler can't deduce it from the parameters.

Templates

- ▶ Function templates can receive parameters with default values

App.cpp

```
template <class T1, class T2, class T3>
T1 Sum(T2 x, T3 y = T3(5))
{
    return (T1)(x + y);
}

void main(void)
{
    int x = Sum<int, char, int>(10);
```

- ▶ “x” receives the value 15 ($10+5 \rightarrow$ the default value for y).
- ▶ The use of default parameters requires the existence of a constructor for the type of the used class. Expressions like “ $T3 y = 10$ ” are not valid unless type T3 accepts equality (or has an explicit assignment operator) with int.

A decorative graphic in the top-left corner consists of several overlapping blue triangles of varying shades, creating a sense of depth and perspective.

- ▶ Class templates

Templates

- ▶ Class templates are defined as follows:

App.cpp

```
template <class T>
class MyClass {
...
};

or
template <typename T>
class MyClass {
...
};
```

- ▶ The *T variable* can be used to define class members, parameters within methods and local variables within methods.

Templates

- ▶ Class templates are defined as follows:

App.cpp

```
template <class T>
class Stack
{
    T List[100];
    int count;
public:
    Stack() : count(0) {}
    void Push(T value) { List[count++] = value; }
    T Pop() { return List[--count]; }
};
void main(void)
{
    Stack<int> s;
    s.Push(1); s.Push(2); s.Push(3);
    printf("%d", s.Pop());
}
```

- ▶ The code above prints “3”

Templates

- ▶ Be careful what types you use for functions when using templates in a class:

App.cpp

```
template <class T>
class Stack
{
    T List[100];
    int count;
public:
    Stack () : count(0) {}
    void Push(T &value) { List[count++] = value; }
    T& Pop() { return List[--count]; }
};
void main(void)
{
    Stack<int> s;
    s.Push(1); s.Push(2); s.Push(3);
    printf("%d", s.Pop());
}
```

error C2664: 'void Stack<int>::Push(T &)':
cannot convert argument 1 from 'int' to 'T &'
with
[
 T=int
]

- ▶ The code above can't be compiled. The Pop function is correct, but the Push function must receive a reference → **s.Push(1) doesn't give it a reference!!!**

Templates

- ▶ Be careful what types you use for functions when using templates in a class:

App.cpp

```
template <class T>
class Stack
{
    T List[100];
    int count;
public:
    Stack() : count(0) {}
    void Push(T &value) { List[count++] = value; }
    T& Pop() { return List[--count]; }
};
void main(void)
{
    Stack<int> s;
    for (int tr = 1; tr <= 3;tr++)
        s.Push(tr);
    printf("%d", s.Pop());
}
```

- ▶ The code can be compiled and prints “3”.

Templates

- ▶ The same as function templates, class templates can use multiple types

App.cpp

```
template <class T1, class T2>
class Pair
{
    T1 Key;
    T2 Value;
public:
    Pair () : Key(T1()), Value(T2()) {}
    void SetKey(const T1 &v) { Key = v; }
    void SetValue(const T2 &v) { Value = v; }
};
void main(void) {
    Pair<const char*, int> p;
    p.SetKey("exam grade");
    p.SetValue(10);
}
```

- ▶ “Pair() : Key(T1()), Value(T2())” → calls the default constructor for Key and Value (for “p” Key will be *nullptr* and Value will be 0).
- ▶ It is important that both T1 and T2 have a default constructor: T1() implies creating an object using its default constructor.

Templates

- ▶ The same as function templates, class templates can use more types

App.cpp

```
template <class T1, class T2>
class Pair
{
    T1 Key;
    T2 Value;
public:
    Pair() : Key(T1()), Value(T2()) {}
    Pair(const T1 &v1, const T2& v2) : Key(v1), Value(v2) {}
    void SetKey(const T1 &v) { Key = v; }
    void SetValue(const T2 &v) { Value = v; }
};
void main(void)
{
    Pair<const char*, int> p("exam_grade", 10);
}
```

- ▶ An explicit constructor can also be used in a class that uses a template. In the example above the constructor receives a string (const char *) and a number.

Templates

- ▶ A more complex example:

App.cpp

```
template <class T>
class Stack
{
...
    void Push(T* value) { List[count++] = (*value); }
    T& Pop() { return List[--count]; }
};

template <class T1,class T2>
class Pair
{
...
};

void main(void)
{
    Stack<Pair<const char*, int>> s;

    s.Push(new Pair<const char*, int>("asm_grade", 10));
    s.Push(new Pair<const char*, int>("oop_grade", 9));
    s.Push(new Pair<const char*, int>("c#_grade", 8));
}
```

Templates

- ▶ Macros can be used together with templates:

App.cpp

```
#define P(k,v) new Pair<const char*, int>(k, v)
#define Stack MyStack<Pair<const char*, int>>
```

```
template <class T>
class MyStack
{
...
};

template <class T1, class T2>
class Pair
{
...
};

void main(void)
{
    Stack s;
    s.Push(P("asm_grade", 10));
    s.Push(P("oop_grade", 9));
    s.Push(P("c#_grade", 8));
}
```

Templates

- ▶ Class templates can also accept parameters without a type(constant)

App.cpp

```
template <class T,int Size>
class Stack
{
    T List[Size];
    int count;
public:
    Stack() : count(0) {}
    void Push(const T& value) { List[count++] = (value); }
    T& Pop() { return List[--count]; }
};
void main(void)
{
    Stack<int, 10> s;
    Stack<int, 100> s2;
    for (int tr = 0; tr < 5; tr++)
        s.Push(tr);
}
```

- ▶ In this scenario, “s” will have 10 elements and “s2” 100 elements
- ▶ Two different classes will be created(one with 10 and one with 100 elements)

Templates

- ▶ Class templates can also accept parameters without a type(constant)

App.cpp

```
template <class T,int Size = 100>
class Stack
{
    T List[Size];
    int count;
public:
    Stack() : count(0) {}
    void Push(const T& value) { List[count++] = (value); }
    T& Pop() { return List[--count]; }
};
void main(void)
{
    Stack<int, 10> s;
    Stack<int> s2;
    for (int tr = 0; tr < 5; tr++)
        s.Push(tr);
}
```

- ▶ In this scenario, “s” will have 10 elements and “s2” 100 elements. In the case of s2 the default value for Size will be used.

Templates

- ▶ Class templates can accept default values for types

App.cpp

```
template <class T = int>
class Stack
{
    T List[100];
    int count;
public:
    Stack() : count(0) {}
    void Push(const T& value) { List[count++] = (value); }
    T& Pop() { return List[--count]; }
};
void main(void)
{
    Stack<double> s;
    Stack<> s2;
}
```

- ▶ In this scenario “s” will be a list of 100 doubles and s2 (because we didn’t specify the type) will be a list of ints.

Templates

- ▶ Class templates can accept default values for types

App.cpp

```
template <class T = int>
class Stack
{
    T List[100];
    int count;
public:
    Stack() : count(0) {}
    void Push(const T& value) { List[count++] = (value); }
    T& Pop() { return List[--count]; }
};
void main(void)
{
    Stack<double> s;
    Stack s2;
```

error C2955: 'Stack': use of class template
requires template argument list
note: see declaration of 'Stack'
error C2133: 's2': unknown size
error C2512: 'Stack': no appropriate default
constructor available
note: see declaration of 'Stack'

- ▶ The above code can't be compiled - even if the type T is by default int
- ▶ It must be specified that when a variable of type Stack is made, that variable ("s2") is in fact a template(**we must use <>**)

Templates

- ▶ Classes can use methods with templates

App.cpp

```
class Integer
{
    int value;
public:
    Integer() : value(0) {}
    template <class T>
    void SetValue(T v) { value = (int)v; }
};
void main(void)
{
    Integer i;
    i.SetValue<float>(0.5f);
    i.SetValue<double>(1.2);
    i.SetValue<char>('a');
}
```

- ▶ In the scenario above, Integer has 3 methods (with float, double and char parameter)

Templates

- ▶ Classes can use methods with templates

App.cpp

```
class Integer
{
    int value;
public:
    Integer() : value(0) {}
    template <class T>
    void SetValue(T v) { value = (int)v; }
};
void main(void) {
    Integer i;
    i.SetValue(0.5f);
    i.SetValue('a');
}
```

- ▶ Specifying the type in the template is not mandatory.
- ▶ In this scenario, Integer has two method:
 - ▶ one with a *float* parameter → deduced by the compiler from *0.5f*
 - ▶ one with a *char* parameter → deduced by the compiler from ‘*a*’

Templates

- ▶ Classes that use templates can also have static members:

App.cpp

```
template<class T>
class Number
{
    T Value;
public:
    static int Count;
};

int Number<int>::Count = 10;
int Number<char>::Count = 20;
int Number<double>::Count = 30;

void main(void)
{
    Number<int> n1;
    Number<char> n2;
    Number<double> n3;

    printf("%d,%d,%d", n1.Count, n2.Count, n3.Count);
}
```

Templates

- ▶ Classes that use templates can also have static members:

App.cpp

```
template<class T>
class Number
{
    T Value;
public:
    static T Count;
};

int Number<int>::Count = 10;
char Number<char>::Count = 'a';
double Number<double>::Count = 30;

void main(void)
{
    Number<int> n1;
    Number<char> n2;
    Number<double> n3;

    printf("%d,%c,%lf", n1.Count, n2.Count, n3.Count);
}
```

Templates

- ▶ Classes that use templates can also have friend functions:

App.cpp

```
template<class T>
class Number
{
    T Value;
    int IntValue;
public:
    friend void Test(Number<T> &t);
};
void Test(Number<double> &t)
{
    t.Value = 1.23;
}
void Test(Number<char> &t)
{
    t.Value = 0;
}
void main(void)
{
    Number<char> n1;
    Number<double> n2;
    Test(n1);
    Test(n2);
}
```

Templates

- ▶ Classes that use templates can also have friend functions:

App.cpp

```
template<class T>
class Number
{
    T Value;
    int IntValue;
public:
    friend void Test(Number<T> &t)
    {
        t.Value = 5;
    }
};

void main(void)
{
    Number<char> n1;
    Number<double> n2;
    Test(n1);
    Test(n2);

}
```

A decorative graphic in the top-left corner consists of several overlapping blue triangles of varying shades, creating a sense of depth and perspective.

Template specialization

Templates

- ▶ Templates have a series of limitations:
 - ▶ The biggest limitation is that a method from a class that uses a template has exactly the same behaviour (the only thing that differs is the type of the parameters)
 - ▶ For example, if we define a function Sum with two parameters x and y, in which we return $x+y$, then we can't change this to another operation (e.g. if x si y are of type char return $x*y$ instead $x+y$)
- ▶ This limitation can be overcome if we use specialized templates
- ▶ Specialized templates represent a way in which for a class we define a method that overwrites the initial code from the template with a more specific one for parameters of a certain type.

Templates

- ▶ An example:

App.cpp

```
template <class T>
class Number
{
    T value;
public:
    void Set(T t) { value = t; }
    int Get() { return (int)value; }
};
void main(void)
{
    Number<int> n1;
    n1.Set(5);
    Number<char> n2;
    n2.Set('7');
    printf("n1=%d, n2=%d", n1.Get(), n2.Get());
}
```

- ▶ The code runs - but prints “n1=5, n2=55” even though we initialized n2 with ‘7’ → and we expected to print “n2 = 7”

Templates

- ▶ Another example:

App.cpp

```
template <class T>
class Number
{
    T value;
public:
    void Set(T t) { value = t; }
    int Get() { return (int)value; }
};

template <>
class Number <char>
{
    char value;
public:
    void Set(char t) { value = t-'0'; }
    int Get() { return (int)value; }
};

void main(void)
{
    Number<int> n1;
    n1.Set(5);
    Number<char> n2;
    n2.Set('7');
    printf("n1=%d, n2=%d", n1.Get(), n2.Get());
}
```

Template
specialized for
char

- ▶ The code runs properly and prints n1=5, n2=7

Templates

- ▶ Specialized templates work for functions the same way:

App.cpp

```
template <class T>
int ConvertToInt(T value) { return (int)value; }

template <>
int ConvertToInt<char>(char value) { return (int)(value-'0'); }

void main(void)
{
    int x = ConvertToInt<double>(1.5);
    int y = ConvertToInt<char>('4');
}
```

Template specialized
for **char**

- ▶ The code runs - x will be 1 and y will be 4

The background features a large, abstract graphic on the left side composed of various shades of blue and dark blue triangles and trapezoids, creating a layered, geometric pattern.

Compile-time ► assertion checking

Compile-time assertion checking

- ▶ Templates are a really powerful tool. However, sometimes, one might need to add some constraints. Let's analyze the following example:

App.cpp

```
template <typename T,int size>
class Stack
{
    T Elements[size];
public:
    Stack() {

    }
};

void main(void)
{
    Stack<float,200> a;
}
```

- ▶ How can we enforce some limits for template parameter **size** ? (e.g. we do not want **size** to be smaller than 2 or bigger than 100)

Compile-time assertion checking

- ▶ C++11 standard provides a keyword ***static_assert*** that can produce an evaluation of an expression during the compile phase and throw a compile error:
- ▶ Format:

```
static_assert ( condition, “<error message in case condition is false> ” )
```
- ▶ Example:
`static_assert (a>10, “‘a’ variable should be smaller or equal to 10”);`

This code will fail in the compile phase if “a” can be evaluated at that time (this usually means that “a” is a constant) and if its value is bigger than 10.
- ▶ Usually, ***static_assert*** is used within different class methods (e.g. constructor) to check the parameters or with templates to provide some constraints.

Compile-time assertion checking

- ▶ The previous code can be modified in the following way:

App.cpp

```
template <typename T,int size>
class Stack
{
    T Elements[size];
public:
    Stack() {
        static_assert (size > 1, "Size for Stack must be bigger than 1");
        static_assert (size < 100, "Size for Stack must be smaller than 100");
    }
};
void main(void)
{
    Stack<float,200> a;
```



error C2338: Size for Stack must be smaller than 100
note: while compiling class template member function
'Stack<float,200>::Stack(void)'
note: see reference to function template instantiation
'Stack<float,200>::Stack(void)' being compiled
note: see reference to class template instantiation
'Stack<float,200>' being compiled

- ▶ Now the code no longer compiles and shows the following error:

Compile-time assertion checking

- ▶ In particular for templates, *static_assert* can also be used to allow only a couple of types to be used in a template.
- ▶ This is however, a 2-step process. First we define a template that can tell us if a type is equal (the same) with another type.

App.cpp

```
template<typename T1, typename T2>
struct TypeCompare
{
    static const bool equal = false;
};

template<typename T>
struct TypeCompare<T,T>
{
    static const bool equal = true;
};
```

Specialized template for template
TypeCompare that has both parameters
of the *same type*

- ▶ The trick here is to use a specialized template where the value of a static constant variable is changed.

Compile-time assertion checking

- ▶ The previous code can be modified in the following way:

App.cpp

```
template <typename T,int size>
class Stack
{
    T Elements[size];
public:
    Stack() {
        static_assert (TypeCompare<T,int>::equal, "Stack can only be used with type int");
    }
};
void main(void)
{
    Stack<float,200> a;
```

error C2338: Stack can only be used with type int
note: while compiling class template member function
'Stack<float,200>::Stack(void)'
note: see reference to function template instantiation
'Stack<float,200>::Stack(void)' being compiled
note: see reference to class template instantiation
'Stack<float,200>' being compiled

- ▶ In this case the following happen: When `TypeCompare<float,int>` is called, the compiler chooses the first (more generic form) of the template that has the value of `equal false`. If we would have created a var using `Stack<int,200>` then `TypeCompare<int,int>` would have selected the specialized template that has the value of `equal true`

Compile-time assertion checking

- ▶ Let's analyze the following code:

App.cpp

```
void main(void)
{
    int x = rand();
    static_assert(x != 0, "X should not be 0 !");
}
```

error C2131: expression did not evaluate to a constant
note: failure was caused by a read of a variable outside its lifetime
note: see usage of 'x'

- ▶ Keep in mind that ***static_assert*** only works with constant value (e.g. values that can be deduce during the compile phase). In this case, the value of “x” is a random one (as a result, the compiler can not evaluate if “x” is 0 or not and an error will be reported).

Q & A

OOP

Gavrilut Dragos
Course 7

Summary

- ▶ Standard Template Library (STL)
- ▶ Sequence containers
- ▶ Adaptors
- ▶ Associative containers
- ▶ I/O Streams
- ▶ Strings
- ▶ Initialization lists

The background features a large, abstract graphic on the left side composed of overlapping blue and dark blue triangles of varying sizes and orientations, creating a sense of depth and motion.

Standard Template Library (STL)

STL (Standard Template Library)

- ▶ A set of templates that contains:
 - ▶ Containers (class templates that may contain other classes)
 - ▶ Iterators (pointer used for template iteration)
 - ▶ Algorithms (functions that can be called for a container)
 - ▶ Adaptors
 - ▶ Allocators
 - ▶ Others
- ▶ To use an STL template, we have to include the header(s) where that template is defined.
- ▶ STL object can be initialized in the following way: (“`vector<int> x`”) if we use “`using namespace std;`”, or using their full scope definition (“`std::vector<int> x`”) otherwise.
- ▶ **There may be various implementation of STL objects. As these implementation are compiler specific, their performance varies from one compiler to another.**

STL-Containere

- ▶ **Sequence containers**

- ▶ vector
- ▶ Array
- ▶ list
- ▶ forward_list
- ▶ deque
- ▶ Adaptors (stack, queue, priority_queue)

- ▶ **Associative containers**

- ▶ Ordered: set, multiset, map, multimap
- ▶ Un-ordered: unordered_set, unordered_map, unordered_multiset, unordered_multimap

The background features a large, abstract graphic on the left side composed of overlapping blue triangles of varying shades of blue, from dark navy to light cyan. It has a subtle, organic, fan-like or leaf-like pattern.

Sequence ► containers

STL - Vector

- ▶ Vector = an unidimensional array for objects
- ▶ Constructors:

App.cpp

```
using namespace std;
#include <vector>

void main(void)
{
    vector<Type> v;
    vector<Type> v(Size);
}
```

- ▶ Memory allocation / resizing is done dynamically
- ▶ Every object added in a **vector** is actually a copy of the original one

STL - Vector

- ▶ Insertion in a vector is done using the following commands: **push_back**, **insert**
- ▶ For deletion, we can use: **pop_back**, **erase** or **clear**
- ▶ For reallocation: the **resize** and **reserve** methods
- ▶ For access to elements: the index operator and the “**at**” method

App.cpp

```
using namespace std;
#include <vector>

void main(void)
{
    vector<int> v;
    v.push_back(1);v.push_back(2);
    int x = v[1];
    int y = v.at(0);
}
```

STL - Vector

App.cpp

```
using namespace std;
#include <vector>

class Integer
{
    int Value;
public:
    Integer() : Value(0) {}
    Integer(int v) : Value(v) {}
    Integer(const Integer &v) : Value(v.Value) {}
    void Set(int v) { Value = v; }
    int Get() { return Value; }
};

void main(void)
{
    vector<Integer> v;
    Integer i(5);
    v.push_back(i);
    i.Set(6);
    printf("i=%d,v[0]=%d\n", i.Get(), v[0].Get());
}
```

After execution: i=6,v[0]=5

STL - Vector

Example:

App.cpp

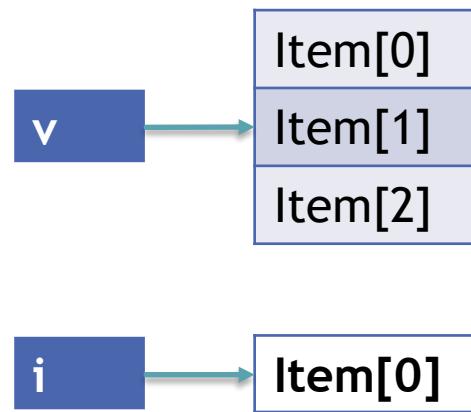
```
class Integer
{
    int Value;
public:
    Integer() : Value(0) { printf("[%p] Default ctor\n", this); }
    Integer(int v) : Value(v) { printf("[%p] Value ctor(%d)\n", this,v); }
    Integer(const Integer &v) : Value(v.Value) { printf("[%p] Copy ctor from (%p,%d)\n",
                                                       this, &v, v.Value); }
    Integer& operator= (const Integer& i) { Value = i.Value; printf("[%p] op= (%p,%d)\n",
                                                               this, &i, i.Value); return *this; }
    void Set(int v) { Value = v; }
    int Get() { return Value; }
};
void main(void)
{
    vector<Integer> v;
    Integer i(5);
    for (int tr = 0; tr < 5; tr++)
    {
        i.Set(1000 + tr);
        v.push_back(i);
    }
}
```

STL - Vector

Example:

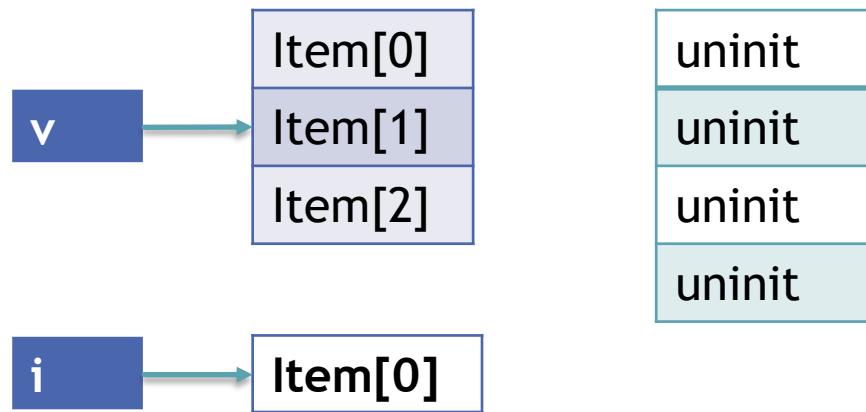
Pas	Output
-	[0098FC90] Value ctor(5)
tr=0	[00D3A688] Copy ctor from (0098FC90,1000)
tr=1	[00D3A6C8] Copy ctor from (00D3A688,1000) [00D3A6CC] Copy ctor from (0098FC90,1001)
tr=2	[00D3A710] Copy ctor from (00D3A6C8,1000) [00D3A714] Copy ctor from (00D3A6CC,1001) [00D3A718] Copy ctor from (0098FC90,1002)
tr=3	[00D3A688] Copy ctor from (00D3A710,1000) [00D3A68C] Copy ctor from (00D3A714,1001) [00D3A690] Copy ctor from (00D3A718,1002) [00D3A694] Copy ctor from (0098FC90,1003)
tr=4	[00D3A6D8] Copy ctor from (00D3A688,1000) ... [00D3A6E8] Copy ctor from (0098FC90,1004)

STL - Vector



`v.push_back(i)`

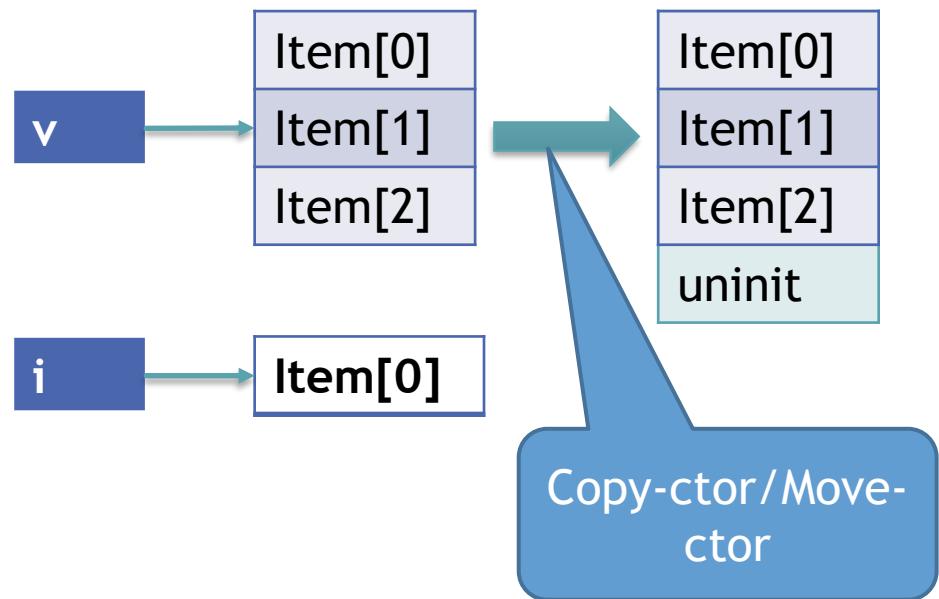
STL - Vector



`v.push_back(i)`

- ▶ Allocate space for `count+1` elements

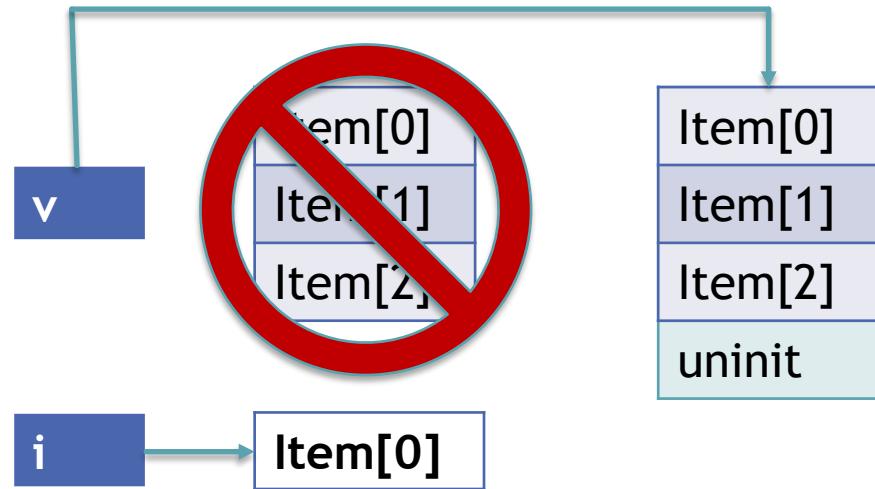
STL - Vector



`v.push_back(i)`

- ▶ Allocate space for `count+1` elements
- ▶ Copy/Move the first `count` elements in the new allocated space

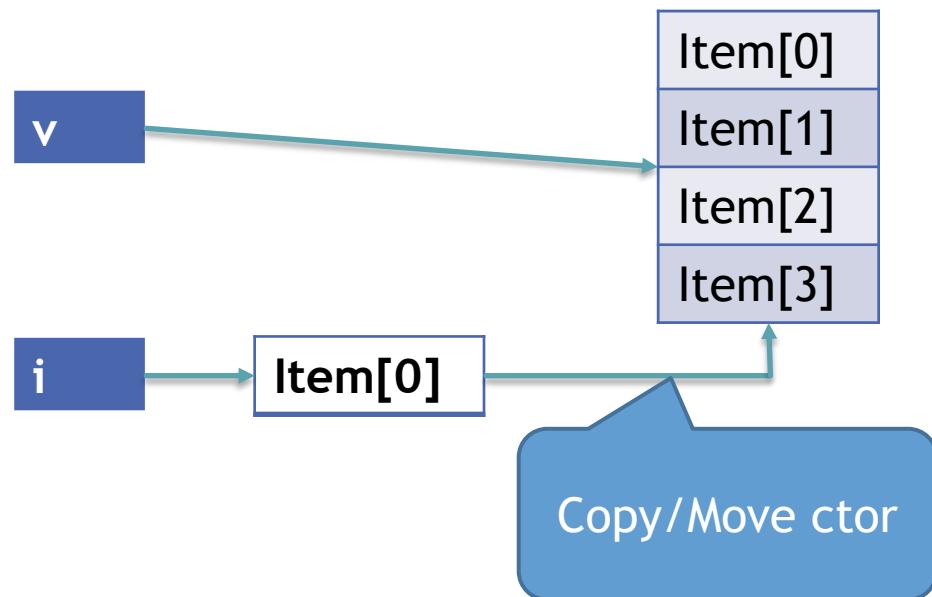
STL - Vector



`v.push_back(i)`

- ▶ Allocate space for `count+1` elements
- ▶ Copy/Move the first `count` elements in the new allocated space
- ▶ Free the space used by the current elements

STL - Vector



`v.push_back(i)`

- ▶ Allocate space for `count+1` elements
- ▶ Copy/Move the first `count` elements in the new allocated space
- ▶ Free the space used by the current elements
- ▶ Copy/Move the new item into the extra allocated space

STL - Vector

Test case:

App-1.cpp

```
#define INTEGER_SIZE 1000
class Integer
{
    int Values[INTEGER_SIZE];
public:
    Integer() { Set(0); }
    Integer(int value) { Set(value); }
    Integer(const Integer &v) { CopyFrom((int*)v.Values); }
    Integer& operator= (const Integer& i) { CopyFrom((int*)i.Values); return *this; }

    void Set(int value)
    {
        for (int tr = 0; tr < INTEGER_SIZE; tr++)
            Values[tr] = value;
    }
    void CopyFrom(int* lista)
    {
        for (int tr = 0; tr < INTEGER_SIZE; tr++)
            Values[tr] = lista[tr];
    }
    void Set(const Integer &v)
    {
        CopyFrom((int*)v.Values);
    }
};
```

STL - Vector

Test case:

Vec-1.cpp

```
void main(void)
{
    vector<Integer> v;
    Integer i;
    for (int tr = 0; tr < 100000; tr++)
    {
        i.Set(tr);
        v.push_back(i);
    }
}
```

Vec-2.cpp

```
void main(void)
{
    vector<Integer> v;
    Integer i;
    v.reserve(100000);
    for (int tr = 0; tr < 100000; tr++)
    {
        i.Set(tr);
        v.push_back(i);
    }
}
```

Ptr-1.cpp

```
void main(void)
{
    Integer *v = new Integer[100000];
    Integer i;
    for (int tr = 0; tr < 100000; tr++)
    {
        i.Set(tr);
        v[tr].Set(i);
    }
}
```

Ptr-2.cpp

```
void main(void)
{
    Integer **v = new Integer*[100000];
    for (int tr = 0; tr < 100000; tr++)
    {
        v[tr] = new Integer(tr);
    }
}
```

STL - Vector

The study was conducted in the following way:

- ▶ Each of the 4 applications was executed for 10 time. Execution time was measured in milliseconds.
- ▶ Execution time was divided into two times: time needed to initialize the container and the time needed to add the data into the container.

App-1.cpp

```
void main(void)
{
    vector<Integer> v;
    Integer i;

    for (int tr = 0; tr < 100000; tr++)
    {
        ...
    }
}
```

Initialization time frame

Add data to container time frame

- ▶ All of these tests were conducted using the following specifications:
 - ▶ OS: Windows 8.1 Pro
 - ▶ Compiler: cl.exe [18.00.21005.1 for x86]
 - ▶ Hardware: Dell Latitude 7440 - i7 - 4600U, 2.70 GHz, 8 GB RAM

STL - Vector

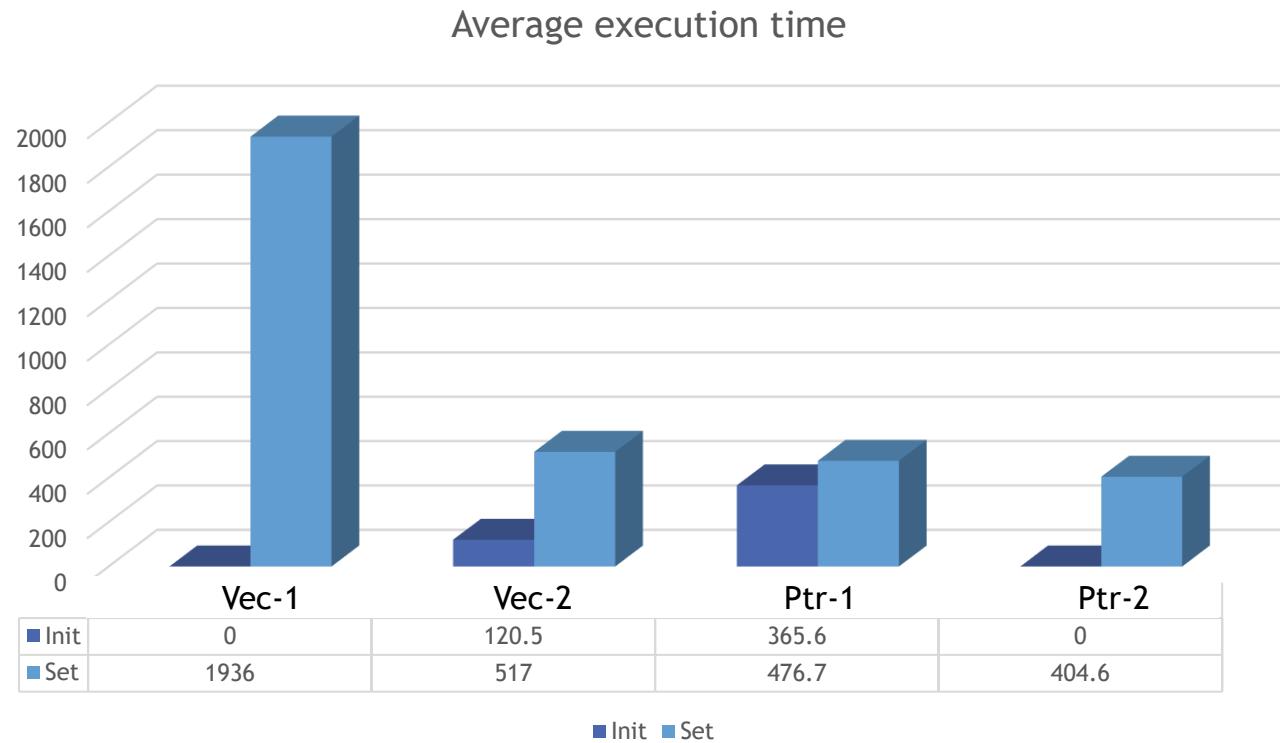
Results:

Alg	Time	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Vec-1	Init	0	0	0	0	0	0	0	0	0	0
	Set	1985	1922	1937	1922	1922	1953	1937	1938	1922	1922
Vec-2	Init	140	125	157	125	109	110	110	109	110	110
	Set	531	515	515	516	516	515	531	516	515	500
Ptr-1	Init	406	359	360	375	359	360	359	359	359	360
	Set	485	485	468	469	485	468	469	485	469	484
Ptr-2	Init	0	0	0	0	0	0	0	0	0	0
	Set	422	453	453	437	375	391	390	375	375	375

- ▶ Vec-1,Vec-2,Ptr-1,Ptr-2 → the 4 methods previously described
- ▶ “Init” → the series of times recorded when we initialize the container
- ▶ “Set” → the series of times needed to add the data in in container
- ▶ T1 .. T10 → result times

STL - Vector

Results:



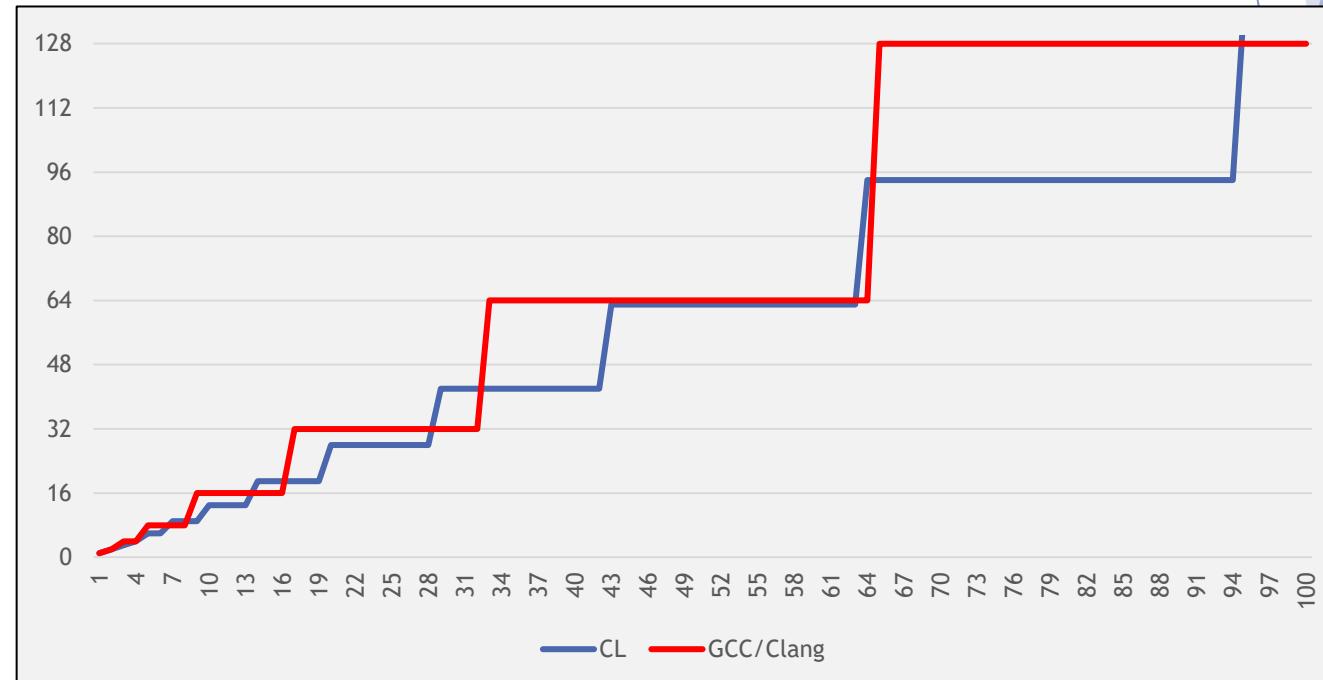
- ▶ Vec-1, Vec-2, Ptr-1, Ptr-2 → the 4 methods previously described
- ▶ “Init” → the series of times recorded when we initialize the container
- ▶ “Set” → the series of times needed to add the data in in container

STL - Vector

- ▶ Let's consider the following code:
- ▶ If we run this code on g++, clang and cl.exe (Microsoft)
- ▶ The results show that the allocation strategy is different depending on the compiler.
- ▶ Clang/G++ prefers powers of 2.

App-1.cpp

```
void main(void) {  
    vector<int> v;  
    for (int tr = 0; tr < 1000; tr++) {  
        v.push_back(0);  
        printf("Elem: %4d, Allocated: %4d\n", v.size(), v.capacity());  
    }  
}
```



STL - Vector

- ▶ To iterate through all of the elements stored in a `vector` we can use *iterators*. An *iterator* is a special object (similar to a pointer)

App.cpp

```
void main(void)
{
    vector<int> v;
    v.push_back(1); v.push_back(2); v.push_back(3); v.push_back(4); v.push_back(5);
    vector<int>::iterator it;
    it = v.begin();
    while (it < v.end())
    {
        printf("%d ", (int)(*it));
        it++;
    }
}
```

- ▶ Iterators work by overloading the following operators:
 - `operator++` , `operator--` (to go forward/backwards within the container)
 - Pointer related operators (`operator->`) to gain access to an element from the container

STL - Vector

- ▶ In this example access to elements is done via **operator→**
- ▶ **vector** template also provides two methods: **front** and **back** that allows access to the first and last elements stored.

App.cpp

```
class Number
{
public:
    int Value;
    Number() : Value(0) {}
    Number(int val) : Value(val) {}
};

void main(void)
{
    vector<Number> v;
    v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
    v.push_back(Number(4)); v.push_back(Number(5));

    vector<Number>::iterator i = v.begin();
    printf("%d\n", i->Value);
    printf("%d\n", (i + 2)->Value);
    printf("%d %d", v.front().Value, v.back().Value);
}
```

- ▶ This example will print on the screen: 1 3 1 5

STL - Vector

- ▶ `vector` template also provides a special iterator (called `reverse_iterator`) that allows iterating/moving through the elements of the vector in the reverse order.

App.cpp

```
class Number
{
...
};

void main(void)
{
    vector<Number> v;
    v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
    v.push_back(Number(4)); v.push_back(Number(5));

    vector<Number>::iterator it;
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);

    vector<Number>::reverse_iterator rit;
    for (rit = v.rbegin(); rit != v.rend(); rit++)
        printf("%d ", rit->Value);
}
```

- ▶ This example will print: “1 2 3 4 5 5 4 3 2 1”

STL - Vector

- ▶ Inserting elements in a vector:

App.cpp

```
class Number
{
...
};

void main(void)
{
    vector<Number> v;
    v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
    v.push_back(Number(4)); v.push_back(Number(5));
    vector<Number>::iterator i = v.begin();

    v.insert(i + 2, Number(10));

    while (i < v.end())
    {
        printf("%d,", i->Value);
        i++;
    }
}
```

- ▶ This example will print:

1,2,**10**,3,4,5

STL - Vector

- ▶ Deleting an element from the vector: (*v.erase(iterator)*)

App.cpp

```
class Number
{
...
};

void main(void)
{
    vector<Number> v;
    v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
    v.push_back(Number(4)); v.push_back(Number(5));
    vector<Number>::iterator i = v.begin();
    v.erase(i+2);
    v.erase(i+3);
    while (i < v.end())
    {
        printf("%d,", i->Value);
        i++;
    }
}
```

- ▶ This example will print: 1,2,4

STL - Vector

- ▶ This code compiles. However, keep in mind that after an element is deleted, an iterator can be invalid. As a general rule, don't *REUSE* an interator after it was deleted like in the next example: “**v.erase(i+1)**”

App.cpp

```
class Number
{
...
};

void main(void)
{
    vector<Number> v;
    v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
    v.push_back(Number(4)); v.push_back(Number(5));
    vector<Number>::iterator i = v.begin();
    v.erase(i);
v.erase(i+1);
    while (i < v.end())
    {
        printf("%d, ", i->Value);
        i++;
    }
}
```

- ▶ During the execution, an exception will be triggered on “**v.erase(i+1)**”

STL - Vector

- ▶ To fix the previous problem, we don't re-use an iterator when we delete an element. Instead, we compute an iterator in place (using `v.begin()` to do this).

App.cpp

```
class Number
{
...
};

void main(void)
{
    vector<Number> v;
    v.push_back(Number(1)); v.push_back(Number(2)); v.push_back(Number(3));
    v.push_back(Number(4)); v.push_back(Number(5));
    v.erase(v.begin());
    v.erase(v.begin() + 1);
    vector<Number>::iterator i = v.begin();
    while (i < v.end())
    {
        printf("%d, ", i->Value);
        i++;
    }
}
```

- ▶ This code compiles and works correctly.

STL - Vector

- ▶ Other operators that are overwritten are relationship operators (`>`, `<`, `!=`, ...)
- ▶ Two vectors are considered to be equals if they have the same number of elements and elements with the same index in those two vectors are equal.

App.cpp

```
class Number
{
...
};

void main(void)
{
    vector<Number> v1;
    vector<Number> v2;
    for (int tr = 0; tr < 10; tr++)
    {
        v1.push_back(Number(tr));
        v2.push_back(Number(tr));
    }
    if (v1 == v2)
        printf("V1 is equal with V2");
    else
        printf("V2 is different than V2");
}
```

- ▶ This example will not compile as `Number` does not have an `operator==` defined !

STL - Vector

- ▶ Make sure that you define *operator==* as *const*. *vector* template requires a *const operator==* to work.
- ▶ This code will not compile.

App.cpp

```
class Number
{
    bool operator==(const Number &n1) { return Value == n1.Value; }
};

void main(void)
{
    vector<Number> v1;
    vector<Number> v2;
    for (int tr = 0; tr < 10; tr++)
    {
        v1.push_back(Number(tr));
        v2.push_back(Number(tr));
    }
    if (v1 == v2)
        printf("V1 is equal with V2");
    else
        printf("V2 is different than V2");
}
```

STL - Vector

- ▶ Now this code compiles and work as expected.
- ▶ You can also replace *operator==* with a friend function with the following syntax: “**bool friend operator== (const Number & n1, const Number & n2);**“

App.cpp

```
class Number
{
    bool operator==(const Number &n1) const { return Value == n1.Value; }
};

void main(void)
{
    vector<Number> v1;
    vector<Number> v2;
    for (int tr = 0; tr < 10; tr++)
    {
        v1.push_back(Number(tr));
        v2.push_back(Number(tr));
    }
    if (v1 == v2)
        printf("V1 is equal with V2");
    else
        printf("V2 is different than V2");
}
```

STL - Vector

- ▶ For < and > compare operations, **vector** template uses the following algorithm (see the pseudo-code below) to decide if a vector is bigger/smaller than another one.

Pseudocode

```
Function IsSmaller (Vector v1, Vector v2)
    Size = MINIM(v1.Size, v2.Size)
    for (i=0; i<Size; i++)
        if (v1[i] < v2[i])
            return "v1 is smaller than v2";
        end if
    end for
    if (v1.Size < v2.Size)
        return "v1 is smaller than v2";
    end if
    return "v1 is NOT smaller than v2";
End Function
```

App.cpp

```
void main(void)
{
    vector<int> v1;
    vector<int> v2;

    v1.push_back(10);
    v2.push_back(5); v2.push_back(20);

    if (v2 < v1)
        printf("v2<v1");
}
```

- ▶ One important observation here is that **operator>** is not needed (only **operator<** is needed to decide).
- ▶ The previous code will print “v2<v1” (even if “v2” has two elements, and “v1” only one).

STL - Vector

- ▶ Let's analyze the following example:

App.cpp

```
class Number
{
    bool operator<(const Number &n1) const { return Value < n1.Value; }
};

void main(void)
{
    vector<Number> v1;
    vector<Number> v2;

    v1.push_back(Number(1)); v1.push_back(Number(2)); v1.push_back(Number(3));
    v2.push_back(Number(1)); v2.push_back(Number(2)); v2.push_back(Number(4));

    if (v1 < v2)
        printf("OK");
    else
        printf("NOT-OK");
}
```

- ▶ This code compiles and upon execution prints “OK” on the screen as $v1[2]=3$ and $v2[2]=4$, and $3 < 4$

STL - Vector

- ▶ This code will not compile as *operator<* is not defined.

App.cpp

```
class Number
{
    bool operator>(const Number &n1) const { return Value < n1.Value; }
};

void main(void)
{
    vector<Number> v1;
    vector<Number> v2;

    v1.push_back(Number(1)); v1.push_back(Number(2)); v1.push_back(Number(3));
    v2.push_back(Number(1)); v2.push_back(Number(2)); v2.push_back(Number(4));

    if (v1 > v2)
        printf("OK");
    else
        printf("NOT-OK");
}
```

- ▶ *vector* template requires *operator<* to be defined in class *Number*. Similarly, *operator!=* is not needed, only *operator==* is.

STL - Vector

- ▶ You can also copy the values from one vector into another.

App.cpp

```
class Number
{
    ...
};

void main(void)
{
    vector<Number> v1;
    vector<Number> v2;

    v1.push_back(Number(1)); v1.push_back(Number(2)); v1.push_back(Number(3));

    v2 = v1;

    for (int i = 0; i < v2.size(); i++)
        printf("%d ", v2[i].Value);
}
```

- ▶ This code compiles and prints 1 2 3

STL - Vector

- ▶ The following methods from template ***vector*** can be used to obtain different information:
 - ▶ `size()` → the amount of elements stored in the vector
 - ▶ `capacity()` → the pre-allocated space in the vector
 - ▶ `max_size()` → maximum number of elements that can be stored in the vector
 - ▶ `empty()` → returns “*true*” if the current vector has no elements
 - ▶ `data()` → provides direct access (a pointer) to all of the elements in the vector.

App.cpp

```
class Number { ... };

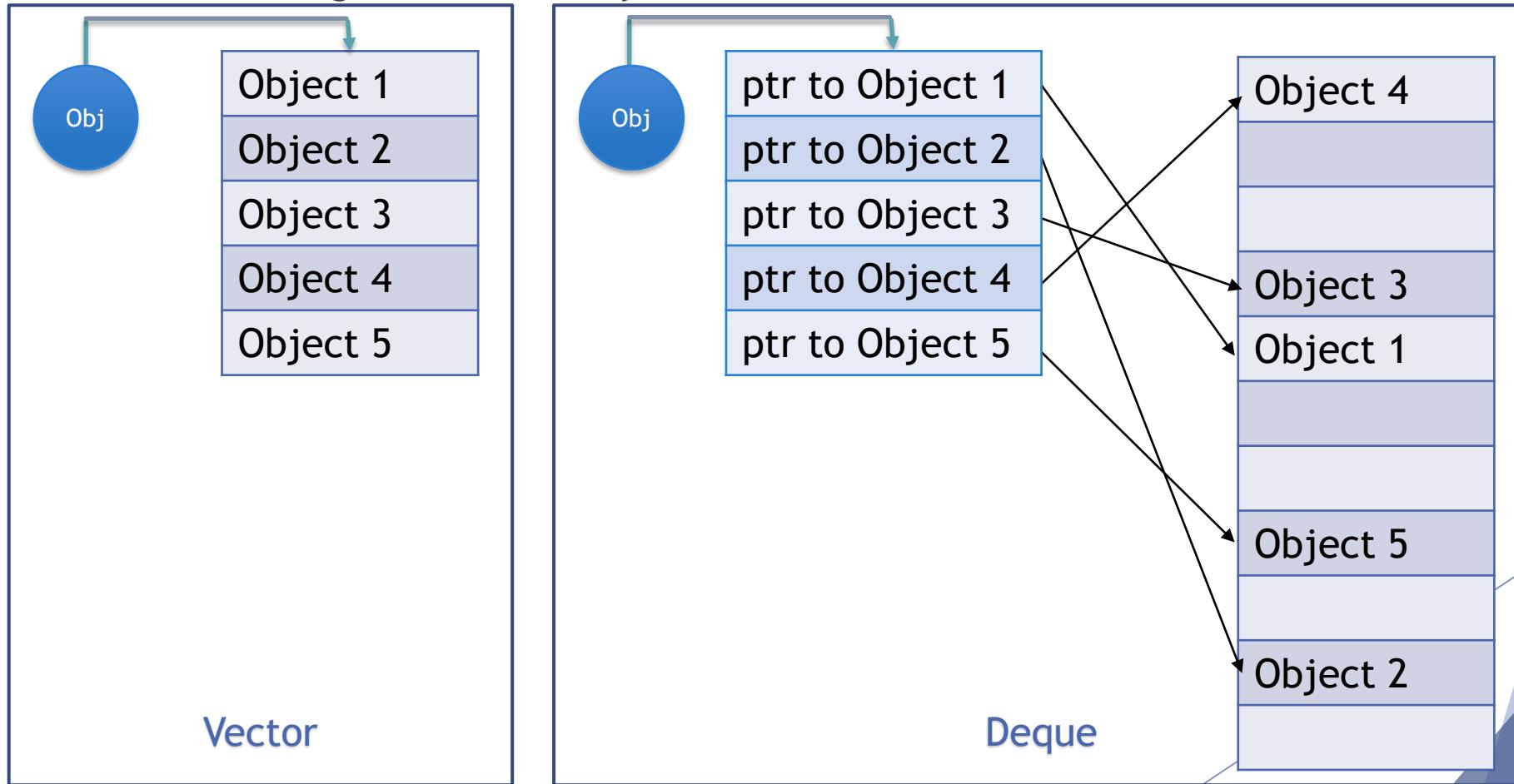
void main(void)
{
    vector<Number> v;
    v.push_back(Number(0)); v.push_back(Number(1));
    printf("%d ", v.max_size());
    printf("%d ", v.size());
    printf("%d ", v.capacity());
    Number *n = v.data();
    printf("[%d %d]", n[0].Value, n[1].Value);
}
```

STL - Deque

- ▶ The “*deque*” template is very similar to “*vector*”.
- ▶ The main different is that elements don't have consecutive memory addresses. That is why the “*data*” method is not available for a deque. This also goes for the “*reserve*” and “*capacity*” methods.
- ▶ But deque has new methods (*push_front* and *pop_front*)
- ▶ The rest of the methods behave just like the ones in the “*vector*” class
- ▶ In order to use a deque: “`#include <deque>`”

STL - Deque

This is an illustrative picture that shows the difference between vector and deque. It should be NOTED that this is but am illustrative picture and does not reflect how the algorithms actually work.



STL - Deque

Let's retest the previous algorithm - this time we will use deque as well.

Vec-1.cpp

```
void main(void)
{
    vector<Integer> v;
    Integer i;
    for (int tr = 0; tr < 100000; tr++)
    {
        i.Set(tr);
        v.push_back(i);
    }
}
```

Vec-2.cpp

```
void main(void)
{
    vector<Integer> v;
    Integer i;
    v.reserve(100000);
    for (int tr = 0; tr < 100000; tr++)
    {
        i.Set(tr);
        v.push_back(i);
    }
}
```

Deq-1.cpp

```
void main(void)
{
    deque<Integer> v;
    Integer i;
    for (int tr = 0; tr < 100000; tr++)
    {
        i.Set(tr);
        v.push_back(i);
    }
}
```

STL - Deque

Results:

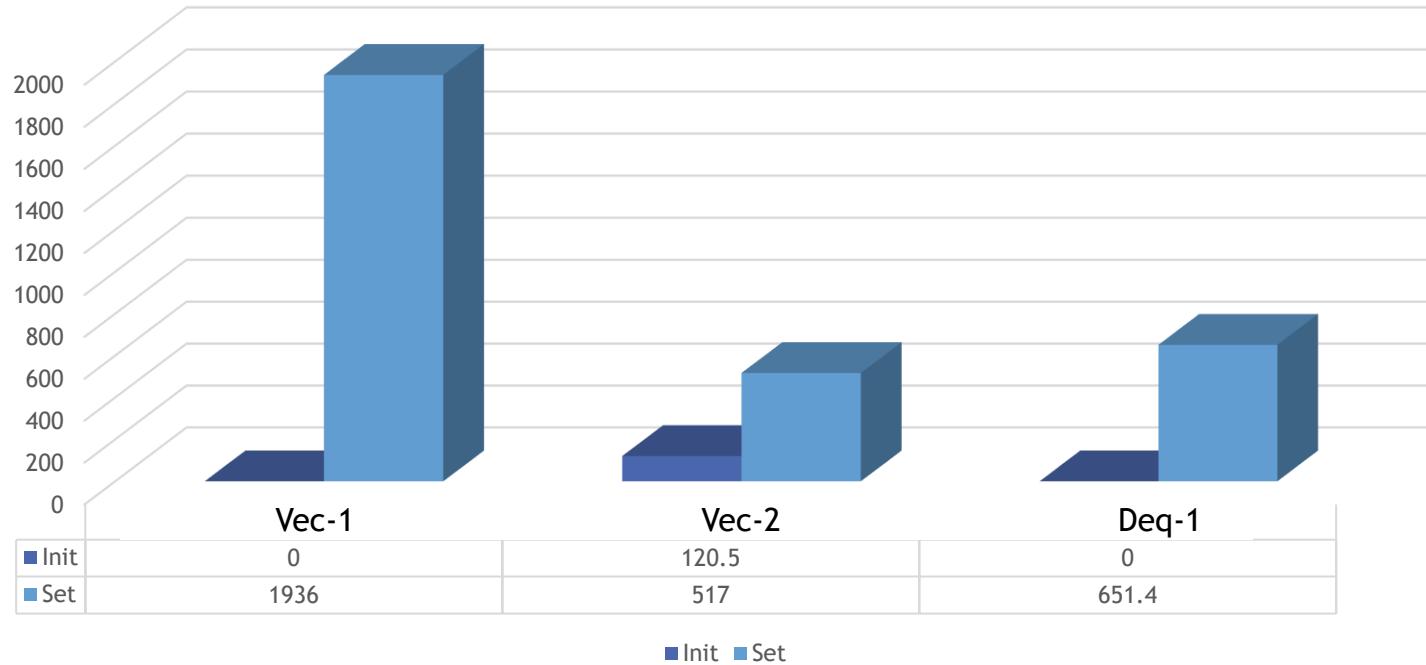
Alg	Time	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Vec-1	Init	0	0	0	0	0	0	0	0	0	0
	Set	1985	1922	1937	1922	1922	1953	1937	1938	1922	1922
Vec-2	Init	140	125	157	125	109	110	110	109	110	110
	Set	531	515	515	516	516	515	531	516	515	500
Deq-1	Init	0	0	0	0	0	0	3509	0	0	0
	Set	687	657	656	640	656	640	656	641	641	640

- ▶ Vec-1 and Vect-2 → two methods using **vector**
- ▶ Deq-1 → same algorithm using **deque**
- ▶ “Init” → the series of times recorded when we initialize the container
- ▶ “Set” → the series of times needed to add the data in in container
- ▶ T1 .. T10 → result times

STL - Deque

Results:

Execution time



- ▶ Vec-1 and Vect-2 → two methods using `vector`
- ▶ Deq-1 → same algorithm using `deque`
- ▶ “Init” → the series of times recorded when we initialize the container
- ▶ “Set” → the series of times needed to add the data in in container

STL - Array

- ▶ The “array” template has been introduced in the C++11 standard.
- ▶ It represents a fixed size vector
- ▶ It has almost the same support as the vector class (iterators, overloaded operators, etc)
- ▶ It doesn't have any add methods (having a fixed size, it doesn't need them)
- ▶ For the same reason, it doesn't have any methods that can be used to erase an element
- ▶ It has some methods that vector does not have (e.g. fill)
- ▶ In order to use an array: “**#include <array>**”

STL - Array

- ▶ An example on how to use array template:

App.cpp

```
class Number
{
    ...
};

void main(void)
{
    array<Number,5> v;
    v.fill(Number(2));
    for (int tr = 0; tr < v.size(); tr++)
        printf("%d ", v[tr].Value);

    for (array<Number, 5>::iterator it = v.begin(); it < v.end(); it++)
        it->Value = 10;

    printf("%d ", v.at(3).Value);
}
```

- ▶ In case of *array* template, the size is predefined → this means that is somehow equivalent to a *vector* template that is initialized with using *resize()* method.

STL - Array & vector

- ▶ Both ***array*** and ***vector*** have 2 ways of accessing elements: the `[]` operator and the “at” method
- ▶ They both return an object reference and have 2 forms:
 - `Type& operator[](size_type)`
 - `const Type& operator[](size_type) const`
 - `Type& at(size_type)`
 - `const Type& at(size_type)`
 - `size_type` is defined as `size_t` (`typedef size_t size_type;`)
- ▶ There are differences between these two implementations that will be discussed in the next slides

STL - Array & vector

- ▶ Let's see how *operator[]* and method “*at*” are defined ?

App.cpp

```
reference at(size_type _Pos)
{
    if (_Size <= _Pos)
        _Xran();
    return (_Elems[_Pos]);
}
```

App.cpp

```
reference operator[](size_type _Pos)
{
#if _ITERATOR_DEBUG_LEVEL == 2
    if (_Size <= _Pos)
        _DEBUG_ERROR("array subscript out of range");
#elif _ITERATOR_DEBUG_LEVEL == 1
    _SCL_SECURE_VALIDATE_RANGE(_Pos < _Size);
#endif
    _Analysis_assume_(_Pos < _Size);
    return (_Elems[_Pos]);
}
with
#define _Analysis_assume_(expr) // for release
```

Compile method	Value of _ITERATOR_DEBUG_LEVEL
Release	0
Release (_SECURE_SCL)	1
Debug	2

STL - List

- ▶ The “*list*” template is a doubly-linked list
- ▶ Elements do not share a contiguous memory block
- ▶ Element access is possible only by moving through the list using iterators. The first and last element are easier to access
- ▶ The list iterator doesn't implement the < operator - the elements aren't stored in consecutive memory addresses, and so this type of comparison between these addresses is pointless. It does, however, implement the == operator.
- ▶ The list iterator does not implement the + and - operators; only ++ and -- are implemented
- ▶ Methods used to add elements: **push_back** and **push_front**
- ▶ To remove elements: **erase**, **pop_front**, or **pop_back** can be used
- ▶ It supports a series of new methods: **merge**, **splice** (work with other lists)
- ▶ In order to use a list: “**#include <list>**”

STL - List

- ▶ An example of using the *list* template

App.cpp

```
class Number { ... };
void main(void)
{
    list<Number> v;
    v.push_back(Number(0)); v.push_back(Number(1)); v.push_back(Number(2));
    v.push_front(Number(3)); v.push_front(Number(4)); v.push_front(Number(5));

    list<Number>::iterator it;
    for (it = v.begin(); it < v.end(); it++)
        printf("%d ", it->value);

    it = v.begin() + 3;
    v.insert(it, Number(20));
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->value);

    it = ++v.begin();
    v.erase(it);
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->value);
}
```

- ▶ This code will not compile as the *list* template does not have *operator<* and *operator>* defined.

STL - List

- ▶ An example of using the *list* template

App.cpp

```
class Number { ... };
void main(void)
{
    list<Number> v;
    v.push_back(Number(0)); v.push_back(Number(1)); v.push_back(Number(2));
    v.push_front(Number(3)); v.push_front(Number(4)); v.push_front(Number(5));

    list<Number>::iterator it;
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);

    it = v.begin() + 3;
    v.insert(it, Number(20));
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);

    it = ++v.begin();
    v.erase(it);
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);
}
```

- ▶ This code will not compile as the *list* template does not have *operator+* defined.

STL - List

- ▶ An example of using the *list* template

App.cpp

```
class Number { ... };
void main(void)
{
    list<Number> v;
    v.push_back(Number(0)); v.push_back(Number(1)); v.push_back(Number(2));
    v.push_front(Number(3)); v.push_front(Number(4)); v.push_front(Number(5));

    list<Number>::iterator it;
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);

    it = v.begin();it++;it++;it++;
    v.insert(it, Number(20));
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);

    it = ++v.begin();
    v.erase(it);
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);
}
```

- ▶ Now the code compiles and prints: “5 4 3 0 1 2 5 4 3 20 0 1 2 5 3 20 0 1 2“

STL - Forward List

- ▶ The “*forward_list*” template is a single linked list container. It has been added in the C++11 standard
- ▶ It respects almost the same logic as in the case of the doubly linked list.
- ▶ The iterator doesn't overload the -- operator, only ++ (the list is unidirectional)
- ▶ Some methods that list has are no longer found here: push_back and pop_back
- ▶ In order to use a *forward_list*: “**#include <forward_list>**”

STL - Forward List

- ▶ An example of using the *forward_list* template

App.cpp

```
class Number { ... };
void main(void)
{
    forward_list<Number> v;
    v.push_front(Number(0)); v.push_front(Number(1)); v.push_front(Number(2));

    forward_list<Number>::iterator it;
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);

    it = v.begin();
    it++; it++;
    v.insert_after(it, Number(20));
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);

    it = ++v.begin();
    v.erase_after(it);
    for (it = v.begin(); it != v.end(); it++)
        printf("%d ", it->Value);
}
```

- ▶ Now the code compiles and prints: “2 1 0 2 1 0 20 2 1 20“

STL

Method	vector	Deque	array	list	forward_list
Access	operator[] .at() .front() .back() .data()	operator[] .at() .front() .back()	operator[] .at() .front() .back() .data()	.front() .back()	.front()
Iterators	.begin() .end()	.begin() .end()	.begin() .end()	.begin() .end()	.begin() .end()
Reverse Iterator	.rbegin() .rend()	.rbegin() .rend()	.rbegin() .rend()	.rbegin() .rend()	
Information	.empty() .size() .max_size()	.empty() .size() .max_size()	.empty() .size() .max_size()	.empty() .size() .max_size()	.empty() .max_size()

STL

Method	vector	Deque	array	list	forward_list
Size/ Capacity	.resize() .reserve() .capacity()	.resize()		.resize()	.resize()
Add	.push_back() .insert()	.push_back() .push_front() .insert()		.push_back() .push_front() .insert()	.push_front() .insert_after()
Delete	.clear() .erase() .pop_back()	.clear() .erase() .pop_back() .pop_front()		.clear() .erase() .pop_back() .pop_front()	.clear() .pop_front() .erase_after()

A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented diagonally, creating a sense of depth and movement.

- ▶ Adaptors

STL - Adaptors

- ▶ Adaptors are not containers - in the sense that they do not have an implementation that can store, they use an existing container that have this feature for storage
- ▶ Adaptors only work if the container that they use implement some specific functions
- ▶ Adaptors do NOT have iterators, but they can use iterators from the container that they use internally
- ▶ Adaptors:
 - stack
 - queue
 - priority_queue

STL - Stack

- ▶ This adaptor implements a stack (LIFO - Last In First Out)
- ▶ To use : “**#include <stack>**”
- ▶ The default container is “**deque**” (in this example “s” uses **deque** while “s2” uses **vector**)

App.cpp

```
void main(void)
{
    stack<int> s;
    s.push(10); s.push(20); s.push(30);

    stack<int, vector<int>> s2;
    s2.push(10); s2.push(20); s2.push(30);
}
```

- ▶ Has the following methods: **push, pop, top, empty, size**
- ▶ It also implements a method called “**_Get_container**” that can be used to create iterators based on the container that is being used internally.

STL - Queue

- ▶ This adaptor implements a queue (FIFO - First In First Out)
- ▶ To use : “**#include <queue>**”
- ▶ The default container is “**deque**” (in this example “s” uses **deque** while “s2” uses **list**)

App.cpp

```
void main(void)
{
    queue<int> s;
    s.push(10); s.push(20); s.push(30);

    queue<int, list<int>> s2;
    s2.push(10); s2.push(20); s2.push(30);
}
```

- ▶ Has the following methods: **push, pop, back, empty, size**
- ▶ It also implements a method called “**_Get_container**” that can be used to create iterators based on the container that is being used internally.

STL - Priority Queue

- ▶ This is a queue where each elements has a priority. All elements are sorted out based on their priority (so that the element with the highest priority is extracted first).
- ▶ To use : “**#include <queue>**”
- ▶ The default container is “**vector**”

App.cpp

```
void main(void)
{
    priority_queue<int> s;
    s.push(10); s.push(5); s.push(20); s.push(15);
    while (s.empty() == false)
    {
        printf("%d ", s.top());
        s.pop();
    }
}
```

- ▶ This code prints: “**20 15 10 5**”
- ▶ Has the following methods: ***push, pop, top, empty, size***
- ▶ It also implements a method called “***_Get_container***” that can be used to create iterators based on the container that is being used internally.

STL - Priority Queue

- ▶ A *priority_queue* can be initialized with a class that has to implement “operator()”. This operator is used to compare two elements (it should return true if the first one is smaller than the second one).

App.cpp

```
class CompareModule
{
    int modValue;
public:
    CompareModule(int v) : modValue(v) {}
    bool operator() (const int& v1, const int& v2) const
    {
        return (v1 % modValue) < (v2 % modValue);
    }
};
void main(void)
{
    priority_queue<int, vector<int>, CompareModule> s(CompareModule(3));
    s.push(10); s.push(5); s.push(20); s.push(15);
    while (s.empty() == false)
    {
        printf("%d ", s.top());
        s.pop();
    }
}
```

- ▶ This example prints: “5 20 10 15”

STL - Adaptors

Method	stack	queue	Priority_queue
push	Container.push_back	Container.push_back	Container.push_back
pop	Container.pop_back	Container.pop_front	Container.pop_back
top	Container.back	N/A	Container.front
back	N/A	Container.back	N/A
size	Container.size	Container.size	Container.size
empty	Container.empty	Container.empty	Container.empty

Adaptor	vector	deque	list	array	forward_list
stack	YES	YES	YES	NO	NO
queue	NO	YES	YES	NO	NO
priority_queue	YES	YES	NO	NO	NO

The background features a large, abstract graphic on the left side composed of various shades of blue and dark blue triangles, creating a layered, polygonal effect.

Associative ▶ containers

STL (associative containers - pair)

- ▶ “pair” is a template that contains two values (two different types)
- ▶ For use “**#include <utility>**”
- ▶ Two objects of type **pair** can be compared because the = operator is overloaded
- ▶ It’s internally used by associative containers
- ▶ The declaration of **pair** template:

App.cpp

```
template <class T1, class T2>
struct pair
{
    T1 first;
    T2 second;
    ...
}
```

STL (associative containers - map)

- ▶ Map is a container that store pairs of the form: key/value; the access to the **value** field can be done using the **key**
- ▶ For use “**#include <map>**”
- ▶ The key field is constant: after a key/value pair is added into a map, the key cannot be modified - only the value can be modified. Other pairs can be added and existing pairs can be deleted.
- ▶ The declaration of map template:

App.cpp

```
template < class Key, class Value, class Compare = less<Key> > class map
{
...
}
```

STL (associative containers - map)

- ▶ An example of using map:

App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades[“Popescu”] = 10;
    Grades[“Ionescu”] = 9;
    Grades[“Marin”] = 7;
    printf(“Ionescu’s grade = %d\n”, Grades[“Ionescu”]);
    printf(“Number of pairs = %d\n”, Grades.size());
    map<const char*, int>::iterator it;
    for (it = Grades.begin(); it != Grades.end(); it++)
        printf(“Grades[%s]=%d\n”, it->first, it->second);
    it = Grades.find(“Ionescu”);
    Grades.erase(it);
    int x = Grades [“Ionescu”];
    printf(“Ionescu’s grade = %d (x=%d)\n”, Grades[“Ionescu”],x)
    printf("Number of pairs = %d\n", Grades.size());
}
```

Output

```
Ionescu’s grade = 9
Number of pairs = 3
Grades[Popescu]=10
Grades[Ionescu]=9
Grades[Marin]=7
Ionescu’s grade = 0 (x=0)
Number of pairs = 3
```

STL (associative containers - map)

- ▶ An example of using map:

App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades[“Popescu”] = 10;
    Grades[“Ionescu”] = 9;
    Grades[“Marin”] = 7;
    printf(“Ionescu’s grade = %d\n”, Grades[“Ionescu”]);
    printf(“Number of pairs = %d\n”, Grades.size());
    map<const char*, int>::iterator it;
    for (it = Grades.begin(); it != Grades.end(); it++)
        printf(“Grades[%s]=%d\n”, it->first, it->second);
    it = Grades.find(“Ionescu”);
    Grades.erase(it);
    int x = Grades [“Ionescu”];
    printf(“Ionescu’s grade = %d (x=%d)\n”, Grades[“Ionescu”],x);
    printf(“Number of pairs = %d\n”, Grades.size());
}
```

The “Ionescu” key does not exist at this point. Because it not exists, a new one is created and the value is instantiated using the default constructor (which for int makes the value to be 0)

STL (associative containers - map)

- ▶ The methods supported by the map container:

Method/operator

Assignment (**operator=**)

Accessing and inserting values (**operator[]** and **at**, **insert** methods)

Deletion (**erase**, **clear** methods)

Search into a map (**find** method)

Iterators (**begin**, **end**, **rbegin**, **rend**, **cbegin**, **cend**, **crbegin**, **crend** (**the last 4 from C++11**)

Informations (**size**, **empty**, **max_size**)

STL (associative containers - map)

- ▶ Accessing elements ([] operator and **at**, **find** methods):

App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    printf("Grade = %d\n", Grades["Popescu"]);
}
```

App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    printf("Grade= %d\n", Grades.at("Popescu"));
}
```

App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    printf("Grade= %d\n", Grades.find("Popescu")->second);
}
```

STL (associative containers - map)

- ▶ Accessing elements ([] operator and at, find methods):

App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    printf("Grade = %d\n", Grades["Popescu"]);
}
```

App.cpp (“Ionescu” key does not exist)

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    printf("Grade= %d\n", Grades.at("Ionescu"));
}
```

App.cpp (“Ionescu” key does not exist)

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    printf("Grade= %d\n", Grades.find("Ionescu")->second);
}
```

STL (associative containers - map)

- ▶ Checking whether an element exists in a map can be done using **find** and **count** methods:

App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    if (Grades.find("Ionescu")==Grades.cend())
        printf("Ionescu does not exist in the grades list!");
}
```

App.cpp

```
void main(void)
{
    map<const char*, int> Grades;
    Grades["Popescu"] = 10;
    if (Grades.count("Ionescu")==0)
        printf("Ionescu does not exist in the grades list!");
}
```

STL (associative containers - map)

- ▶ The elements from a map containers are stored into a red-black tree
- ▶ This represents a compromise between the access/insertion time and the amount of allocated memory for the container
- ▶ Depending on what we are interested in, a map container is not always the best solution (e.g. if the number of insertions is much smaller than the number of reads, there are containers that are more efficient)
- ▶ We do the following experiment - the same algorithm is written using a map and a simple vector and we evaluate the insertion time
- ▶ The experiment is repeated ten times for each algorithm and the execution times are measured in milliseconds

STL (associative containers - map)

- ▶ The two algorithms:

Map-1.cpp

```
void main(void)
{
    map<int, int> Test;
    for (int tr = 0; tr < 1000000; tr++)
        Test[tr] = tr;
}
```

Map-2.cpp

```
void main(void)
{
    int *Test = new int[1000000];
    for (int tr = 0; tr < 1000000; tr++)
        Test[tr] = tr;
}
```

- ▶ Even if it is obvious that App-2 is more efficient, the hash collisions must be considered. For the above case, there are no hash collisions because the keys are integers.

STL (associative containers - map)

► Results:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	19901
Map-2	0	16	0	0	16	0	15	0	16	0	6.3

► The specifications of the system:

- ❖ OS: Windows 8.1 Pro
- ❖ Compiler: cl.exe [18.00.21005.1 for x86]
- ❖ Hardware: Dell Latitude 7440 -i7 -4600U, 2.70 GHz, 8 GB RAM

STL (associative containers - multimap)

- ▶ “multimap” is a container similar to map. The difference is that a key can contain more values.
- ▶ For use “**#include <map>**”
- ▶ Accessing elements: the [] operator and the **at** method can not be used anymore.
- ▶ The declaration of multimap template:

App.cpp

```
template < class Key, class Value, class Compare = less<Key> > class multimap
{
...
}
```

STL (associative containers - multimap)

- ▶ An example of using multimap:

App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));

    multimap<const char*, int>::iterator it;
    for (it = Grades.begin(); it != Grades.end(); it++)
    {
        printf("%s [%d]\n", it->first,it->second);
    }
}
```

Output

```
Ionescu [10]
Ionescu [8]
Ionescu [7]
Popescu [9]
```

STL (associative containers - multimap)

- ▶ An example of using multimap:

App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    multimap<const char*, int>::iterator it;
    it = Grades.begin();
    printf("%s->%d\n", it->first, it->second);

    it = Grades.upper_bound(it->first);
    printf("%s->%d\n", it->first, it->second);

    it = Grades.upper_bound(it->first);
    printf("%s->%d\n", it->first, it->second);
}
```

Output

```
Ionescu->10
Popescu->9
Georgescu->8
```

STL (associative containers - multimap)

- ▶ An example of using multimap:

App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    multimap<const char*, int>::iterator it;
    for (it = Grades.begin(); it != Grades.end(); it = Grades.upper_bound(it->first))
    {
        printf("Unique key: %s\n", it->first);
    }
}
```

Output

```
Unique key: Ionescu
Unique key: Popescu
Unique key: Georgescu
```

STL (associative containers - multimap)

- ▶ An example of using multimap:

App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    multimap<const char*, int>::iterator it;
    it = Grades.begin();
    while (it != Grades.end())
    {
        pair <multimap<const char*, int>::iterator, multimap<const char*, int>::iterator> range;
        range = Grades.equal_range(it->first);
        printf("%s's grades: ", it->first);
        for (it = range.first; it != range.second; it++)
            printf("%d,", it->second);
        printf("\n");
    }
}
```

Output

```
Ionescu's grades: 10,8,7  
Popescu's grades:9,6  
Georgescu's grades:8
```

STL (associative containers - multimap)

- ▶ An example of using multimap:

App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    multimap<const char*, int>::iterator it,it2;
    it = Grades.find("Popescu");
    it2 = Grades.upper_bound(it->first);

    printf("Popescu's grades: ");
    while (it != it2)
    {
        printf("%d,", it->second);
        it++;
    }
}
```

Output

Popescu's grades: 9,6

STL (associative containers - multimap)

- ▶ An example of using multimap:

App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    multimap<const char*, int>::iterator it,it2;
    for (it = Grades.begin(); it != Grades.end(); it = Grades.upper_bound(it->first))
    {
        printf("%s has %d grades\n", it->first, Grades.count(it->first));
    }
}
```

Output

```
Ionescu has 3 grades
Popescu has 2 grades
Georgescu has 1 grades
```

STL (associative containers - multimap)

- ▶ An example of using multimap:

App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    multimap<const char*, int>::iterator it,it2;
    it = Grades.find("Ionescu");
    it++;
    Grades.erase(it); // the "8" grade from Ionescu is deleted
    for (it = Grades.begin(); it != Grades.end(); it++)
    {
        printf("%s [%d]\n", it->first, it->second);
    }
}
```

Output

```
Ionescu [10]
Ionescu [7]
Popescu [9]
Popescu [6]
Georgescu [8]
```

STL (associative containers - multimap)

- ▶ An example of using multimap:

App.cpp

```
void main(void)
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));
    Grades.insert(pair<const char*, int>("Popescu", 9));
    Grades.insert(pair<const char*, int>("Popescu", 6));
    Grades.insert(pair<const char*, int>("Georgescu", 8));

    if (Grades.find("Ionescu") != Grades.cend())
        printf("Ionescu exists!\n");

    if (Grades.find("Marin") == Grades.cend())
        printf("Marin DOES NOT exist!\n");
}
```

Output

```
Ionescu exists!
Marin DOES NOT exist!
```

STL (associative containers - multimap)

- ▶ The methods supported by the multimap container:

Method/operator
Assignment (operator=)
Insertion (insert)
Deletion (erase , clear methods)
Accessing elements (find method)
Iterators (begin , end , rbegin , rend , cbegin , cend , crbegin , crend (the last 4 from C++11))
Informations (size , empty , max_size methods)
Special methods (upper_bound and lower_bound - to access the intervals in which there are elements with the same key; equal_range - to obtain an interval for all the elements stored for a key)

STL (associative containers - set)

- ▶ “set” is a container that store unique elements
- ▶ For use “**#include <set>**”
- ▶ The declaration of **set** template:

App.cpp

```
template < class Key, class Compare = less<Key> > class set
{
...
}
```

STL (associative containers - set)

- ▶ An example of using set:

App.cpp

```
void main(void)
{
    set<int> s;
    s.insert(10);
    s.insert(20);
    s.insert(5);
    s.insert(10);
    set<int>::iterator it;

    for (it = s.begin(); it != s.end(); it++)
        printf("%d ", *it);
}
```

Output

```
5 10 20
```

STL (associative containers - set)

- ▶ An example of using set:

App.cpp

```
struct Comparator {
    bool operator() (const int& leftValue, const int& rightValue) const
    {
        return (leftValue / 20) < (rightValue / 20);
    }
};

void main(void)
{
    set<int, Comparator> s;
    s.insert(10);
    s.insert(20);
    s.insert(5);
    s.insert(10);
    set<int, Comparator>::iterator it;

    for (it = s.begin(); it != s.end(); it++)
        printf("%d ", *it);
}
```

Output

```
10 20
```

STL (associative containers - set)

- ▶ The elements from a “set” follows a specific order.
- ▶ This brings a performance penalty.

Set-1.cpp

```
void main(void)
{
    set<int> s;
    for (int tr = 0; tr < 1000000; tr++)
        s.insert(tr);
}
```

- ▶ Let's remake the previous experiment:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	19901
Map-2	0	16	0	0	16	0	15	0	16	0	6.3
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	16158

STL (associative containers - set)

- ▶ The methods supported by the set container:

Method/operator

Assignment (**operator=**)

Insertion (**insert** method)

Deletion (**erase**, **clear** methods)

Accessing elements (**find** method)

Iterators (**begin**, **end**, **rbegin**, **rend**, **cbegin**, **cend**, **crbegin**, **crend** (**the last 4 from C++11**)

Informations (**size**, **empty**, **max_size** methods)

STL (associative containers - multiset)

- ▶ “multiset” is similar with set, but duplicate elements are allowed
- ▶ For use “**#include <set>**”
- ▶ The declaration of multiset template:

App.cpp

```
template < class Key, class Compare = less<Key> > class multiset
{
...
}
```

STL (associative containers - multiset)

- ▶ An example of using multiset:

App.cpp

```
void main(void)
{
    multiset<int> s;
    s.insert(10);
    s.insert(20);
    s.insert(5);
    s.insert(10);
    multiset<int>::iterator it;

    for (it = s.begin(); it != s.end(); it++)
        printf("%d ", *it);
}
```

Output

```
5 10 10 20
```

STL (associative containers - multiset)

- ▶ The methods supported by the **multiset** container:

Method/operator
Assignment (operator=)
Insertion (insert)
Deletion (erase , clear methods)
Accessing elements (find method)
Iterators (begin , end , rbegin , rend , cbegin , cend , crbegin , crend (the last 4 from C++11))
Informations (size , empty , max_size methods)
Special methods (upper_bound and lower_bound - to access the intervals in which there are elements with the same key; equal_range - to obtain an interval that includes all the elements which have a specific key)

STL (associative containers - unordered_map)

- ▶ The elements from an **unordered_map** container are stored into a hash-table
- ▶ Introduced in C++11
- ▶ For use “**#include <unordered_map>**”
- ▶ Supports the same methods as **map** plus methods for the control of buckets.
- ▶ The declaration of **unordered_map** template:

App.cpp

```
template < class Key, class Value, class Hash, class Equal > class unordered_map
{
...
}
```

STL (associative containers - unordered_map)

- ## ► How hash tables works:

Popescu

Ionescu

Georgescu

Hash function(transforms
a string into a index)

STL (associative containers - unordered_map)

- ▶ How hash tables works:

Popescu

Ionescu

Georgescu

We consider the following hashing function:

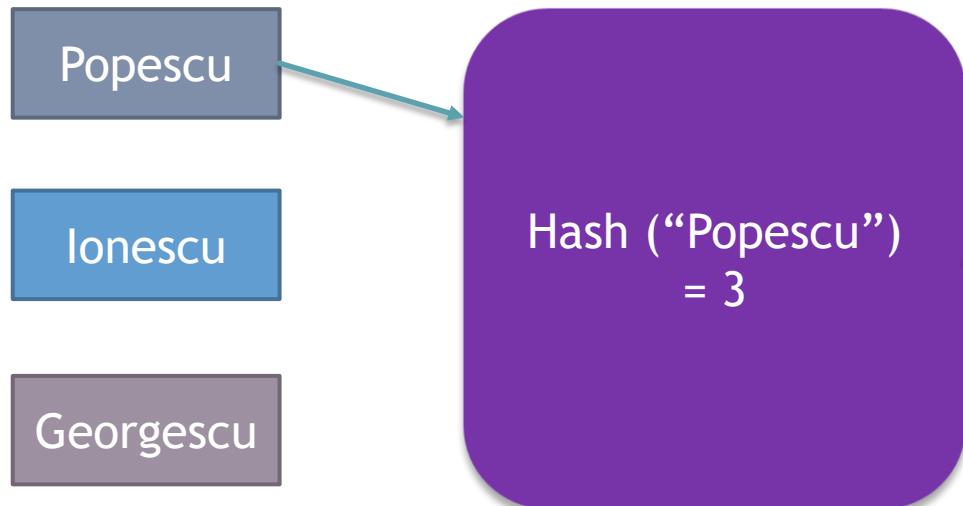
HashFunction

```
int HashFunction(const char* s)
{
    int sum = 0;
    while ((*s) != 0)
    {
        sum += (*s);
        s++;
    }
    return sum % 12;
}
```

Index	Value
0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL
9	NULL
10	NULL
11	NULL

STL (associative containers - unordered_map)

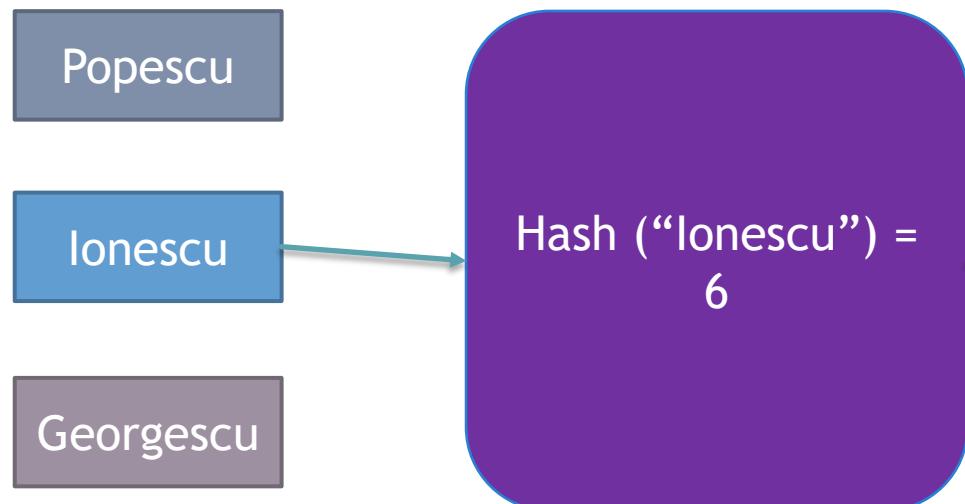
- ▶ How hash tables works:



Index	Value
0	NULL
1	NULL
2	NULL
3	Popescu
4	NULL
5	NULL
6	NULL
7	NULL
8	NULL
9	NULL
10	NULL
11	NULL

STL (associative containers - unordered_map)

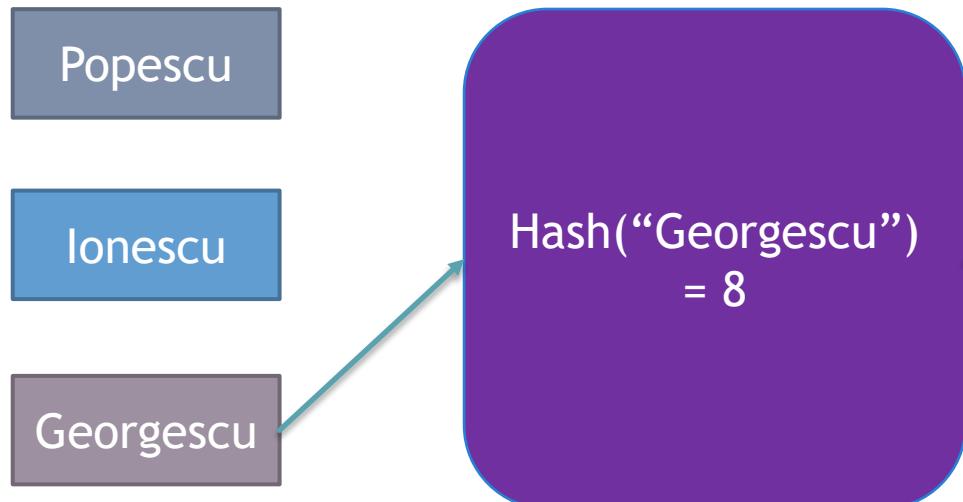
- ▶ How hash tables works:



Index	Value
0	NULL
1	NULL
2	NULL
3	Popescu
4	NULL
5	NULL
6	Ionescu
7	NULL
8	NULL
9	NULL
10	NULL
11	NULL

STL (associative containers - unordered_map)

- ▶ How hash tables works:



Index	Value
0	NULL
1	NULL
2	NULL
3	Popescu
4	NULL
5	NULL
6	Ionescu
7	NULL
8	Georgescu
9	NULL
10	NULL
11	NULL

STL (associative containers - unordered_map)

- ▶ Consider the following code:

Umap-1.cpp

```
void main(void)
{
    unordered_map<int,int> Test;
    for (int tr = 0; tr < 1000000; tr++)
        Test[tr]=tr;
}
```

- ▶ Let's remake the previous experiment:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	19901
Map-2	0	16	0	0	16	0	15	0	16	0	6.3
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	16158
Umap-1	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	15310

STL (associative containers - unordered_map)

- ▶ Consider the following code:

Umap-2.cpp

```
void main(void)
{
    unordered_map<int,int> Test;
    Test.reserve(1000000);
    for (int tr = 0; tr < 1000000; tr++)
        Test[tr]=tr;
}
```

- ▶ Let's remake the previous experiment :

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	19901
Map-2	0	16	0	0	16	0	15	0	16	0	6.3
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	16158
Umap-1	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	15310
Umap-2	9594	9703	10610	9890	10672	9922	10047	9984	9703	9938	10006

STL (associative containers - unordered_set)

- ▶ “unordered_set” is similar with the set container, but the elements are not sorted
- ▶ Introduced in C++11
- ▶ For use “`#include <unordered_set>`”
- ▶ Supports the same methods as set plus methods for the control of buckets
- ▶ The declaration of `unordered_set` template:

App.cpp

```
template < class Key, class Hash, class Equal > class unordered_set
{
...
}
```

STL (associative containers - unordered_set)

- ▶ Consider the following code:

Uset-1.cpp

```
void main(void)
{
    unordered_set<int> s;
    for (int tr = 0; tr < 1000000; tr++)
        s.insert(tr);
}
```

- ▶ Let's remake the previous experiment:

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Average
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	19901
Map-2	0	16	0	0	16	0	15	0	16	0	6.3
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	16158
Umap-1	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	15310
Umap-2	9594	9703	10610	9890	10672	9922	10047	9984	9703	9938	10006
Uset-1	12140	11625	12047	11984	12109	12078	11609	11578	11782	11672	11862

STL (associative containers - unordered_set)

- And the variant with `reserve`:

Uset-2.cpp

```
void main(void)
{
    unordered_set<int> s;
    s.reserve(1000000);
    for (int tr = 0; tr < 1000000; tr++)
        s.insert(tr);
}
```

	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	Medie
Map-1	20156	20625	20672	20453	19922	19547	19219	19516	19563	19344	19901
Map-2	0	16	0	0	16	0	15	0	16	0	6.3
Set-1	15375	15469	16407	16563	16359	15750	16094	16625	17032	15906	16158
Umap-1	14891	15984	15578	15063	15250	15234	15704	14953	15265	15186	15310
Umap-2	9594	9703	10610	9890	10672	9922	10047	9984	9703	9938	10006
Uset-1	12140	11625	12047	11984	12109	12078	11609	11578	11782	11672	11862
Uset-2	6516	6328	6875	6844	6812	6453	6453	6531	6500	6515	6582

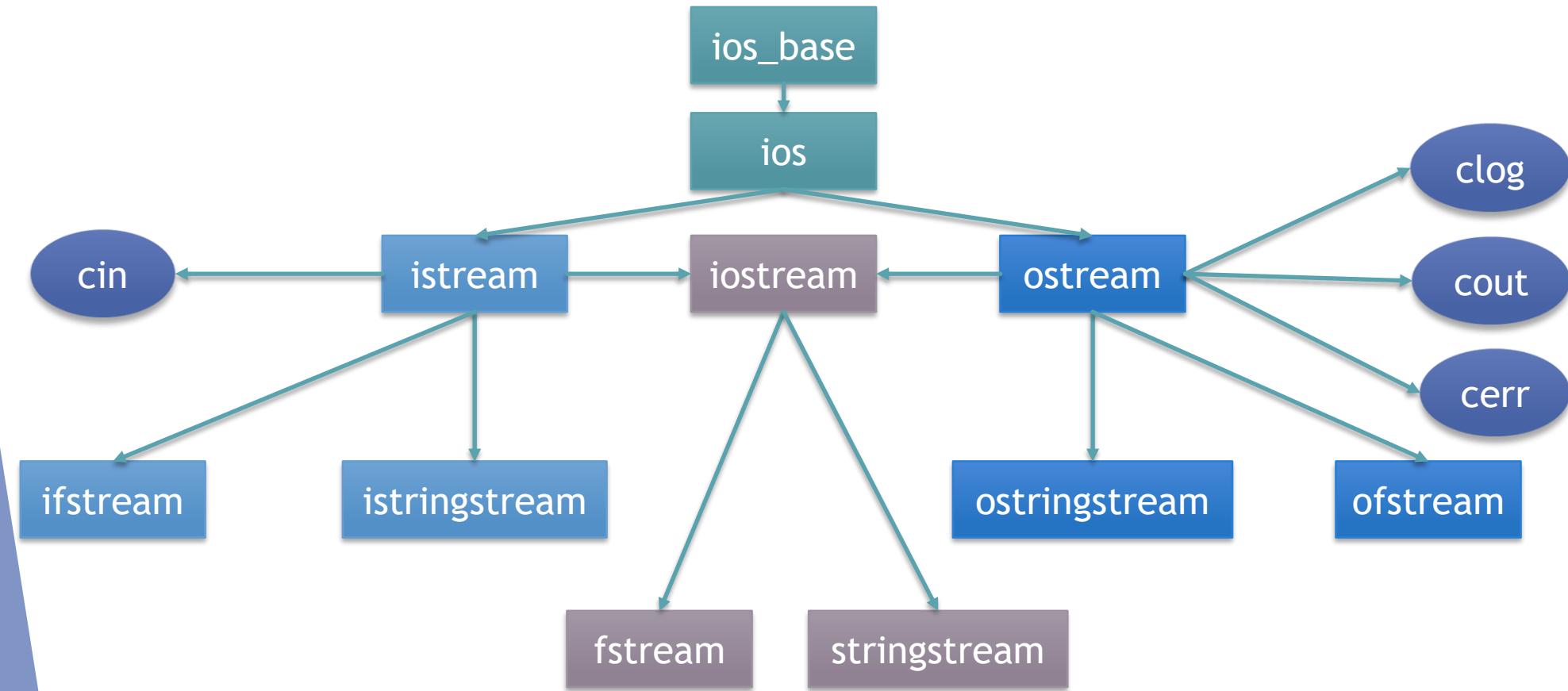
STL (associative containers)

- ▶ Besides the presented containers, two other associative containers exists:
 - ▶ `unordered_multimap`
 - ▶ `unordered_multiset`
- ▶ Similar with `multimap/multiset` (the difference is that hash tables are used, not sorted trees)
- ▶ When choosing between `map` and `unordered_map` (`set` and `unordered_set`, ...), the amount of available memory and the execution times (insertion, deletion, access of elements, ...) must be considered.

A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented diagonally, creating a sense of depth and movement.

- ▶ I/O Streams

STL (IOS)



STL(IOS)

- ▶ From the previous presented classes, the most used are **istream**, **ostream** and **iostream**.
- ▶ These classes allows the I/O access to miscellaneous streams (most well-known is the file system)
- ▶ Another use is represented by the **cin** and **cout** objects, which can be used to write to/read from the terminal
- ▶ Two operators are overloaded for these classes (**operator>>** and **operator<<**), representing the input from stream and the output to stream, respectively
- ▶ Besides the two operators, manipulators were also added to these classes (elements that can change the way of processing the data that follows them)

STL (IOS - manipulators)

manipulators	effect
endl	Adds a line terminator (“\n” , “\r\n”, etc) and flush
ends	Adds ‘\0’ (NULL)
flush	Clears the stream’s intern cache
dec	The numbers will be written in the base 10.
hex	The numbers will be written in the base 16.
oct	The numbers will be written in the base 8.
ws	Extracts and discards whitespace characters (until a non-whitespace character is found) from input
showpoint	Shows the decimal point
noshowpoint	Doesn’t show the decimal point
showpos	Add the “+” character in front of the positive numbers
noshowpos	Does not add the “+” character in front of the positive numbers

STL (IOS - manipulators)

manipulator	effect
boolalpha	Bool values will be written using “true” and ”false”
noboolalpha	Bool values will be written using 1 and 0
scientific	Scientific notation for float/double numbers
fixed	Floating-point values will be written using fixed-point notation
left	Left alignment
right	Right alignment
setfill(char)	Sets the fill character
setprecision(n)	Sets the precision for real numbers
setbase(b)	Sets the base

The background features a dark blue gradient with a subtle geometric pattern of lighter blue triangles and lines.

► Strings

STL (basic_string)

- ▶ Template that provides the most used operations over strings
- ▶ The declaration:

App.cpp

```
template <class CharacterType, class traits = char_traits<CharacterType>>
class basic_string
{
...
}
```

- ▶ The most common objects that implements this template are **string** and **wstring**

App.cpp

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

- ▶ Other objects introduced Cx11 based on this template:

App.cpp

```
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
```

STL (`basic_string`)

- ▶ “`char_traits`” is a template that provides a list of operations over the strings (not necessarily characters of type `char`) used by `basic_string` for some operations
- ▶ The main methods of `char_traits` are:

Definition	Functionality
<code>static bool eq (CType c1, CType c2)</code>	Returns <code>true</code> if <code>c1</code> is equal with <code>c2</code>
<code>static bool lt (CType c1, CType c2)</code>	Returns <code>true</code> if <code>c1</code> is lower than <code>c2</code>
<code>static size_t length (const CType* s);</code>	Returns the size of a string
<code>static void assign (CType& character, const CType& value)</code>	Assignment (<code>character = value</code>)
<code>static int compare (const CType* s1, const CType* st2, size_t n);</code>	Compares two strings; returns 1 if <code>s1 > s2</code> , 0 if <code>s1 = s2</code> and -1 if <code>s1 < s2</code>
<code>static const char_type* find (const char_type* s, size_t n, const char_type& c);</code>	Returns a pointer to the first character equal with <code>c</code> from the string <code>s</code>

STL (basic_string)

- ▶ The main methods of `char_traits` are:

Definition	Functionality
<pre>static char_type* move (char_type* dest, const char_type* src, size_t n);</pre>	Moves the content of a string between two locations
<pre>static char_type* copy(char_type* dest, const char_type* src, size_t n);</pre>	Copies the content of a string from one location to another
<pre>static int_type eof()</pre>	Returns a value for EOF (usually -1)

- ▶ “`char_traits`” has specializations for `<char>`, `<wchar>` which are using efficient functions like `memcpy`, `memmove`, etc.
- ▶ For normal use of strings, a `char_traits` object is not needed. However, it is useful for the cases where we want a particular behavior (some comparisons or assignments to be made differently - e.g. case insensitive)

STL (basic_string)

- ▶ “basic_string” supports various forms of initialization:

App.cpp

```
void main(void)
{
    string s1("Tomorrow");
    string s2(s1+ " I'll go ");
    string s3(s1, 3, 3);
    string s4("Tomorrow I'll go to my cousin.", 13);
    string s5(s4.substr(9,10));
    string s6(10, '1');

    printf("%s\n%s\n%s\n%s\n%s\n", s1.c_str(),
           s2.c_str(),
           s3.c_str(),
           s4.c_str(),
           s5.c_str(),
           s6.c_str());
}
```

Output

```
Tomorrow
Tomorrow I'll go
orr
Tomorrow I'll
I'll go to
1111111111
```

STL (basic_string)

- ▶ Methods/operators supported by the objects derived from the basic_string template:

Method/operator
Assignment (operator=)
Append (operator+= and append , push_back methods)
Characters insertion (insert method)
Access to characters (operator[] and at , front (C++11) , back (C++11) methods)
Substrings (substr method)
Replace (replace method)
Characters deletion (erase , clear , and pop_back (C++11) methods)
Search (find , rfind , find_first_of , find_last_of , find_first_not_of , find_last_not_of methods)
Comparisons (operator> , operator< , operator== , operator!= , operator>= , operator<= and compare method)
Iterators (begin , end , rbegin , rend , cbegin , cend , crbegin , crend (the last four from C++11)
Informations (size , length , empty , max_size , capacity methods)

STL (basic_string)

- ▶ An example of working with **String** objects:

App.cpp

```
void main(void)
{
    string s1;
    s1 += "Now";
    printf("Size = %d\n", s1.length());
    s1 = s1 + " " + s1;
    printf("S1 = %s\n", s1.data());
    s1.erase(2, 4);
    printf("S1 = %s\n", s1.c_str());
    s1.insert(1, "_");
    s1.insert(3, "__");
    printf("S1 = %s\n", s1.c_str());
    s1.replace(s1.begin(), s1.begin() + 2, "123456");
    printf("S1 = %s\n", s1.c_str());
}
```

Output

```
Size = 3
S1 = Now Now
S1 = Now
S1 = N_o__w
S1 = 123456o__w
```

STL (basic_string)

- ▶ Some example of using *char_traits* (a string with *ignore case*):

App.cpp

```
struct IgnoreCase : public char_traits<char> {
    static bool eq(char c1, char c2) {
        return (upper(c1)) == (upper(c2));
    }
    static bool lt(char c1, char c2) {
        return (upper(c1)) < (upper(c2));
    }
    static int compare(const char* s1, const char* s2, size_t n) {
        while (n>0)
        {
            char c1 = upper(*s1);
            char c2 = upper(*s2);
            if (c1 < c2) return -1;
            if (c1 > c2) return 1;
            s1++; s2++; n--;
        }
        return 0;
    }
};

void main(void)
{
    basic_string<char, IgnoreCase> s1("Salut");
    basic_string<char, IgnoreCase> s2("sAlUt");
    if (s1 == s2)
        printf("Siruri egale !");
}
```

The background features a large, abstract graphic on the left side composed of various shades of blue and dark blue triangles and trapezoids, creating a sense of depth and perspective.

► Initialization lists

Initialization lists

- ▶ Initialization lists work with STL as well:

App.cpp

```
using namespace std;
#include <vector>
#include <map>

struct Student
{
    const char * Name;
    int Grade;
};

void main()
{
    vector<int> v = { 1, 2, 3 };
    vector<string> s = { "POO", "C++" };
    vector<Student> st = { { "Popescu", 10 }, { "Ionescu", 9 } };
    map<const char*, int> st2 = { { "Popescu", 10 }, { "Ionescu", 9 } };
}
```

- ▶ The code complies ok and all objects are initialized properly.

Initialization lists

- ▶ Initialization lists work with STL as well:

App.cpp

```
using namespace std;
#include <vector>
#include <map>

struct Student
{
    const char * Name;
    int Grade;
};

void main()
{
    vector<int> v = { 1, 2, 3 };
    vector<string> s = { "POO", "C++" };
    vector<Student> st = { { "Popescu", 10 }, { "Ionescu", 9 } };
    map<const char*, int> st2 = { { "Popescu", 10 }, { "Ionescu", 9 } };

    map<const char*, int> st3 = { { "Popescu", 10 }, { "Popescu", 9 } };
}
```

In this case, “st3” will contain only one item (Popescu with the grade 10) the first one that was added. This is because the `<map>` template allows only one item for each key.

Initialization lists

- ▶ STL also provide a special container (called `std::initializer_list` that can be used to pass a initialization list to a function).

App.cpp

```
using namespace std;
#include <initializer_list>

int Sum(std::initializer_list<int> a)
{
    int result = 0;
    std::initializer_list<int>::iterator it;
    for (it = a.begin(); it != a.end(); it++)
        result += (*it);
    return result;
}

void main()
{
    printf("Sum = %d\n", Sum( { 1, 2, 3, 4, 5 } ));
}
```

- ▶ This code will compile and will print to the screen the value 15.

Initialization lists

- ▶ If `std::initializer_list` is used in a constructor the following expression for initializing an object can be used:

App.cpp

```
using namespace std;
#include <initializer_list>

class Data
{
    int v[10];
public:
    Data(std::initializer_list<int> a)
    {
        int index = 0;
        std::initializer_list<int>::iterator it;
        for (it = a.begin(); (it != a.end()) && (index<10); it++,index++)
            v[index] = (*it);
    }
};
void main()
{
    Data d1({ 1, 2, 3, 4, 5 });
    Data d2 = { 1, 2, 3, 4, 5 };
}
```

Initialization lists

- ▶ Let's consider the following code:

App.cpp

```
#define DO_SOMETHING(x) x  
void main() { DO_SOMETHING_printf("Test"); }
```

- ▶ This code will work ok, and the compiler will print “Test” to the screen. Now let's consider the following one:

App.cpp

```
#define DO_SOMETHING(x) x  
void main() { DO_SOMETHING( vector<int> x{1, 2, 3}; ); }
```

- ▶ In this case the code will not compile. When analyzing a MACRO the compiler does not interpret in any way the “{“ character. This means that from the compiler point of view, DO_SOMETHING macro has received 3 parameters:

- ▶ vector<int> x{1}
- ▶ 2
- ▶ 3}

Initialization lists

- ▶ In this cases the solution is to change the macro from:

App.cpp

```
#define DO_SOMETHING(x) x
```

to

App.cpp

```
#define DO_SOMETHING(...) __VA_ARGS__  
void main() { DO_SOMETHING( vector<int> x{1, 2, 3}; ); }
```

- ▶ This code will compile and run correctly.

Q & A

OOP

Gavrilut Dragos
Course 8

Summary

- ▶ Constant expressions
- ▶ For each (Range-based for loop)
- ▶ Type inference
- ▶ Structured binding (destructuring)
- ▶ Static Polymorphism (CRTP)
- ▶ Plain Old Data (POD)

A decorative graphic in the top-left corner consists of several overlapping blue triangles of varying shades, creating a sense of depth and motion.

Constant ► expressions

Constant expressions

- ▶ Constant expressions are pieces of code that the compiler can optimize by replacing them with their value that is computed before compilation phase
- ▶ Constant expression can be easily deducted for variables (especially “`const`” variables). However, in case of functions this is more difficult.
- ▶ Let's analyze the following code:

App.cpp

```
void main()
{
    int x = 10;
    int y = x;
}
```

Constant expressions

- ▶ Constant expressions are pieces of code that the compiler can optimize by replacing them with their value that is computed before compilation phase
- ▶ Constant expression can be easily deducted for variables (especially “const” variables). However, in case of functions this is more difficult.
- ▶ Let's analyze the following code:

App.cpp

```
void main()
{
    int x = 10;
    int y = x;
}
```

The diagram illustrates the compilation process. A blue arrow points from the highlighted assignment statement `int y = x;` in the C++ code to a callout bubble containing the corresponding assembly code. The assembly code consists of two parts: the initialization of `x` and the assignment of `y`. The first part is `int x = 10;` followed by the assembly instruction `mov dword ptr [x],0Ah`. The second part is `int y = x;` followed by the assembly instructions `mov eax,dword ptr [x]` and `dword ptr [y],eax`.

```
int x = 10;
mov     dword ptr [x],0Ah

int y = x;
mov     eax,dword ptr [x]
       dword ptr [y],eax
```

- ▶ When creating “y” the compiler copies the value from “x”

Constant expressions

- ▶ Constant expressions are pieces of code that the compiler can optimize by replacing them with their value that is computed before compilation phase
- ▶ Constant expression can be easily deducted for variables (especially “`const`” variables). However, in case of functions this is more difficult.
- ▶ Let's analyze the following code:

App.cpp

```
void main()
{
    const int x = 10;
    int y = x;
}
```

const int x = 10;
int y = x;

mov dword ptr [x], 0Ah
mov dword ptr [y], 0Ah

- ▶ However, adding a “`const`” declaration in front of “`x`” makes the compiler change the way it creates “`y`” (now the compiler will directly assign the value 10 (the constant value of “`x`” to “`y`”))

Constant expressions

- ▶ Constant expressions are pieces of code that the compiler can optimize by replacing them with their value that is computed before compilation phase
- ▶ Constant expression can be easily deducted for variables (especially “const” variables). However, in case of functions this is more difficult.
- ▶ Let's analyze the following code:

App.cpp

```
void main()
{
    int x = 1 + 2 + 3;
    int y = x;
}
```

int x = 1+2+3;
mov dword ptr [x], 6

- ▶ The same thing applies for expressions where the result is always a constant value. In this case, the compiler computes the value of the expression “1+2+3” and assigns that value to “x” directly.

Constant expressions

- ▶ Constant expressions are in particular important when declaring arrays:

App.cpp

```
int GetCount()
{
    return 5;
}

void main()
{
    int x[GetCount()];
}
```

error C2131: expression did not evaluate to a constant
note: failure was caused by call of undefined function or one not declared 'constexpr'
note: see usage of 'GetCount'

- ▶ This code will not compile. In reality - GetCount() returns a “const” values, but the compiler does not know if it can replace it with its value (for example GetCount() might do something else → like modifying some global variables).
- ▶ The compiler will yield an error: “expecting constant expression” when defining “x”

Constant expressions

- ▶ Constant expressions are in particular important when declaring arrays:

App.cpp

```
const int GetCount()
{
    return 5;
}

void main()
{
    int x[GetCount()];
}
```

error C2131: expression did not evaluate to a constant
note: failure was caused by call of undefined function or one not declared 'constexpr'
note: see usage of 'GetCount'

- ▶ Even if we add a “**const**” keyword at the beginning of the function, the result is still the same.
- ▶ The compiler only knows that the result can not be modified (this does not imply that the result is a constant value, and that the compiler can replace the entire call for that function with its value).

Constant expressions

- ▶ C++11 adds a new keyword: “`constexpr`” that tells the compiler that a specific expression should be considered constant.

App.cpp

```
constexpr int GetCount()
{
    return 5;
}
void main()
{
    int x[GetCount()];
}
```

- ▶ Now the code will compile.

Constant expressions

- ▶ C++11 adds a new keyword: “`constexpr`” that tells the compiler that a specific expression should be considered constant.

App.cpp

```
constexpr int GetCount()
{
    return 5;
}
void main()
{
    int x[GetCount()];
}
```

push ebp
mov ebp,esp
sub esp, 20

- ▶ As `GetCount()` will be replaced with 5, the space needed for “`x`” will be 5 integers (`sizeof(int) = 4, 4 x 5 = 20`)

Constant expressions

- ▶ Using “`constexpr`” comes with some limitations:
 - A `constexpr` function should not be void

App.cpp

```
constexpr void GetCount()
{
    //return 5;
}
void main()
{
    int x[GetCount()];
}
```

- ▶ In this case the compiler will state that it can not create an array from a void value

Constant expressions

- ▶ Using “**constexpr**” comes with some limitations:
 - A **constexpr** function should not have any local variables uninitialized

App.cpp

```
constexpr int GetCount()
{
    int x;           ← error C3615: constexpr function 'GetCount' cannot result in a constant expression
    x = 10;
    return 5;
}
void main()
{
    int x[GetCount()];
}
```

note: failure was caused by an uninitialized variable declaration
note: see usage of 'x'

- ▶ As a general rule, the compiler tries to evaluate (in the compiling phase) the result of a **constexpr** function. If a local variable is uninitialized, then there is a possibility than several execution flows may lead to a different results → thus the entire function can not be replaced with another value.

Constant expressions

- ▶ Using “`constexpr`” comes with some limitations:
 - A `constexpr` function should not have any local variables uninitialized. You can have, however multiple constant variable defined !

App.cpp

```
constexpr int GetCount()
{
    const int x = 100;
    return 5+x;
}
void main()
{
    int x[GetCount()];
}
```

App.cpp

```
constexpr int GetCount()
{
    int x = 100;
    return 5+x;
}
void main()
{
    int x[GetCount()];
}
```

error: body of `constexpr` function
‘`constexpr int GetCount()`’ not a
return-statement → **ONLY for C++11**

Local variables are allowed in a `constexpr` function starting with C++14. This code will NOT compile if a C++11 standard is used !

Constant expressions

- ▶ Using “`constexpr`” comes with some limitations:
 - If `constexpr` function has parameters, it should be called with a constant value for those parameters. Further more, the result of the evaluation should be a constant value.

App.cpp

```
constexpr int GetCount(int x)
{
    return x+x;
}
void main()
{
    int x[GetCount(10)];
}
```

- ▶ In this case the code will compile correctly (“X” will have 20 elements)
- ▶ Some compiler have some workarounds for this rule. In terms of optimization, if the exact value of a function can not be computed, inline replacement will not be possible.

Constant expressions

- ▶ Using “`constexpr`” comes with some limitations:
 - If `constexpr` function in C++11 must have only one return statement.
C++14 and above do not have this limitation anymore.

App.cpp

```
constexpr int GetCount(int x)
{
    if (x>10) return 5; else return 6;
}
void main()
{
    int x[GetCount(10)];
}
```

- ▶ This code will not compile with Cx++11 standards, but will work for C++14 standards (g++). The compiler evaluates that `GetCount(10)` can actually be replaced with 6 without changing the logic behind the construction.

Constant expressions

- ▶ Let's analyze the following code:

App.cpp

```
constexpr int SomeValue() { return 5; }
int main()
{
    int x = SomeValue();
    printf("%d", x);
}
```

- ▶ Code was compiled with VS 2017, with C++17 Standards enabled.
- ▶ While “X” is clearly 5, the compiler still generated a function (SomeValue) and calls it to get the value of “X”

```
int x = SomeValue();
call     SomeValue
mov      dword ptr [x],eax
printf   "%d", x;
mov      eax,dword ptr [x]
push    eax
push    offset string "%d"
call    printf
add     esp,8
```

Constant expressions

- ▶ Let's analyze the following code:

App.cpp

```
constexpr int SomeValue() { return 5; }
int main()
{
    constexpr int x = SomeValue();
    printf("%d", x);
}
```

```
constexpr int x = SomeValue();
mov         dword ptr [x],5
printf("%d", x);
push        5
push        offset string "%d"
call        printf
add         esp,8
```

- ▶ Code was compiled with VS 2017, with C++17 Standards enabled.
- ▶ However, declaring x as a **constexpr** will produce a different code (SomeValue is replaced automatically by its value !!!)
- ▶ This is not completely identical as declaring “x” as a **const** ! (if we would have used a **const** specifier **SomeValue** function would still be called !)

Constant expressions

- ▶ Let's analyze the following code:

Normal variable	With <code>constexpr</code>	With <code>const</code>
<pre>constexpr int SomeValue() { return 5; } int main() { int x = SomeValue(); printf("%d", x); }</pre>	<pre>constexpr int SomeValue() { return 5; } int main() { constexpr int x = SomeValue(); printf("%d", x); }</pre>	<pre>constexpr int SomeValue() { return 5; } int main() { const int x = SomeValue(); printf("%d", x); }</pre>
<pre>call SomeValue mov dword ptr [x],eax printf("%d", x); mov eax,dword ptr [x] push eax push offset string "%d" call printf add esp,8</pre>	<pre>mov dword ptr [x],5 printf("%d", x); push 5 push offset string "%d" call printf add esp,8</pre>	<pre>call SomeValue mov dword ptr [x],eax printf("%d", x); push 5 push offset string "%d" call printf add esp,8</pre>

- ▶ Code was compiled with VS 2017, with C++17 Standards enabled (debug mode)

Constant expressions

- ▶ Let's analyze the following code:

App.cpp

```
constexpr int SomeValue() { return 5; }
int main()
{
    constexpr int x = SomeValue();
    x = 100; // error C3892: 'x': you cannot assign to a variable that is const
    printf("%d", x);
}
```

- ▶ Code will NOT compile (with VS 2017, with C++17 Standards enabled).
- ▶ “x” being declared as a **constexpr** it’s similar to “x” is a **const** → you can not modify “x” value.
- ▶ However, keep in mind that **constexpr** and **const** are not identical !

Constant expressions

- ▶ Let's analyze the following code:

App.cpp (1)	App.cpp (2)	App.cpp (3)
<pre>class A { public: constexpr int x; A() : x(10) {} };</pre>	<pre>class A { public: const int x; A() : x(10) {} };</pre>	<pre>class A { public: static constexpr int x = 10; A() {} } int main() { printf("A::x = %d, sizeof(A) = %d", A::x,sizeof(A)); return 0; }</pre>

- ▶ For these pieces of code, VS 2017 was used with C++17 Standards enabled.
- ▶ The first code (#1) that uses **constexpr** will not compile ! (*A::x' cannot be declared with 'constexpr' specifier*)
- ▶ The second one (#2) that uses **const** will compile !
- ▶ The third one (#3) will compile and will print 10 and 1 to the screen. In the third code if we replace **constexpr** with **const**, the result is identical.

Constant expressions

- ▶ As a general consent, consider **constexpr** as different from **const**
- ▶ **constexpr** means that the exact value of an expression can be computed at the compile time given a set of parameters required by the expression (constant values).
- ▶ **const** means that the value returned by an expression can not be modified after its value is attributed. That is why, **const** can be apply to a class member, while a **constexpr** can not.
- ▶ **constexpr** can however be applied to class methods (including constructor, operators, etc). This technique is useful when creating another **constexpr** instance of that class.

Constant expressions

- ▶ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int x;
    A(int value): x(value*value) {}
};

int main()
{
    A a(5);
    printf("%d", a.x);
}
```

A a(5);
push 5
lea ecx,[a]
call A::A
printf("%d", a.x);
mov eax,dword ptr [a.x]
push eax
push offset string "%d"
call printf
add esp,8

- ▶ Code will compile and will generate the following assembly code.
- ▶ Code was compiled with VS 2017, with C++17 Standards enabled.

Constant expressions

- ▶ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int x;
    A(int value): x(value*value) {}
};

int main()
{
    constexpr A a(5);
    printf("%d", a.x);
}
```

error C2127: 'a': illegal initialization
of 'constexpr' entity with a non-
constant expression

- ▶ Code will NOT compile. Can not create a value (in this case an instance) from a function (in this case the constructor function) that is not **constexpr**

Constant expressions

- ▶ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int x;
    constexpr A(int value): x(value*value) {}
};

int main()
{
    constexpr A a(5);
    printf("%d", a.x);
}
```

```
A a(5);
mov    dword ptr [a.x], 25
printf("%d", a.x);
mov    eax,dword ptr [a.x]
push   eax
push   offset string "%d"
call   printf
add    esp,8
```

- ▶ Code will compile and will generate the following assembly code.
- ▶ The constructor is no longer called, but x is set to its proper value.
- ▶ Code was compiled with VS 2017, with C++17 Standards enabled.

Constant expressions

- ▶ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int x;
    constexpr A(int value): x(value*value) { printf("ctor"); }
};

int main()
{
    constexpr A a(5);
    printf("%d", a.x);
}
```

error C3615: constexpr function
'A::A' cannot result in a
constant expression

- ▶ Code will NOT compile. Using constexpr implies that you do not need to call the constructor (this can be done if there is no code that needs to be called → pretty much just assign the values to data member).
- ▶ In this case, creating an instance of type A means running a *printf("ctor")* command that can not be done if A() is constexpr.

Constant expressions

- ▶ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int x;
    constexpr A(int value) {}
};

int main()
{
    constexpr A a(5);
    printf("%d", a.x);
}
```

error C3615: constexpr function 'A::A' cannot result in a constant expression
note: failure was caused by 'constexpr' constructor not initializing member 'A::x'

- ▶ Code will NOT compile.
- ▶ The same logic applies here as well. We can not construct an instance of type A if we do not have a value for all data members in class A.

Constant expressions

- ▶ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int x, y;
    constexpr A(int value) : x(10) {}
};

int main()
{
    constexpr A a(5);
    printf("%d", a.x);
}
```

error C3615: constexpr function 'A::A' cannot result in a constant expression
note: failure was caused by 'constexpr' constructor not initializing member 'A::y'

- ▶ Code will NOT compile.
- ▶ The same logic applies here as well. We can not construct an instance of type A if we do not have a value for all data members in class A.
- ▶ In this case, A::x is instantiated, but not A::y

Constant expressions

- ▶ Let's analyze the following code:

App.cpp

```
class A
{
public:
    int x;
    A(int value) : x(value*value) { }
    constexpr int GetValue() { return 5; }
};
int main()
{
    A a(5);
    constexpr int x = a.GetValue();
    printf("%d", x);
}
```

- ▶ Code will compile.
- ▶ “x” will have the value 5.

Constant expressions

- ▶ The compiler can not always compute (during the compile time) the value of an expression. Let's consider the following code:

App.cpp

```
class A
{
public:
    const int x;
    A(int value) : x(value*value) { }
    constexpr int GetValue() { return x; }
};
int main()
{
    A a(5);
    constexpr int x = a.GetValue();
    printf("%d", x);
}
```

error C2131: expression did not evaluate to a constant
note: failure was caused by a read of a variable outside its lifetime
note: see usage of 'a'

- ▶ This code will **NOT** compile. However, *x* is a constant value, and *a.x* will always be 25 (due to the initialization from the constructor). This means that for this particular case, the compiler should have been able to assign value 25 to local variable “*x*” from main function.

Constant expressions

- ▶ The compiler can not always compute (during the compile time) the value of an expression. Let's consider the following code:

App.cpp

```
constexpr int cmmdc(int x, int y)
{
    while (x!=y)
    {
        if (x>y) x-=y; else y-=x;
    }
    return x;
}
int main()
{
    int x = cmmdc(24,18);
    int a[cmmdc(100, 5)];
    printf("x=%d, len(a)=%d\n", x, sizeof(a));
}
```

- ▶ Code will compile and run correctly.
- ▶ In this case, the simple algorithm from *cmmdc* function can be computed at the compile time by the compiler.

Constant expressions

- ▶ The compiler can not always replace expression. Let's consider:

App.cpp

```
constexpr int cmmdc(int x, int y)
{
    while (x!=y)
    {
        if (x>y) x-=y; else
        }
    return x;
}
int main()
{
    int x = cmmdc(24,18);
    int a[cmmdc(100, 5)];
    printf("x=%d, len(a)=%d\n", x, sizeof(a));
}
```

```
push    12h
push    18h
call    cmmdc
add    esp,8
mov    dword ptr [x],eax
push    14h
mov    eax,dword ptr [x]
push    eax
push    offset string "x=%d, len(a)=%d\n"
call    printf
add    esp,0Ch
```

Note that **cmmdc** function is not replaced by its value (in this case value 6) !

However the size of vector **a** is clearly known (and it is not computed dynamically)

Constant expressions

- ▶ The compiler can not always compute (during the compile time) the value of an expression. Let's consider the following code:

App.cpp

```
constexpr int cmmdc(int x, int y)
{
    while (x!=y)
    {
        if (x>y) x-=y; else y-=x;
    }
    for (int tr = 0; tr < 100; tr++)
        x += y;
    return x;
}
int main()
{
    int x = cmmdc(24,18);
    int a[cmmdc(100, 5)];
    printf("x=%d, len(a)=%d\n", x, sizeof(a));
}
```

- ▶ Code will still compile and run correctly (even if we made *cmmdc* function more complex). We have also used a temporary variable (*tr*) but with **constant** values.

Constant expressions

- ▶ The compiler can not always compute (during the compile time) the value of an expression. Let's consider the following code:

App.cpp

```
constexpr int cmmdc(int x, int y)
{
    while (x!=y)
    {
        if (x>y) x-=y; else y-=x;
    }
    for (int tr = 0; tr < 100; tr++)
        x += y;
    return x;
}
int main()
{
    int x = cmmdc(24,18);
    int a[cmmdc(100, 5)];
    printf("x=%d, len(a)=%d\n", x, sizeof(a));
}
```

- ▶ Code will still compile and run correctly (even if we made *cmmdc* function more complex). We have also used a temporary variable (*tr*) but with **constant** values.

Constant expressions

- ▶ The compiler can not always compute (during the compile time) the value of an expression. Let's consider the following code:

App.cpp

```
constexpr int cmmdc(int x, int y)
{
    while (x!=y)
    {
        if (x>y) x-=y; else y-=x;
    }
    for (int tr = 0; tr < y; tr++)
        x += y;
    return x;
}
int main()
{
    int x = cmmdc(24,18);
    int a[cmmdc(100, 5)];
    printf("x=%d, len(a)=%d\n", x, sizeof(a));
}
```

- ▶ Code will still compile and run correctly. This time we have changed a constant value with another constant value (y) !!!

Constant expressions

- ▶ The compiler can not always compute (during the compile time) the value of an expression. Let's consider the following code:

App.cpp

```
constexpr int cmmdc(int x, int y)
{
    while (x!=y)
    {
        if (x>y) x-=y; else y-=x;
    }
    for (int tr = 0; tr <x; tr++)
        x += y;
    return x;
}
int main()
{
    int x = cmmdc(24,18);
    int a[cmmdc(100, 5)];
    printf("x=%d, len(a)=%d\n",x, sizeof(a));
}
```

- ▶ Code will **NOT** compile. In this case, the compiler sees that “x” is also modified in the loop. At some point an integer overflow will be produced and the loop will stop, but this can not be in advance pre-computed.

Constant expressions

- ▶ For some cases, the compiler can pre-compute the result even for complex functions (ex: recursive functions)

App.cpp

```
constexpr int fibonacci(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main()
{
    constexpr int x = fibonacci(10);
    printf("%d", x);
}
```

mov
push
push
call
add
dword ptr [x], 55
55
offset string "%d"
printf
esp,8

- ▶ Code was compiled with VS 2017, with C++17 Standards enabled. The code compiles correctly and prints number 55 on the screen.

Constant expressions

- ▶ For some cases, the compiler can pre-compute the result even for complex functions (ex: recursive functions)

App.cpp

```
constexpr int fibonacci(int n)
{
    if (n == 0)
        return 0;
    if (n == 1)
        return 1;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main()
{
    constexpr int x = fibonacci(100).  // error C2131: expression did not evaluate to a constant
    printf("%d", x);
}
```

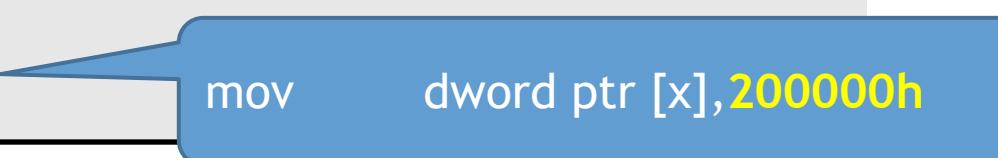
- ▶ Code was compiled with VS 2017, with C++17 Standards enabled. This code will not compile → the compiler can precompute some things but to some degree (in this case, 100 step recursion is too much).

Constant expressions

- ▶ `constexpr` can be used with literals to precompute values.

App.cpp

```
constexpr unsigned long long operator"" _Mega(unsigned long long value)
{
    return value * 1024 * 1024;
}
int main()
{
    constexpr int x = 2_Mega;
    return 0;
}
```



A red rectangular box highlights the line `constexpr int x = 2_Mega;`. A blue callout bubble originates from this box and points to the assembly instruction `mov dword ptr [x], 200000h`.

mov dword ptr [x], 200000h

- ▶ Code was compiled with VS 2017, with C++17 Standards enabled.

The background features a large, abstract graphic on the left side composed of overlapping blue triangles of varying shades of blue. It has a dark blue background with white text.

For each
► (Range-based for loop)

For each (Range-based for loop)

- ▶ C++11 standards add a new syntax for “for” statement that allows iteration within a range (similar to what a “for each” statement could do)
- ▶ The format is as follows:
for (variable_declaration : range_expression) loop_statement
- ▶ A range_expression in this context means:
 - An array of a fixed size
 - An object that has “**begin()**” and “**end()**” functions (pretty much most of the containers from STL library)
 - An initialization list
- ▶ For statement is usually used with "auto" keyword (see the next section for details).

For each (Range-based for loop)

- ▶ Examples:

App.cpp

```
void main()
{
    int x[3] = { 1, 2, 3 };
    for (int i : x)
        printf("%d", i);
}
```

- ▶ This code will print all three elements of vector x. The following code does the exact same thing but it works with a `std::vector` object.

App.cpp

```
void main()
{
    vector<int> x = { 1, 2, 3 };
    for (int i : x)
        printf("%d", i);
}
```

For each (Range-based for loop)

- ▶ For can also use initialization lists (but the code needs to include the `initializer_list` template).

App.cpp

```
#include <initializer_list>
void main()
{
    for (int i : {1, 2, 3, 4, 5})
        printf("%d", i);
}
```

- ▶ To do this, the compiler creates a `std::initializer_list` object and iterates in it.

```
mov      dword ptr [ebp-38h],1
mov      dword ptr [ebp-34h],2
mov      dword ptr [ebp-30h],3
mov      dword ptr [ebp-2Ch],4
mov      dword ptr [ebp-28h],5
lea       eax,[ebp-24h]
push    eax
lea       ecx,[ebp-38h]
push    ecx
lea       ecx,[ebp-1Ch]
call   constructor for initializer_list<int>
```

For each (Range-based for loop)

- ▶ In case of a normal array (where size is known) the compiler simulates a for loop:

App.cpp

```
void main()
{
    int x[3] = { 1, 2, 3 };
    for (int i : x)
        printf("%d", i);
}
```



```
for (index = 0; index < 3; index++)
{
    i = x[index];
    printf("%d", i);
}
```

- ▶ The same will not work if the compiler can not deduce the size of an array:

App.cpp

```
void main() {
    int *x = new int[3] {1,2,3};
    for (int i : x)
        printf("%d", i);
}
```

- ▶ This code will not compile → the compiler can not know, in advance how many elements are stored in “x” array.

For each (Range-based for loop)

- ▶ The following code will not compile as x is a matrix and not a vector. The compiler can still iterate but each element will be a “int [3]”

App.cpp

```
void main()
{
    int x[][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
    for (int i : x)
        printf("%d", i);
}
```

- ▶ To make it work, “i“ must be change to a pointer:

App.cpp

```
void main()
{
    int x[][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
    for (int* i : x)
        for (int index = 0; index < 3; index++)
            printf("%d", i[index]);
}
```

For each (Range-based for loop)

- ▶ References can also be used. In this case, the content of that loop can be modified accordingly.

App.cpp

```
void main()
{
    int x[] = { 1, 2, 3 };

    for (int &i : x)
        i *= 2;

    for (int i : x)
        printf("%d, ", i);
}
```

- ▶ The output will be 2,4,6 (as the elements from x have been modified in the first for loop).

For each (Range-based for loop)

- ▶ References can also be used. In this case, the content of that loop can be modified accordingly.

App.cpp

```
void main()
{
    const int x[] = { 1, 2, 3 };

    for (int &i : x)
        i *= 2;

    for (int i : x)
        printf("%d, ", i);
}
```

- ▶ This code will not work because x is a const vector. The compiler can't assign a “const int &” to a “int &”

For each (Range-based for loop)

- ▶ References can also be used. In this case, the content of that loop can be modified accordingly.

App.cpp

```
void main()
{
    const int x[] = { 1, 2, 3 };

    for (const int &i : x)
        i *= 2;

    for (int i : x)
        printf("%d, ", i);
}
```

- ▶ This code will still not work because even if now the compiler can pass the constant reference, it can not modify “i“ as it is a constant.

For each (Range-based for loop)

- ▶ For each can also be applied on an object. However, that object must have a **begin()** and an **end()** functions defined.

App.cpp

```
class MyVector
{
    int x[10];
public:
    MyVector() { for (int tr = 0; tr < 10; tr++) x[tr] = tr; }
};

void main()
{
    MyVector v;
    for (int i : v)
        printf("%d,",i);
}
```

- ▶ This code will not compile as no “begin()” and “end()” functions are available for class MyVector.

For each (Range-based for loop)

- ▶ For each can also be applied on an object. However, that object must have a **begin()** and an **end()** functions defined.

App.cpp

```
class MyVector
{
    int x[10];
public:
    MyVector() { for (int tr = 0; tr < 10; tr++) x[tr] = tr; }
    int* begin() { return &x[0]; }
    int* end() { return &x[10]; }
};

void main()
{
    MyVector v;
    for (int i : v)
        printf("%d,",i);
}
```

- ▶ Now the code works correctly.

For each (Range-based for loop)

- ▶ Be careful when using references. `MyVector::x` is a private field. However, it can be accessed by using references.

App.cpp

```
class MyVector
{
    int x[10];
public:
    MyVector() { for (int tr = 0; tr < 10; tr++) x[tr] = tr; }
    int* begin() { return &x[0]; }
    int* end() { return &x[10]; }
};

void main()
{
    MyVector v;
    for (int& i : v)
        i *= 2;
}
```

- ▶ The code works and `v::x` will be modified.

For each (Range-based for loop)

- ▶ The solution for this problem is to use **const** for “begin()” and “end()” functions.

App.cpp

```
class MyVector
{
    int x[10];
public:
    MyVector() { for (int tr = 0; tr < 10; tr++) x[tr] = tr; }
    const int* begin() { return &x[0]; }
    const int* end() { return &x[10]; }
};

void main()
{
    MyVector v;
    for (int& i : v)
        i *= 2;
}
```

- ▶ Now the code will not compile as “v” can iterate through constant values and “i” is not a constant (a value returned by “v” can not be assigned to “i”)

For each (Range-based for loop)

- ▶ There is also the possibility of creating your own iterator that can be returned from the `begin()` and `end()` functions:

App.cpp

```
class MyIterator {
public:
    int* p;
};
class MyVector {
...
    MyIterator begin() { MyIterator tmp; tmp.p = &x[0]; return tmp; }
    MyIterator end() {MyIterator tmp; tmp.p = &x[10]; return tmp; }
};
void main() {
    MyVector v;
    for (int i : v)
        printf("%d,",i);
}
```

- ▶ This code will not compile. For this to work the iterator must have: “`operator++`”, “`operator!=`” and “`operator*`” implementations

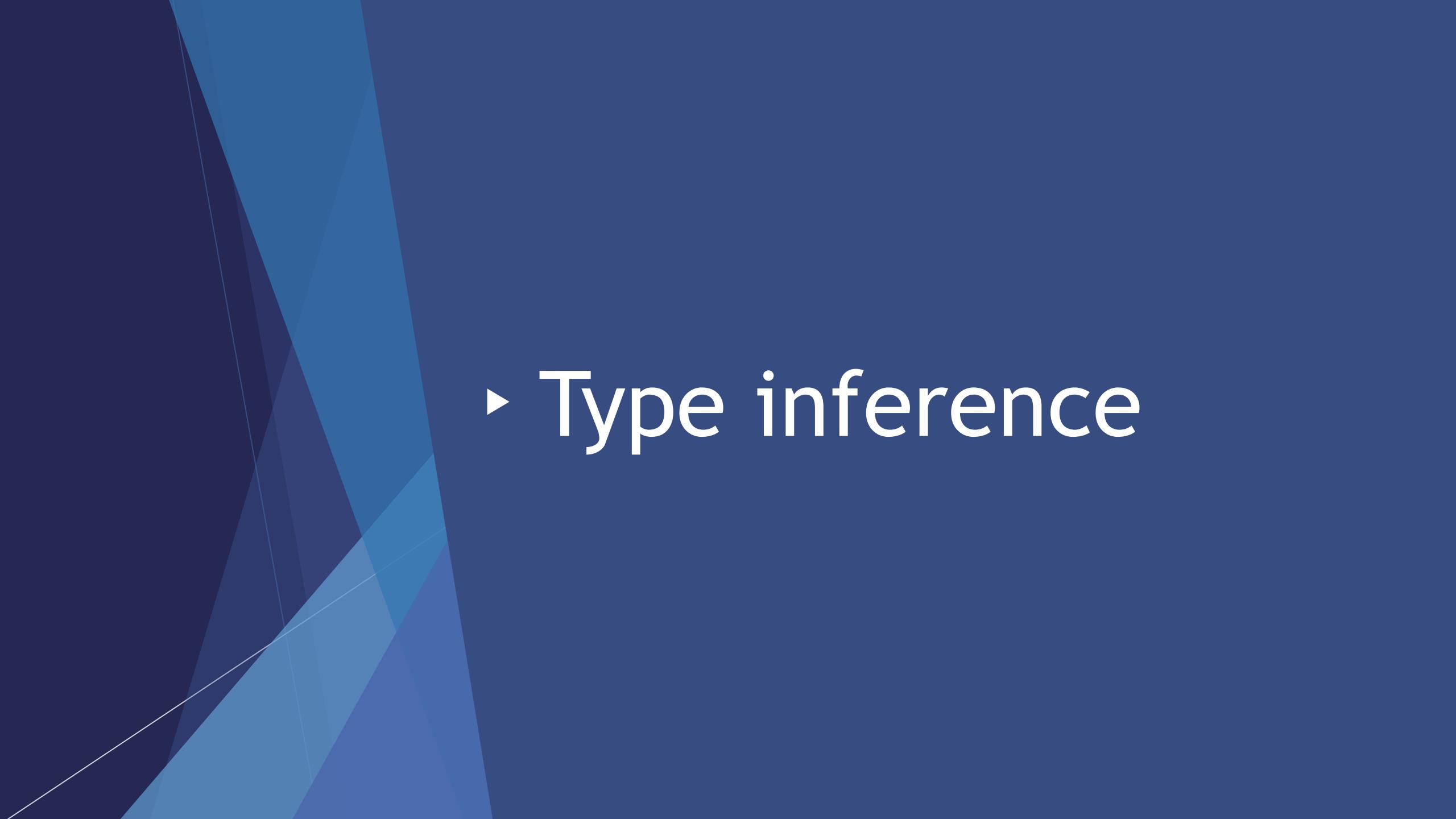
For each (Range-based for loop)

- ▶ There is also the possibility of creating your own iterator that can be returned from the `begin()` and `end()` functions:

App.cpp

```
class MyIterator {
public:
    int* p;
    MyIterator& operator++(){ p++; return *this; }
    bool operator != (MyIterator &m) { return p != m.p; }
    int operator* () { return *p; }
};
class MyVector {
...
    MyIterator begin() { MyIterator tmp; tmp.p = &x[0]; return tmp; }
    MyIterator end() {MyIterator tmp; tmp.p = &x[10]; return tmp; }
};
void main() {
    MyVector v;
    for (int i : v)
        printf("%d,",i);
}
```

- ▶ Now the code works.

- 
- A large, abstract graphic on the left side of the slide features a series of overlapping blue triangles and trapezoids. The colors range from dark navy to light cyan. The shapes are oriented diagonally, creating a sense of depth and movement.
- ▶ Type inference

"auto" keyword

- ▶ C++11 introduces a new keyword : "auto" that can be used when declaring a variable or a function
- ▶ The format is as follows:

App.cpp

```
auto <variable_name> = <value>;  
auto <function_name> ([parameters]) -> return_type {...}
```

- ▶ The compiler tries to deduce the type of the variable from its value. A similar approach exists for function and will be discussed later.

"auto" keyword

► Examples:

C++11	Translation
void main() { auto x = 10; auto y = 10.0f; auto z = 10.0; auto b = true; auto c = "test"; auto l = 100L; auto ll = 100LL; auto ui = 100U; auto ul = 100UL; auto ull = 100ULL; auto ch = 'x'; auto wch = L'x'; auto d = NULL; auto p = nullptr; }	void main() { int x = 10; float y = 10.0f; double z = 10.0; bool b = true; const char* c = "test"; long l = 100L; long long ll = 100LL; unsigned int ui = 100U; unsigned long ul = 100UL; unsigned long long ull = 100ULL; char ch = 'x'; wchar_t wch = L'x'; int d = NULL; void* p = nullptr; }

"auto" keyword

► Examples:

C++11	Translation
void main() { auto x = 10; auto y = 10.0f; auto z = 10.0; auto b = true; auto c = "test"; auto d = NULL; }	void main() { int x = 10; float y = 10.0f; double z = 10.0; bool b = true; const char* c = "test"; int d = NULL; }

NULL is defined in a way that makes the compiler translate it into int:

```
#ifndef NULL  
#ifdef __cplusplus  
    #define NULL 0  
#else  
    #define NULL ((void *)0)  
#endif  
#endif
```

"auto" keyword

- ▶ "auto" can be forced if a casting occurs during initialization.

C++11	Translation
void main() { auto x = (char*)"test"; x[0] = 0; }	void main() { char* x= (char*)"test" x[0] = 0; }

- ▶ However, the code will still crashes as “x” point to a const char* value.
- ▶ Using “new” operator also forces a cast.

C++11	Translation
void main() { auto x = new char[10] x[0] = 0; }	void main() { char* x= new char[10]; x[0] = 0; }

- ▶ In this case the code works properly (x will be a char*)

"auto" keyword

- ▶ "auto" can be used with user defined classes as well:

C++11	Translation
class Test { public: int x, y; }; void main() { auto x = new Test(); }	class Test { public: int x, y; }; void main() { Test* x = new Test(); }

- ▶ "auto" can be used with "const" keyword

C++11	Translation
void main() { const auto x = 5; }	void main() { const int x = 5; }

"auto" keyword

- ▶ "auto" can be used with another variable / expression.

C++11	Translation
void main() { auto x = 5; auto y = x; auto &z = x; auto *ptr = &x; }	void main() { int x = 5; int y = x; int &z = x; int *ptr = &x; }

- ▶ In this case because "x" is evaluated by the compiler as an "int" variable, the rest of the "auto" assignments will be considered of type "int" as well.
- ▶ In case of expressions, the resulted type of an expression is used:

C++11	Translation
void main() { auto x = 5; auto y = x * 1.5; auto z = x > 100; }	void main() { int x = 5; double y = x * 1.5; bool z = x > 100; }

"auto" keyword

- ▶ "auto" can also be used to create pointer to a function:

C++11	Translation
int sum(int x, int y, int z) { return x + y + z; } void main() { auto f = sum; auto result = f(1, 2, 3); }	int sum(int x, int y, int z) { return x + y + z; } void main() { int (*f)(int,int,int) = sum; int result = f(1, 2, 3); }

- ▶ In this case because "f" becomes a pointer to function "sum", and "result" will be of type "int" because "sum" returns an "int"
- ▶ In the end, "result" will have the value 6.

"auto" keyword

- ▶ "auto" is also useful when dealing with templates:

C++11	Translation
using namespace std; #include <vector> void main() { vector<int> v; auto it = v.begin(); }	using namespace std; #include <vector> void main() { vector<int> v; vector<int>::iterator it = v.begin(); }

- ▶ In this case, it is much easier to declare something as "auto" than to write the entire declaration as a template.

"auto" keyword

- ▶ "auto" is also useful when dealing with templates:

Cpp code

```
using namespace std;
#include <map>

void main()
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));

    multimap<const char*, int>::iterator it;
    pair <multimap<const char*, int>::iterator, multimap<const char*, int>::iterator> range;

    range = Grades.equal_range(Grades.find("Ionescu")->first);
    for (it = range.first; it != range.second; it++)
        printf("%s -> %d \n", it->first, it->second);

}
```

- ▶ In this example we two variables defined ("it" and "range").

"auto" keyword

- ▶ "auto" is also useful when dealing with templates:

Cpp code

```
using namespace std;
#include <map>

void main()
{
    multimap<const char*, int> Grades;
    Grades.insert(pair<const char*, int>("Ionescu", 10));
    Grades.insert(pair<const char*, int>("Ionescu", 8));
    Grades.insert(pair<const char*, int>("Ionescu", 7));

    auto range = Grades.equal_range(Grades.find("Ionescu")->first);

    for (auto it = range.first; it != range.second; it++)
        printf("%s -> %d \n", it->first, it->second);

}
```

- ▶ Much easier !

Type alias

- ▶ The same can option can be achieved using `typedef` or type alias.
- ▶ A type alias is functionally similar to a type def, and implies the following syntax:

using <alias_type> = <the actual type>.

Cpp code (using alias)

```
#include <map>

int main() {
    using map = std::map<const char *, int>;
    map m = { {"Popescu",10}, {"Ionescu", 9} };
}
```

Cpp code (using typedef)

```
#include <map>

int main() {
    typedef std::map<const char *, int> map;
    map m = { {"Popescu",10}, {"Ionescu", 9} };
}
```

Cpp code (using alias)

```
#include <map>
using map = std::map<const char *, int>;

int main() {
    map m = { {"Popescu",10}, {"Ionescu", 9} };
}
```

Cpp code (using typedef)

```
#include <map>
typedef std::map<const char *, int> map;

int main() {
    map m = { {"Popescu",10}, {"Ionescu", 9} };
}
```

"auto" keyword

- ▶ "auto" is usually used with for statement:

Cpp code

```
#include <vector>

void main()
{
    std::vector<int> a = { 1, 2, 3, 4, 5 };
    for (auto elem : a)
        printf("%d,", elem);
}
```

- ▶ Or as a reference:

Cpp code

```
#include <vector>

void main()
{
    std::vector<std::pair<int, char>> a = { { 1, 'A' }, { 2, 'B' }, { 3, 'D' } };
    for (auto& elem : a)
        printf("Pair: %d->%c \n", elem.first, elem.second);
}
```

" decltype " keyword

- ▶ Besides “auto” C++11 also provides a new keyword “decltype” that returns the type of an object. It is mainly used to declare a variable as of the same type of another one.

Cpp code

```
using namespace std;
#include <vector>
#include <map>

void main()
{
    vector<pair<vector<int>, map<int,const char*>>> a;
    int x;
    float y;

    decltype(x) xx;
    decltype(y) yy;
    decltype(a) aa;
}
```

- ▶ In this example “xx” has the same type as “x”, “yy” has the same type as “y” and “aa” has the same type as “a”.

" decltype " keyword

- ▶ decltype can be used with constants as well:

Cpp code

```
void main()
{
    decltype(10) x;
    decltype(10.2f) y;
    decltype(nullptr) z;
    decltype(true) b;
}
```

- ▶ In this example:
 - “x” will be of type int (because 10 is an int)
 - “y” will be of type float (because 10.2f is a float)
 - “z” will be of type void* (because nullptr is a void*)
 - “b” will be a bool (because “true” is a bool)

" decltype " keyword

- ▶ decltype can be used with arrays:

Cpp code

```
void main()
{
    int v[10];
    int w[10][20];

    decltype(v) x;
    decltype(w) y;
}
```

- ▶ In this example:
 - “x” will be of type int[10] → just like “v” is
 - “y” will be of type int[10][20] → just like “w” is

" decltype " keyword

- ▶ **decltype** can be used with elements from an array - but the result will be a reference of that type.

Cpp code

```
void main()
{
    int v[10];
    decltype(v[0]) x; ← error C2530: 'x': references must be initialized
}
```

- ▶ This code will **NOT** compile because “x” is of type “int &” and it is not initialized. For this a reference must be added to the initialization of x.

Cpp code

```
void main()
{
    int v[10];
    decltype(v[0]) x = v[0];
}
```

- ▶ Now the code compiles and “x” is a reference to the first element from “v”

" decltype " keyword

- ▶ Using references to constant strings / vectors has some limitations. The following example will not work:

Cpp code

```
void main() {  
    decltype(&"Te") x;  
}
```

- ▶ “x” will be of type “const char (*)[3]” because sizeof(“Te”) is 3 (2 characters and ‘\0’ at the end. Being a reference it needs to be initialized.

Cpp code

```
void main() {  
    decltype(&"Te") x = &"C++";  
}
```

- ▶ This code will also fail because &"C++" means “const char (*)[4]” that is not compatible with “const char (*)[3]”. To make it work, one must use the exact same number of characters as in the declaration.

Cpp code (correct code)

```
void main() {  
    decltype(&"Te") x = &"CC";  
}
```

“decltype” keyword

- ▶ The same logic applies when using a string directly as a constant in a decltype statement.

Cpp code

```
void main()
{
    decltype("Te") x = *(&"CC");
}
```

- ▶ In this case, “x” will be of type “const char[3] &”

Structured binding ► (destructuring)

Structured binding

- ▶ Starting with C++17, a new concept has been added to C++ language: *structured binding*
- ▶ This concept implies that a structure and/or array can be split down into its basic elements, and each of its elements can be assigned to a variable.
- ▶ The concept is related to what other languages (like Python) have → the possibility of returning a tuple with values (instead of one value).

App.py (Python code)

```
def GetCarSpecifics():
    return ("Toyota",180,22.5)
def main():
    car_name,max_speed,co2 = GetCarSpecifics()
```

- ▶ In C++17, *structured binding* is done using **auto** keyword in the following way:

```
auto [v1, v2, ... vn] = expression
auto& [v1, v2, ... vn] = expression
```

where v₁, v₂ ... v_n are variables that are going to be binded.

Structured binding

- ▶ Let's analyze the following code:

App.cpp

```
int main()
{
    int a[2] = { 1,2 };
    auto [x, y] = a;
    x = 10;
    return 0;
}
```

- ▶ In reality, what the compiler does is to copy the value of a[0] to “x” and the value of a[1] to “y” similar to the code bellow:

App.cpp

```
int main() {
    int a[2] = { 1,2 };
    auto x = a[0];
    auto y = a[1];
    x = 10;
    return 0;
}
```

```
int a[2] = { 1,2 };
mov     dword ptr [&a+0],1
mov     dword ptr [&a+4],2
auto [x, y] = a;
lea     eax,[a]
mov     dword ptr [temp_ptr_to_a],eax
mov     eax,4
imul   ecx,eax,0
mov     edx,dword ptr [temp_ptr_to_a]
mov     eax,dword ptr [edx+ecx]
mov     dword ptr [x],eax
mov     eax,4
shl    eax,0
mov     ecx,dword ptr [temp_ptr_to_a]
mov     edx,dword ptr [ecx+eax]
mov     dword ptr [y],edx
x = 10;
mov     eax,4
imul   ecx,eax,0
mov     dword ptr x[ecx],0Ah
```

Structured binding

- ▶ Let's analyze the following code:

App.cpp

```
int main()
{
    int a[2] = { 1,2 };
    auto[x, y] = a;
    x = 10;
    printf("a=[%d,%d] and x=%d", a[0], a[1], x);
    return 0;
}
```

- ▶ This code will compile and will print upon execution the following:
a=[1,2] and x=10

Structured binding

- ▶ Let's analyze the following code:

App.cpp

```
int main()
{
    int a[2] = { 1,2 };
    auto&[x, y] = a; ←
    x = 10;
    printf("a=[%d,%d] and x=%d", a[0], a[1], x);
    return 0;
}
```

`auto& x = a[0];
auto& y = a[1];`

- ▶ Structure binding can also be used with references “`auto&`”. In this case a reference to an object is created.
- ▶ This code will compile and will print upon execution the following:
a=[10,2] and x=10

Structured binding

- ▶ One of the most common usage of this technique is to bind the result of a function/method that returns a structure to its basic components:

App.cpp

```
struct Student
{
    const char * Name;
    int         Grade;
};

Student GetInfo()
{
    return Student{ "Popescu",10 };
}

int main()
{
    auto[name, grade] = GetInfo();
    printf("Student: %s, Grade:%d ", name, grade);
    return 0;
}
```

- ▶ This code will compile and will print upon execution the following:
Student: Popescu, Grade:10

Structured binding

- ▶ Structured bindings takes into account the access specifier.

App.cpp

```
class A
{
public:
    int x, y, z;
    A(int value) : x(value), y(value * 2), z(value * 4) {}
};

int main()
{
    A a(1);
    auto[x, y, z] = a;
    printf("x=%d, y=%d, z=%d", x, y, z);
    return 0;
}
```

- ▶ In this case, “x”, “y” and “z” are public and the binding is possible.
- ▶ This code will compile and will print upon execution the following:
x=1, y=2, z=4

Structured binding

- ▶ Structured bindings takes into account the access specifier.

App.cpp

```
class A
{
    int x, y, z;
public:
    A(int value) : x(value), y(value * 2), z(value * 4) {};
};

int main()
{
    A a(1);
    auto[x, y, z] = a; ←
    printf("x=%d, y=%d, z=%d", x, y, z);
    return 0;
}
```

```
error C3647: 'A': cannot decompose type with non-public members
note: see declaration of 'A::x'
error C2248: 'A::x': cannot access private member declared in
class 'A'
note: see declaration of 'A::x'
note: see declaration of 'A'
error C2248: 'A::y': cannot access private member declared in
class 'A'
note: see declaration of 'A::y'
note: see declaration of 'A'
error C2248: 'A::z': cannot access private member declared in
class 'A'
note: see declaration of 'A::z'
note: see declaration of 'A'
```

- ▶ In this case, “x”, “y” and “z” are private and the binding is NOT possible.
- ▶ This code will not compile !

Structured binding

- ▶ You can not bind only some data members - you have to bind all of them.

App.cpp

```
class A
{
public:
    int x, y, z;
    A(int value) : x(value), y(value * 2), z(value * 4) {};
};

int main()
{
    A a(1);
    auto[x, y] = a; ←
    printf("x=%d, y=%d", x, y);
    return 0;
}
```

error C3448: the number of identifiers must match
the number of array elements or members in a
structured binding declaration

- ▶ In this case, “x”, “y” and “z” are public and the binding is possible, but as “**auto[x,y]**” only tries to bind two parameters (and class A has 3), the code will not compile.

Structured binding

- ▶ Structured bindings copies vectors/arrays as well.

App.cpp

```
class A
{
public:
    int x[2], y;
    A(int value) : x{ value,value*2 }, y(value * 3) {};
};

int main()
{
    A a(1);
    auto[x,y] = a;
    a.x[0] = 10;
    printf("x=%d,%d, y=%d, a={x=[%d,%d], y=%d}", x[0],x[1], y,a.x[0],a.x[1],a.y);
    return 0;
}
```

- ▶ In this case local variable “x” is an array with two elements that copied the content from A::x.
- ▶ This code will compile and will print upon execution the following:
x=1,2, y=3, a={x=[10,2], y=3}

Structured binding

- ▶ Structured bindings are often used with for-each loops and STL, especially for maps where access to both components (key and value) can be obtained simultaneously.

App.cpp

```
#include <map>
using namespace std;

int main()
{
    map<const char *, int> Grades = { {"Popescu",10}, {"Ionescu",9} };
    for (auto[name, grade] : Grades)
        printf("Name:%s, Grade:%d\n", name, grade);
    return 0;
}
```

- ▶ This code will compile and will print upon execution the following:
Name:Popescu, Grade:10
Name:Ionescu, Grade:9

Structured binding

- ▶ STL also has two functions: `std::make_tuple` and `std::tie` that can be used to create a similar functionality (for C++ compilers prior to C++17 standard).

App.cpp

```
struct Student
{
    const char * Name;
    int Grade;
    auto GetParams() {
        return std::make_tuple(Name, Grade);
    }
};
int main()
{
    Student s = { "Popescu",10 };
    const char * name;
    int grade;
    std::tie(name, grade) = s.GetParams();
    printf("Name:%s, Grade:%d\n", name, grade);
    return 0;
}
```

This method is however not that effective as it implies creating your own local variables and a translation function within the class (something that can return a `std::tuple`)

- ▶ This code will compile and will print upon execution the following:
Name:Popescu, Grade:10

A decorative graphic in the top-left corner consists of several overlapping blue triangles of varying shades, creating a geometric pattern.

Static Polymorphism ► (CRTP)

Static Polymorphism

- ▶ Static polymorphism (also called Curiously Recurring Template Pattern or CRTP) is a technique that takes advantage that a template is not instantiated (constructed) when it is written - but when its instance is actually created. This allows one to use some functions in a template that are not available at the time the template was written.

Example

```
template <typename T>
class Base { ... };

class Derived: public Base<Derived> { ...};
```

- ▶ In this case - we can create a class (Derived) that has as a base class a template that can further be used with the exact class that we are creating (the Derived class).

Static Polymorphism

- ▶ Let's see an example:

App.cpp

```
template <typename T>
struct Car {
    void PrintName() {
        printf("%s\n", (static_cast<T*>(this))->GetName());
    }
};

struct Toyota : public Car<Toyota> {
    const char * GetName() { return "Toyota"; }
};
struct Dacia : public Car<Dacia> {
    const char * GetName() { return "Dacia"; }
};

int main() {
    Toyota t;
    Dacia d;
    t.PrintName();
    d.PrintName();
    return 1;
}
```

This code compiles and upon execution will print on the screen: **Toyota** and then **Dacia**

Static Polymorphism

- ▶ Let's see an example:

App.cpp

```
template <typename T>
struct Car {
    void PrintName() {
        printf("%s\n", static_cast<T*>(this)->GetName());
    }
};

struct Toyota : public Car<Toyota> {
    const char * GetName() { return "Toyota"; }
};
struct Dacia : public Car<Dacia> {
    const char * GetName() { return "Dacia"; }
};

int main() {
    Toyota t;
    Dacia d;
    t.PrintName();
    d.PrintName();
    return 1;
}
```

The main *trick* here is that `static_cast<T*>` Template Car assumes that the object of type T has a method called GetName that returns a `const char *`

Static Polymorphism

- ▶ Let's see an example:

App.cpp

```
template <typename T>
struct Car {
    void PrintName() {
        printf("%s\n", (static_cast<T*>(this))->GetName());
    }
};

struct Toyota : public Car<Toyota> {
    const char * GetName() { return "Toyota"; }
};
struct Dacia : public Car<Dacia> {
    const char * GetName() { return "Dacia"; }
};

int main() {
    Toyota t;
    Dacia d;
    t.PrintName();
    d.PrintName();
    return 1;
}
```

Because *Car* is a template, it is evaluated when it is used. This means, that the code from the method *Car::PrintName* will only be evaluated when creating the class *Toyota*. As this class has a method *GetName*, everything will work as expected.

Static Polymorphism

- ▶ The same logic can be used for data members.

App.cpp

```
template <typename T>
struct Car {
    void PrintName() {
        printf("%s\n", (static_cast<T*>(this))->Name);
    }
};

struct Toyota : public Car<Toyota> { const char * Name = "Toyota"; };
struct Dacia : public Car<Dacia> { const char * Name = "Dacia"; };

int main() {
    Toyota t;
    Dacia d;
    t.PrintName();
    d.PrintName();
    return 1;
}
```

- ▶ In this case, it is **expected** that class associated with type **T** have a data member of type **const char *** named **Name**.
- ▶ The code compiles correctly and upon execution will print **Toyota** and then **Dacia**

Static Polymorphism

- ▶ It works in a similar way for static data members (however in this case casting *this pointer* is not required (we can use $T::$ to refer to static members of type T))

App.cpp

```
template <typename T>
struct Car {
    static void PrintName() {
        printf("%s\n", T::Name);
    }
};

struct Toyota : public Car<Toyota> { static const char * Name; };
struct Dacia : public Car<Dacia> { static const char * Name; };

const char * Toyota::Name = "Toyota";
const char * Dacia::Name = "Dacia";

int main() {
    Toyota::PrintName();
    Dacia::PrintName();
    return 1;
}
```

- ▶ The code compiles correctly and upon execution will print **Toyota** and then **Dacia**

Static Polymorphism

Polymorphic chaining

- ▶ Another interesting thing that can be achieved in this way is called *polymorphic chaining*.
- ▶ It implies that the base class returns a value that is a self reference to the template type !

Example

```
template <typename T>
class Base
{
    T& SomeMethod() {
        ...
        return static_cast<T&>(*this);
    }
};
class Derived: public Base<Derived> { ...};
```

- ▶ In this case, we make sure that the method *SomeMethod* returns a reference to the type T (template type)

Static Polymorphism

Polymorphic chaining

- ▶ Let's analyze this example:

App.cpp

```
#include <iostream>

template <typename T>
struct Number {
    T& Inc() { static_cast<T*>(this)->Value += 1; return static_cast<T&>(*this); }
    T& Dec() { static_cast<T*>(this)->Value -= 1; return static_cast<T&>(*this); }
    T& Print() { std::cout << static_cast<T*>(this)->Value << " "; return static_cast<T&>(*this); }
};

struct Integer : public Number<Integer> { int Value; };
struct Float : public Number<Float> { float Value; };

int main() {
    Integer i; i.Value = 10;
    i.Inc().Print().Dec().Inc().Inc().Print();
    Float f; f.Value = 1.5;
    f.Inc().Print().Dec().Inc().Inc().Print();
}
```

- ▶ The code will print “11 12 2.5 3.5”. What happens is the *i.Inc()* will not return a reference to type *Number<>*, but to type *Integer*, thus allowing the chaining to continue.

Static Polymorphism

Barton-Nackman trick

- ▶ Barton-Nackman trick implies using CRTP and an inner friend function definition to move the friend function from the base class to de derived one.
- ▶ This is in particular useful to automatically overload relationship operators.

Example

```
template <typename T>
struct Comparable {
    friend bool operator== (const T& obj1, const T& obj2) { return obj1.CompareWith(obj2) == 0; }
    friend bool operator< (const T& obj1, const T& obj2) { return obj1.CompareWith(obj2) < 0; }
};
struct Integer : public Comparable<Integer> {
    int Value;
    Integer(int v): Value(v) {}
    int CompareWith(const Integer& obj) const {
        if (Value < obj.Value) return -1;
        if (Value > obj.Value) return 1;
        return 0;
    }
};
void main() {
    Integer i1(10);
    Integer i2(20);
    if (i1 < i2) printf("i1 is smaller than i2");
}
```

In this case, *Integer* class has both **operator==** and **operator<** defined and as such a syntax like (**if (i1 < i2)**) will compile.

Static Polymorphism

- ▶ **Static polymorphism** has the following advantages:
 - We no longer need virtual table, dynamic types, etc to perform polymorphism.
 - Since the linkage is static and not real-time, the performance is much better than with the usage of virtual function (no vptr call)
- ▶ **Static polymorphism** has the following pitfalls:
 - In reality, there is not a common root like in case of inheritance (if class A is derived from Base<A> and class B is derived from Base we CAN NOT say that they are both derived out of Base !!!)
 - This means that casting to the base class is not possible → so we can create a pointer of type Base that has multiple elements (one that points to an object A, another one that points to an object B)

Static Polymorphism

- ▶ Differences between static polymorphism and dynamic polymorphism.

Static polymorphism

```
template <typename T> struct Base { };

class A : public Base<A> { };
class B : public Base<B> { };

int main() {
    A a;
    B b;
    Base * base[2];
    base[0] = &a;
    base[1] = &b;
    return 0;
}
```

Dynamic polymorphism

```
struct Base { };

class A: public Base { };
class B: public Base { };

int main() {
    A a;
    B b;
    Base * base[2];
    base[0] = &a;
    base[1] = &b;
    return 0;
}
```

error C2955: 'Base': use of class template
requires template argument list
error C2440: '=': cannot convert from 'A *' to
'Base *'
error C2440: '=': cannot convert from 'B *' to
'Base *'

Code will compile and run
as expected.

A decorative graphic in the top-left corner consists of several overlapping blue triangles of varying shades, creating a sense of depth and perspective.

Plain Old Data ► (POD)

POD

- ▶ Plain old data (POD) means a type that has a C-like memory layout.
- ▶ In many cases a class / struct in C/C++ has other fields such as virtual functions or indexes for members from a virtually derived class
- ▶ This means that a compiler has some problems when copying such objects.
- ▶ To ease this process, a type of data can be:
 - Trivial
 - Standard layout
- ▶ POD data is important for initialization lists.

POD

- ▶ Trivial types means that:
 - Has a default constructor (that is not provided by the programmer)
 - Has a default destructor (that is not provided by the programmer)
 - Has a default copy - constructor (that is not provided by the programmer)
 - Has a assignment operator (=) (that is not provided by the programmer)
 - It has no virtual functions
 - It has no base class that has a user provided (specific) constructor / destructor / copy-constructor or assignment operator
 - It has no members that have a user provided (specific) constructor / destructor / copy-constructor or assignment operator
 - It has not data member that is a reference value

Trivial types can be copied using `memcpy` from an object to a memory buffer or an array. The compiler can change the order of data members

Trivial types can have different access modifier for their members.

POD

- ▶ STL provides a function to check if a type is trivial or not : `std::is_trivial`

App.cpp

```
#include <type_traits>
#include <iostream>

class TypeA {
    int x, y;
};

class TypeB {
    int x, y;
public:
    TypeB(int value) { x = y = value; }
};

void main()
{
    std::cout << std::boolalpha << std::is_trivial<TypeA>::value << std::endl;
    std::cout << std::boolalpha << std::is_trivial<TypeB>::value << std::endl;
}
```

- ▶ This code will print “true” for TypeA and “false” for TypeB (because it has a user defined constructor)

POD

- ▶ STL provides a function to check if a type is trivial or not : `std::is_trivial`

App.cpp

```
#include <type_traits>
#include <iostream>

class TypeC
{
    int x, y;
public:
    int z;
    const char* ptr;
    void Set(int _x, int _y, int _z) { x = _x; y = _y; z = _z; }
};

void main()
{
    cout << std::boolalpha << std::is_trivial<TypeC>::value << endl;
}
```

- ▶ This code will print “true” for TypeC

POD

- ▶ STL provides a function to check if a type is trivial or not : `std::is_trivial`

App.cpp

```
#include <type_traits>
#include <iostream>

class TypeD
{
    int x, y;
public:
    int z = 10;
    const char* ptr;
    void Set(int _x, int _y, int _z) { x = _x; y = _y; z = _z; }
};

void main()
{
    cout << std::boolalpha << std::is_trivial<TypeD>::value << endl;
}
```

- ▶ This code will print “**false**” for TypeD (because it is using a initialization function → it will be discuss in the Initialization list chapter)

POD

- ▶ Standard layout types means that:
 - A type that has no virtual functions or virtual inheritance
 - It has not data member that is a reference value
 - All data members (except static ones) have the same access control
 - All data members have a standard layout
 - The diamond problems is not applied for the type (it has no two sub-classes that are derived from the same class).
 - The first member (non-static) of the class, is not of the same type as one of the base classes (this is a condition related to empty base optimization problem)
- ▶ STL also provides a function that can be used to see if a type has a standard layout or not: `std::is_standard_layout`
- ▶ A **class** or a **struct** that is **trivial** and has a **standard layout** is a POD (plain old data). Scalar types (int,char, etc) are also considered to be POD.

POD

Empty base optimization

- ▶ Let's consider the following code:

App.cpp

```
class Base {};\n\nclass Derived : Base {\n    int x;\n};\n\nvoid main()\n{\n    printf("SizeOf(Base) = %d\n", sizeof(Base));\n    printf("SizeOf(Derived) = %d\n", sizeof(Derived));\n}
```

- ▶ The code compiles and the result is 1 byte for Base class and 4 bytes for Derived class.
- ▶ Base class has 1 byte because it is empty (it has no fields).

POD

Empty base optimization

- ▶ Let's consider the following code:

App.cpp

```
class Base {};\n\nclass Derived : Base {\n    Base b;\n    int x;\n};\n\nvoid main()\n{\n    printf("SizeOf(Base) = %d\n", sizeof(Base));\n    printf("SizeOf(Derived) = %d\n", sizeof(Derived));\n}
```

- ▶ The code compiles but now the size of Derived class is 8. Normally as Base class is empty, the result should have been 4, but because the first member of the class is of type Base it forces an alignment.
- ▶ This form of layout is considered to be non-standard.

POD

- ▶ Examples:

App.cpp

```
using namespace std;
#include <type_traits>
#include <iostream>

class MyType
{
    int x, y;
public:
    int z;
    const char* ptr;
    void Set(int _x, int _y, int _z) { x = _x; y = _y; z = _z; }
};

void main()
{
    cout << boolalpha << is_trivial<MyType>::value << "," << is_standard_layout<MyType>::value;
}
```

- ▶ This code will print “`true, false`” for `MyType`. It is not a standard layout because it has both public and private members.

POD

- ▶ Examples:

App.cpp

```
using namespace std;
#include <type_traits>
#include <iostream>

class MyType
{
public:
    int x, y;
    int z;
    const char* ptr;
    void Set(int _x, int _y, int _z) { x = _x; y = _y; z = _z; }
};

void main()
{
    cout << boolalpha << is_trivial<MyType>::value << "," << is_standard_layout<MyType>::value;
}
```

- ▶ This code will print “true,true” for MyType.

POD

- ▶ Examples:

App.cpp

```
using namespace std;
#include <type_traits>
#include <iostream>

class MyType
{
public:
    int x, y;
    int& z;
    const char* ptr;
    void Set(int _x, int _y, int _z) { x = _x; y = _y; z = _z; }
};

void main()
{
    cout << boolalpha << is_trivial<MyType>::value << "," << is_standard_layout<MyType>::value;
}
```

- ▶ This code will print “**false,false**” for MyType. It is not trivial nor standard layout because it has a field that is of a reference value.

POD

- ▶ Examples:

App.cpp

```
using namespace std;
#include <type_traits>
#include <iostream>

class Base
{
    int xx;
};

class MyType: Base
{
public:
    int x, y;
    MyType() : x(0), y(1) {}
};

void main()
    cout << boolalpha << is_trivial<MyType>::value << "," << is_standard_layout<MyType>::value;
}
```

- ▶ This code will print “**false, false**” for MyType. It is not a standard layout because MyType has a private member “Base::xx” . It is not trivial because the constructor from class MyType is defined.

Q & A

OOP

Gavrilut Dragos
Course 9

Summary

- ▶ Lambda expressions
- ▶ Modeling lambda expression behavior
- ▶ Implicit conversion to a pointer to a function
- ▶ Lambdas and STL
- ▶ Using lambda with templates (Generic lambdas)
- ▶ Mutable capture
- ▶ Initialized lambda capture
- ▶ New feature in C++17 and beyond

A decorative graphic in the top-left corner consists of several overlapping blue triangles of varying shades, creating a geometric pattern.

Lambda ► expressions

Lambda expressions

- ▶ Lets assume that we have the following structure:

App.cpp

```
struct Student
{
    const char * Name;
    int Grade;
    int Group;
    int Age;
};

Student students[] = {
    { "Popescu", 10, 5, 19 },
    { "Ionescu", 8, 3, 20 },
    { "Georgescu", 9, 4, 21 },
};
```

- ▶ ‘students’ is a global list of students.

Lambda expressions

- ▶ Now let's assume that we want to sort the entire student list in different ways (alphabetically, after the 'Age' field, based on the 'Grade' field, etc).
- ▶ The easiest way would be to create a sort algorithm that uses a pointer to a function used to compare to Student structures.

App.cpp

```
void Sort(Student *list, int count, bool(*BiggerFnc)(Student &s1, Student &s2)) {  
    bool sorted;  
    do {  
        sorted = true;  
        for (int tr = 0; tr < count - 1; tr++)  
        {  
            if (BiggerFnc(list[tr], list[tr + 1]))  
            {  
                Student aux = list[tr];  
                list[tr] = list[tr + 1];  
                list[tr + 1] = aux;  
                sorted = false;  
            }  
        }  
    } while (!sorted);  
}
```

Lambda expressions

- ▶ The main function will look as follows:

App.cpp

```
struct Student { ... }
Student students[3] = { ... }
void Sort(Student *list, int count, bool(*BiggerFnc)(Student &s1, Student &s2) ) { ... }

bool ByGrade (Student &s1, Student &s2)
{
    return s1.Grade > s2.Grade;
}
bool ByAge (Student &s1, Student &s2)
{
    return s1.Age > s2.Age;
}
bool ByName (Student &s1, Student &s2)
{
    return strcmp(s1.Name, s2.Name) > 0;
}

int main() {
    Sort(students, 3, ByGrade );
    Sort(students, 3, ByAge );
    Sort(students, 3, ByName );
}
```

Lambda expressions

- ▶ Instead of creating functions for each comparation, wouldn't it be easier if we could just write the comparation whenever we call the Sort function.

App.cpp

```
struct Student { ... }
Student students[3] = { ... }

void Sort(Student *list, int count, bool(*BiggerFnc)(Student &s1, Student &s2) ) { ... }

int main() {
    Sort(students, 3, [] (Student &s1, Student &s2) { return s1.Grade > s2.Grade; } );
}
```

- ▶ The code compiles and works as expected, the list of students being sorted based on the 'Grade' field.

Lambda expressions

- ▶ A lambda expression also has the ability to capture a state (variables) via copy or reference and use it in a function. This behavior is also called a closure.
- ▶ Let's analyze the following python code (chosen for simplicity):

App.py

```
def Multiply(factor):
    return lambda x: x * factor

def main():
    f1 = Multiply(5)
    f2 = Multiply(7)
    print f1(3),f2(3)

main()
```

Lambda expressions

- ▶ A lambda expression also has the ability to capture a state (variables) via copy or reference and use it in a function. This behavior is also called a closure.
- ▶ Let's analyze the following python code (chosen for simplicity):

App.py

```
def Multiply(factor):  
    return Lambda x: x * factor
```

```
def main():  
    f1 = Multiply(5)  
    f2 = Multiply(7)  
    print f1(3),f2(3)
```

```
main()
```

Multiply function actually returns another function (that has a prototype like the following):
def Function(x): returns x * factor

Lambda expressions

- ▶ A lambda expression also has the ability to capture a state (variables) via copy or reference and use it in a function. This behavior is also called a closure.
- ▶ Let's analyze the following python code (chosen for simplicity):

App.py

```
def Multiply(factor):
    return lambda x: x * factor

def main():
    f1 = Multiply(5)
    f2 = Multiply(7)
    print f1(3),f2(3)

main()
```

f1 will be a function that has the following prototype:
def Function(x): returns x * 5
The *factor* parameter from the Multiply function is used in this function (as the value 5).

Lambda expressions

- ▶ A lambda expression also has the ability to capture a state (variables) via copy or reference and use it in a function. This behavior is also called a closure.
- ▶ Let's analyze the following python code (chosen for simplicity):

App.py

```
def Multiply(factor):
    return lambda x: x * factor

def main():
    f1 = Multiply(5)
    f2 = Multiply(7)
    print f1(3), f2(3)

main()
```

f1 will be a function that has the following prototype:
def Function(x): returns x * 7
The *factor* parameter from the Multiply function is used in this function (as the value 7).

Lambda expressions

- ▶ A lambda expression also has the ability to capture a state (variables) via copy or reference and use it in a function. This behavior is also called a closure.
- ▶ Let's analyze the following python code (chosen for simplicity):

App.py

```
def Multiply(factor):
    return lambda x: x * factor

def main():
    f1 = Multiply(5)
    f2 = Multiply(7)
    print f1(3),f2(3)

main()
```

The code runs and prints 15 (3×5) and 21 (3×7)

Lambda expressions

- ▶ A lambda expression is defined in the following ways:

Lambda expressions

```
[captures] (parameters) -> return type { body }  
[captures] (parameters) { body }  
[captures] { body }
```

- ▶ Examples:

Lambda expressions

```
[x,y] (int a, float b) -> bool { return (a*b)<(x+y); }  
[x] (int *xx) { return *xx+x; } // the return type is deduced to be int from the body  
[a,b] { return a+b; }
```

Lambda expressions

- ▶ The *capture* component from the lambda expression can be:

Capture	Observation
[]	Captures nothing
[a,b]	Captures variables “a” and “b” by making a copy of them
[&a,&b]	Captures variables “a” and “b” using their reference
[this]	Captures current object
[&]	Captures all variables <u>used</u> in the body of the lambda by using their reference. If “this” is available it is also captured (by reference)
[=]	Captures all variables <u>used</u> in the body of the lambda by making a copy of them. If “this” is available it is also captured (by reference).
[=,&a]	Captures all variables <u>used</u> in the body of the lambda by making a copy of them, except for “a” that is captured by reference.
[&,a]	Captures all variables <u>used</u> in the body of the lambda by using their reference, except for “a” that is captured by making a copy.

Lambda expressions

- ▶ Example:

App.cpp

```
int main()
{
    auto f = [] (int x, int y) { return x + y; };
    int x = f(10, 20);
    printf("X = %d", x);
    return 0;
}
```

- ▶ This example compiles and prints value 30 on the screen.

Lambda expressions

- ▶ Example :

App.cpp

```
int main()
{
    int value = 100;
    auto f = [value] (int x, int y) { return x + y + value; };
    printf("%d\n", f(10, 20));
    value = 200;
    printf("%d\n", f(10, 20));
    return 0;
}
```

- ▶ This example compiles and prints value 130 twice on the screen.

Lambda expressions

- ▶ Example :

App.cpp

```
int main()
{
    int value = 100;
    auto f = [value] (int x, int y) { return x + y + value; };
    printf("%d\n", f(10, 20));
    value = 200;
    printf("%d\n", f(10, 20));
    return 0;
}
```

Capture local variable
‘value’ by making a copy

- ▶ This example compiles and prints value 130 twice on the screen.
- ▶ Local variable ‘value’ is capture by making a copy of its value. This means that even if we change its value the result from the lambda function will be the same.

Lambda expressions

- ▶ Example :

App.cpp

```
int main()
{
    int value = 100;
    auto f = [&value] (int x, int y) { return x + y + value; };
    printf("%d\n", f(10, 20));
    value = 200;
    printf("%d\n", f(10, 20));
    return 0;
}
```

Capture local variable
‘value’ by reference

- ▶ Now the code runs and prints 130 and them 230 on the screen.

Lambda expressions

- ▶ Example :

App.cpp

```
int main()
{
    int value = 100;
    auto f = [&value] (int x, int y) -> char { return x + y + value; };
    printf("%d\n", f(10, 20));
    value = 200;
    printf("%d\n", f(10, 20));
    return 0;
}
```

Lambda expression type

- ▶ In this case we set up the type of the lambda expression. If not set it is deduced from the return type of the lambda expression.
- ▶ The result will be -126 and -30 (char representation for int values 130 and 230)

Lambda expressions

- ▶ Example :

App.cpp

```
void MyFunction(int aa)
{
    int a, b, c;
    a = b = c = 10;
    auto f = [=] (int x, int y) { return x + y + a + b + c + aa; };
    printf("%d\n", f(10, 20));
}
int main()
{
    MyFunction(1000);
    return 0;
}
```

- ▶ The code compiles and prints the value 1060.
- ▶ All local variables and parameters from the function “MyFunction” are captured.

Lambda expressions

- ▶ Example :

App.cpp

```
void MyFunction(int aa)
{
    int a, b, c;
    a = b = c = 10;
    auto f = [=] (int x, int y) { return x + y + a + b + c + aa; };
    printf("%d\n", f(10, 20));
}
int main()
{
    MyFunction(1000);
    return 0;
}
```

Capture all local
variables and
parameters by making a
copy of them

- ▶ The code compiles and prints the value 1060.
- ▶ All local variables and parameters from the function “MyFunction” are captured.

Lambda expressions

- ▶ Example :

App.cpp

```
void MyFunction(int aa)
{
    int a, b, c;
    a = b = c = 10;
    auto f = [&] (int x, int y) { return x + y + a + b + c + aa; };
    printf("%d\n", f(10, 20));
    a = b = c = 100;
    aa *= 2;
    printf("%d\n", f(10, 20));
}
int main()
{
    MyFunction(1000);
    return 0;
}
```

- ▶ The code compiles and prints the value 1060 and then 2330
- ▶ All local variables and parameters from the function “MyFunction” are captured.

Lambda expressions

- ▶ Example :

App.cpp

```
void MyFunction(int aa)
{
    int a, b, c;
    a = b = c = 10;
    auto f = [&](int x, int y) { return x + y + a + b + c + aa; };
    printf("%d\n", f(10, 20));
    a = b = c = 100;
    aa *= 2;
    printf("%d\n", f(10, 20));
}
int main()
{
    MyFunction(1000);
    return 0;
}
```

Capture all local
variables and
parameters by reference

- ▶ The code compiles and prints the value 1060 and then 2330
- ▶ All local variables and parameters from the function “MyFunction” are captured (by reference).

Lambda expressions

- ▶ Example :

App.cpp

```
void MyFunction(int aa)
{
    int a, b, c;
    a = b = c = 10;
    auto f = [&, a] (int x, int y) { return x + y + a + b + c + aa; };
    printf("%d\n", f(10, 20));
    a = b = c = 100;
    aa *= 2;
    printf("%d\n", f(10, 20));
}
int main()
{
    MyFunction(1000);
    return 0;
}
```

- ▶ The code compiles and prints the value 1060 and then 2240. All variables are capture by reference except for local variable “a” that is capture by making a copy of itself.

Lambda expressions

- ▶ Example :

App.cpp

```
void MyFunction(int aa)
{
    int a, b, c;
    a = b = c = 10;
    auto f = [&,amp; a] (int x, int y) { return x + y + a + b + c + aa; };
    printf("%d\n", f(10, 20));
    a = b = c = 100;
    aa *= 2;
    printf("%d\n", f(10, 20));
}
int main()
{
    MyFunction(1000);
    return 0;
}
```

- ▶ Depending on the compiler this code might work. “Cl.exe” does not compile, GCC compiles with a warning.

Lambda expressions

- ▶ Example :

App.cpp

```
void MyFunction(int aa
{
    int a, b, c;
    a = b = c = 10;
    auto f = [&,amp; a] (int x, int y) { return x + y + a + b + c + aa; };
    printf("%d\n", f(10, 20));
    a = b = c = 100;
    aa *= 2;
    printf("%d\n", f(10, 20));
}
int main()
{
    MyFunction(1000);
    return 0;
}
```

GCC: warning: explicit by-reference capture of 'a'
redundant with by-reference capture default
CL: error C3488: '&a' cannot be explicitly captured
when the default capture mode is by reference (&t)

- ▶ Depending on the compiler this code might work. “Cl.exe” does not compile,
GCC compiles with an warning.

Lambda expressions

- ▶ Example :

App.cpp

```
void MyFunction(int aa)
{
    int a, b, c;
    a = b = c = 10;
    auto f = [=, &a] (int x, int y) { return x + y + a + b + c + aa; };
    printf("%d\n", f(10, 20));
    a = b = c = 100;
    aa *= 2;
    printf("%d\n", f(10, 20));
}
int main()
{
    MyFunction(1000);
    return 0;
}
```

- ▶ The code compiles and prints the value 1060 and then 1150. All variables are captured by making a copy of themselves except for local variable “a” that is captured by reference.

Lambda expressions

- ▶ Example :

App.cpp

```
void MyFunction(int aa)
{
    int a, b, c;
    a = b = c = 10;
    auto f = [=, a](int x, int y) { return x + y + a + b + c + aa; };
    printf("%d\n", f(10, 20));
    a = b = c = 100;
    aa *= 2;
    printf("%d\n", f(10, 20));
}
int main()
{
    MyFunction(1000);
    return 0;
}
```

- ▶ Depending on the compiler this code might work. “Cl.exe” does not compile, GCC compiles with a warning.

Lambda expressions

- ▶ Example :

App.cpp

```
void MyFunction(int aa)
{
    int a, b, c;
    a = b = c = 10;
    auto f = [a, a] (int x, int y) { return x + y + a; };
    printf("%d\n", f(10, 20));
    a = b = c = 100;
    aa *= 2;
    printf("%d\n", f(10, 20));
}
int main()
{
    MyFunction(1000);
    return 0;
}
```

error C3483: 'a' is already part of the lambda capture list

- ▶ This code will not compile as local variable 'a' can not be capture twice.

Lambda expressions

- ▶ Example :

App.cpp

```
void MyFunction(int aa)
{
    int a, b, c;
    a = b = c = 10;
    auto f = [a, &a] (int x, int y) { return x + y + a; };
    printf("%d\n", f(10, 20));
    a = b = c = 100;
    aa *= 2;
    printf("%d\n", f(10, 20));
}
int main()
{
    MyFunction(1000);
    return 0;
}
```

error C3483: 'a' is already part of the lambda capture list

- ▶ This code will not compile as local variable 'a' can not be capture twice. In this case we tried to capture 'a' making a copy of itself and also by reference.

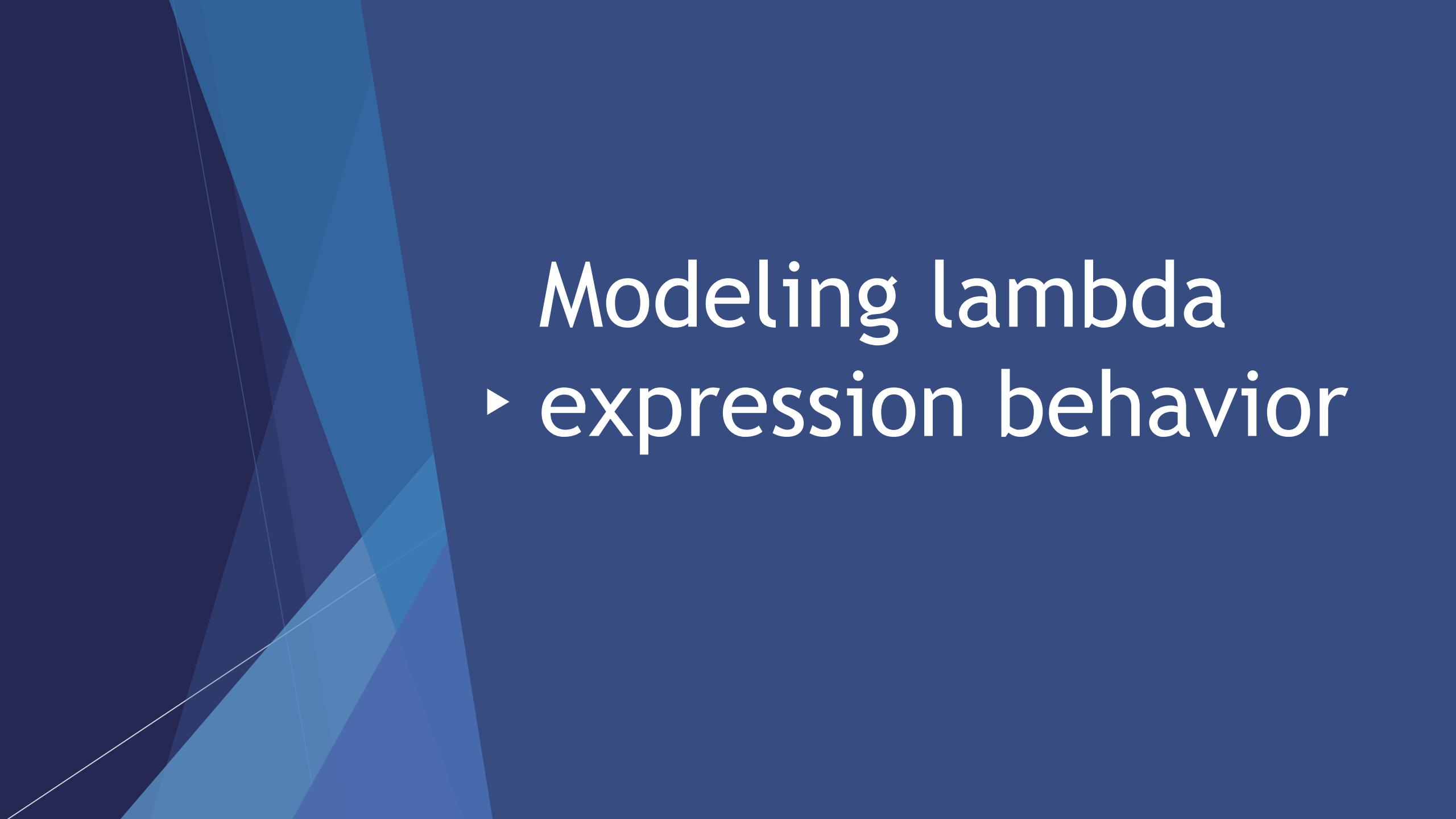
Lambda expressions

- ▶ Example :

App.cpp

```
int Add (int x, int y)
{
    return x + y;
}
void MyFunction(int aa)
{
    int a, b, c;
    a = b = c = 10;
    auto ptr_f = Add;
    auto f = [ptr_f](int x, int y) { return ptr_f(x, y); };
    printf("%d\n", f(10, 20));
}
int main()
{
    MyFunction(1000);
    return 0;
}
```

- ▶ This code compiles and prints “30” on the screen. In this case the capture variable is a pointer to a function (**Add**).

The background features a large, abstract graphic on the left side composed of overlapping blue triangles of varying shades of blue. It has a dark blue background with white text.

Modeling lambda
expression behavior

Lambda expressions

- ▶ Example:

App.cpp

```
int main()
{
    auto f = [] (int x, int y) { return x + y; };
    int x = f(10, 20);
    printf("X = %d", x);
    return 0;
}
```

- ▶ This example compiles and prints value 30 on the screen.

Lambda expressions

- ▶ Example:

Assembly code

```
push      ebp
mov       ebp,esp
sub       esp,4Ch
auto f = [](int x, int y) { return x + y; };
int x = f(10, 20);
push      14h
push      0Ah
lea       ecx,[f]
call     <lambda_1b12082d1acdf839b51735232aba4b6a>::operator()
mov       dword ptr [x],eax

printf("X = %d", x);
mov       eax,dword ptr [x]
push      eax
push      3A935Ch // address of "X = %d" string
call     printf
add       esp,8

return 0;
xor       eax,eax
```

Lambda expressions

- ▶ Example:

Assembly code

```
push  
mov  
sub  
auto f = [](i  
int x = f(10,  
push  
push 0Ah  
lea    ecx,[f]  
call   <lambda_1b12082d1acdf839b51735232aba4b6a>::operator()  
mov    dword ptr [x],eax  
  
printf("X = %d", x);  
mov    eax,dword ptr [x]  
push   eax  
push   3A935Ch // address of "X = %d" string  
call   printf  
add    esp,8  
  
return 0;  
xor    eax,eax
```

From the compiler point of view, lambda expressions are modeled as an object that has the () operator overwritten.

Lambda expressions

- ▶ Example:

Assembly code

```
push    ebp
mov     ebp,esp
sub     esp,4Ch
auto f = [](int x, int y) { return x + y; }
int x = f(10, 20);
push    14h
push    0Ah
lea     ecx,[f]
call    <lambda_1b12082d1acdf839b51735232aba4b6a>::operator()
mov     dword ptr [x],eax

printf("X = %d", x);
mov     eax,dword ptr [x]
push    eax
push    3A935Ch // address of "X = %d" string
call    printf
add    esp,8

return 0;
xor    eax,eax
```

```
push    ebp
mov     ebp,esp
sub     esp,44h
dword ptr [this],ecx

mov     eax,dword ptr [x]
add     eax,dword ptr [y]

mov     esp,ebp
pop    ebp
ret    8
```

Lambda expressions

- ▶ This means that this code is actually translated by the compiler as follows:

App.cpp

```
int main()
{
    auto f = [] (int x, int y) { return x + y; };
    int x = f(10, 20);
    printf("X = %d", x);
    return 0;
}
```

Translated code

```
class lambda_1b12082d1acdf839b51735232aba4b6a {
public:
    int operator() (int x,int y) const { return x+y; }
    lambda_1b12082d1acdf839b51735232aba4b6a () = delete;
};
int main()
{
    lambda_1b12082d1acdf839b51735232aba4b6a f;
    int x = f(10, 20);
    printf("X = %d", x);
    return 0;
}
```

Lambda expressions

- ▶ This means that this code is actually translated by the compiler as follows:

App.cpp

```
int main()
{
    auto f = [] (int x, int y) { return x + y; };
    int x = f(10, 20);
    printf("X = %d", x);
    return 0;
}
```

Translated code

```
class lambda_1b12082d1acdf839b51735232aba4b6a {
public:
    int operator() (int x,int y) const { return x+y; }
    lambda_1b12082d1acdf839b51735232aba4b6a () = delete;
}
int main()
{
    lambda_1b12082d1acdf839b51735232aba4b6a f;
    int x = f(10, 20);
    printf("X = %d", x);
    return 0;
}
```

Lambda expressions

- ▶ This means that this code is actually translated by the compiler as follows:

App.cpp

```
int main()
{
    auto f = [] (int x, int y) { return x + y; };
    int x = f(10, 20);
    printf("X = %d", x);
    return 0;
}
```

Translated code

```
class lambda_1b12082d1acdf839b51735232aba4b6a {
public:
    int operator() (int x,int y) const { return x+y; }
    lambda_1b12082d1acdf839b51735232aba4b6a () = delete;
}
int main()
{
    lambda_1b12082d1acdf839b51735232aba4b6a f;
    int x = f(10, 20);
    printf("X = %d", x);
    return 0;
}
```

Lambda expressions

- ▶ This means that this code is actually translated by the compiler as follows:

App.cpp

```
int main()
{
    auto f = [] (int x, int y) { return x + y; };
    int x = f(10, 20);
    printf("SizeOf(f) = %d", sizeof(f));
    return 0;
}
```

- ▶ The results will be 1 (consistent with the fact that “f” is indeed an object of type `lambda_1b12082d1acdf839b51735232aba4b6a`

Translated code

```
class lambda_1b12082d1acdf839b51735232aba4b6a {
public:
    int operator() (int x,int y) const { return x+y; }
    lambda_1b12082d1acdf839b51735232aba4b6a () = delete;
}
```

Lambda expressions

- ▶ The compiler generate `lambda_xxxxxxxxxxxxxx` classes for each encountered lambda structure.

App.cpp

```
int main()
{
    auto f1 = [](int x, int y) { return x + y; };
    auto f2 = [](int x, int y) { return x + y; };
    int x1 = f1(10, 20);
    int x2 = f2(10, 20);
    return 0;
}
```

- ▶ In the previous case, even if according to definition, both f1 and f2 are identical, two separate classes with two separate (but identical implementation) functions that overwrite operator() will be created.

Lambda expressions

- ▶ The compiler generate `lambda_xxxxxxxxxxxxxx` classes for each encountered lambda structure.

Assembly code

```
push      ebp
mov       ebp,esp
sub       esp,50h
auto f1 = [](int x, int y) { return x + y; };
auto f2 = [](int x, int y) { return x + y; };
int x1 = f1(10, 20);
push      14h
push      0Ah
lea        ecx,[f1]
call      <lambda_1b12082d1acdf839b51735232aba4b6a>::operator() (0923160h)
mov       dword ptr [x1],eax

int x2 = f2(10, 20);
push      14h
push      0Ah
lea        ecx,[f2]
call      <lambda_e213977a927692e36f5320f87e493de8>::operator() (0923280h)
mov       dword ptr [x2],eax

return 0;
xor       eax,eax
```

Lambda expressions

- ▶ The compiler generate `lambda_xxxxxxxxxxxxxx` classes for each encountered lambda structure.

Assembly code

```
push      ebp
mov       ebp,esp
sub       esp,50h
auto f1 = [](int x, int y) { return x + y; };
auto f2 = [](int x, int y) { return x + y; };
int x1 = f1(10, 20);
push      14h
push      0Ah
lea        ecx,[f1]
call      <lambda_1b12082d1acdf839b51735232aba4b6a>::operator() (0923160h)
mov       dword ptr [x1],eax
int x2 = f2(10, 20);
push      14h
push      0Ah
lea        ecx,[f2]
call      <lambda_e213977a927692e36f5320f87e493de8>::operator() (0923280h)
mov       dword ptr [x2],eax
return 0;
xor       eax,eax
```

Different classes

Lambda expressions

- ▶ The compiler generate `lambda_xxxxxxxxxxxxxx` classes for each encountered lambda structure.

Assembly code

```
push      ebp
mov       ebp,esp
sub       esp,50h
auto f1 = [](int x, int y) { return x + y; };
auto f2 = [](int x, int y) { return x + y; };
int x1 = f1(10, 20);
push      14h
push      0Ah
lea        ecx,[f1]
call      <lambda_1b12082d1acdf839b51735232aba4b6a>::operator() (0923160h)
mov       dword ptr [x1],eax
int x2 = f2(10, 20);
push      14h
push      0Ah
lea        ecx,[f2]
call      <lambda_e213977a927692e36f5320f87e493de8>::operator() (0923280h)
mov       dword ptr [x2],eax
return 0;
xor       eax,eax
```

Different functions for
operator()

Lambda expressions

- ▶ The compiler generate `lambda_xxxxxxxxxxxxxx` classes for each encountered lambda structure.
- ▶ This means that:
 - ▶ For every lambda construction that the programmer uses, a class will be created (it is therefore recommended that a lambda construction to be small so that they do not increase the size of the compiled program unnecessary).
 - ▶ The type of the class that uses lambda expressions is generated at the compile time → this means that whenever a lambda is used “auto” should be used as well.
 - ▶ The same lambda expression can be used multiple times if “`decltype`” is used (**this is valid for some compilers - not all of them allow this behavior !!!**)

Lambda expressions

- In the following case the usage of `decltype` allows us to reutilize the same construct multiple times

App.cpp (default constructor)

```
int main()
{
    auto f1 = [](int x, int y) { return x + y; };
    decltype(f1) f2;
    int x1 = f1(10, 20);
    int x2 = f2(10, 20);
    return 0;
}
```

App.cpp (copy constructor)

```
int main()
{
    auto f1 = [](int x, int y) { return x + y; };
    decltype(f1) f2 = f1;
    int x1 = f1(10, 20);
    int x2 = f2(10, 20);
    return 0;
}
```

- Now both “f1” and “f2” are of the same class/type.
- Default constructor does not work for every compiler (gcc does not support it, cl.exe (18.0.x.x supports it), cl.exe (19.16.27030.1 does not). The difference in this case is that the deleted constructor was not added in cl.exe (18.0.x.x)
- Copy constructor is supported by both cl (19.16.27030.1) and gcc.

Lambda expressions

- In the following case the usage of `decltype` allows us to reutilize the same construct multiple times

Assembly code

```
push      ebp
mov       ebp,esp
sub       esp,50h
auto f1 = [](int x, int y) { return x + y; };
decltype(f1) f2;
int x1 = f1(10, 20);
push      14h
push      0Ah
lea       ecx,[f1]
call     <lambda_1b12082d1acdf839b51735232aba4b6a>::operator() (0923160h)
mov       dword ptr [x1],eax
int x2 = f2(10, 20);
push      14h
push      0Ah
lea       ecx,[f2]
call     <lambda_1b12082d1acdf839b51735232aba4b6a>::operator() (0923160h)
mov       dword ptr [x2],eax
return 0;
xor       eax,eax
```

The same class

Lambda expressions

- ▶ Let's analyze the following case:

App.cpp

```
int main()
{
    int a, b;
    auto f = [a,b] (int x, int y) { return x + y + a +b;  };
    int x = f(10, 20);
    printf("sizeof(f) = %d", sizeof(f));
    return 0;
}
```

- ▶ The code compiles correctly and upon execution prints to the screen value 8.
- ▶ The size changed from 1 to 8 because of the 2 variables that were captured.

Lambda expressions

- ▶ Let's analyze the following case:

Assembly code

```
push      ebp
mov       ebp,esp
sub       esp,54h
int a, b;
auto f = [a,b](int x, int y) { return x + y + a +b;  };
lea       eax,[b]
push      eax
lea       ecx,[a]
push      ecx
lea       ecx,[f]
call     <lambda_3c006326...>::<lambda_3c006326...> (0D928E0h)
int x = f(10, 20);
push      14h
push      0Ah
lea       ecx,[f]
call     <lambda_3c006326...>::operator() (0D92730h)
mov       dword ptr [x],eax
printf("sizeof(f) = %d", sizeof(f));
.....
```

Lambda expressions

- ▶ Let's analyze the following case:

Assembly code

```
push    ebp
mov     ebp,esp
sub     esp,54h
int a, b;
auto f = [a,b](int x, int y) { return x + y +
lea     eax,[b]
push    eax
lea     ecx,[a]
push    ecx
lea     ecx,[f]
call    <lambda_3c006326...>::<lambda_3c006326...> (0D928E0h)
int x = f(10, 20);
push    14h
push    0Ah
lea     ecx,[f]
call    <lambda_3c006326...>::operator() (0D92730h)
mov     dword ptr [x],eax
printf("sizeof(f) = %d", sizeof(f));
.....
```

One difference from the previous times is that now we have a **constructor** for the lambda object

Lambda expressions

- ▶ Let's analyze the following case:

Assembly code

```
push    ebp
mov     ebp,esp
sub     esp,54h
int a, b;
auto f = [a,b](int x, int y) { return x + y;
lea     eax,[b]
push    eax
lea     ecx,[a]
push    ecx
lea     ecx,[f]
call    <lambda_3c006326...>::<lambda_3c006326...
int x = f(10, 20);
push    14h
push    0Ah
lea     ecx,[f]
call    <lambda_3c006326...>::operator() (0D92730h)
mov     dword ptr [x],eax
printf("sizeof(f) = %d", sizeof(f));
.....
```

```
push    ebp
mov     ebp,esp
sub     esp,44h
dword ptr [this],ecx
eax,dword ptr [this]
ecx,dword ptr [param_1]
edx,dword ptr [ecx]
dword ptr [eax],edx
eax,dword ptr [this]
ecx,dword ptr [param_2]
edx,dword ptr [ecx]
dword ptr [eax+4],edx
eax,dword ptr [this]
esp,ebp
ebp
ret
8
```

Lambda expressions

- ▶ This means that the same code can be translated as follows:

App.cpp

```
int main() {
    int a, b;
    auto f = [a,b] (int x, int y) { return x + y + a + b; };
    ...
    return 0;
}
```

Translated code

```
class lambda_3c006326 {
    int a,b;
public:
    lambda_3c006326(int &ref a, int &ref b): b(ref b), a(ref a) { }
    int operator() (int x,int y) const { return x + y + a + b; }
    lambda_3c006326() = delete;
}
int main() {
    int a, b;
    lambda_3c006326 f (a,b);
    ...
    return 0;
}
```

Lambda expressions

- ▶ Using the references changes the code as follows:

App.cpp

```
int main() {
    int a, b;
    auto f = [a & b] (int x, int y) { return x + y + a + b; };
    ...
    return 0;
}
```

Translated code

```
class lambda_3c006326 {
    int a;
    int & b;
public:
    lambda_3c006326(int & ref_a, int & ref_b): b(ref_b), a(ref_a) { }
    int operator() (int x, int y) const { return x + y + a + b; }
    lambda_3c006326() = delete;
}
int main() {
    int a, b;
    lambda_3c006326 f(a, b);
    ...
    return 0;
}
```

Lambda expressions

- ▶ Using `decltype` can be used for lambdas with no capture (that have a default constructor). In case of lambdas with capture `decltype` does not work.

App.cpp

```
int main() {
    int a = 10 , b = 20;
    auto f1 = [a,b] (int x, int y) { return x + y + a +b;  };

    decltype(f1) f2(a,b);

    printf("%d", f2(1, 2));
    return 0;
}
```

“f1” lambda class has a constructor with two integer parameters. However, the code will not compile (as it is not allowed to initialize a lambda in this way).

error C3497: you cannot construct an instance of a lambda

Lambda expressions

- ▶ Using **decltype** can be used for lambdas with no capture (that have a default constructor). In case of lambdas with capture **decltype** does not work.

App.cpp

```
int main() {
    int a = 10 , b = 20;
    auto f1 = [a,b] (int x, int y) { return x + y + a +b;  };

    decltype(f1) f2 = f1;

    printf("%d", f2(1, 2));
    return 0;
}
```

- ▶ This code will compile - a copy constructor between “f1” and “f2” is called.
- ▶ The code works and prints 33 into the screen.

Lambda expressions

- ▶ Using `decltype` can be used for lambdas with no capture (that have a default constructor). In case of lambdas with capture `decltype` does not work.

App.cpp

```
int main() {
    int a = 10 , b = 20;
    auto f1 = [a,b] (int x, int y) { return x + y + a +b;  };

    decltype(f1) f2 = {1,2};

    printf("%d", f2(1, 2));
    return 0;
}
```

- ▶ This code works on cl.exe (19.16.27030.1) for Windows but does not work on gcc
- ▶ Because of the initializer list “f2” in instantiated with two different values for internal (captured) “a” and “b”. On cl.exe for Windows the code works and prints 6.

Lambda expressions

- ▶ Copy constructor is used whenever the capture is done based on the value.

App.cpp

```
class MyNumber {  
public:  
    int a, b;  
    MyNumber(int x,int y): a(x), b(y) { }  
    MyNumber(const MyNumber &m) { a = m.b; b = m.a; }  
};  
int main() {  
    MyNumber m(2, 3);  
    auto f = [m](int x, int y) { return x * m.a + y * m.b; };  
    printf("%d\n", f(10, 20));  
    return 0;  
}
```

- ▶ In this case, when object “f” is created , the copy constructor for MyNumber is called and the actual object that is created within the lambda object has fields “a” and “b” reversed.
- ▶ The result of this code will be: $x (10) * m.a (3) + y (20) * m.b (2) = 70$

Lambda expressions

- ▶ However, if using references the copy constructor is not called and the result will be different.

App.cpp

```
class MyNumber {  
public:  
    int a, b;  
    MyNumber(int x,int y): a(x), b(y) { }  
    MyNumber(const MyNumber &m) { a = m.b; b = m.a; }  
};  
int main() {  
    MyNumber m(2, 3);  
    auto f = [&m](int x, int y) { return x * m.a + y * m.b; };  
    printf("%d\n", f(10, 20));  
    return 0;  
}
```

- ▶ The result of this code will be: $x (10) * m.a (2) + y (20) * m.b (3) = 80$

Lambda expressions

- ▶ Using “=” and/or “&” means that only values used in the body of the lambda are actually used (copied/referenced) in la lambda class.

App.cpp

```
int main(){
    int a1, a2, a3, a4, a5, a6;
    auto f = [=](int x, int y) { return x + y + a1 + a3; };
    printf("%d\n", sizeof(f));
    return 0;
}
```

- ▶ The result is 8 (only a1 and a3 are copied).

App.cpp

```
int main(){
    int a1, a2, a3, a4, a5, a6;
    auto f = [=] (int x, int y) { return x + y + a1 + a3 + a5; };
    printf("%d\n", sizeof(f));
    return 0;
}
```

- ▶ Now the result is 12 (a1, a3 and a5 are used)

Lambda expressions

- ▶ Lambdas can be used with classes and can capture `this` pointer

App.cpp

```
class Student{
public:
    const char *Name;
    int Grade;
public:
    Student(const char *n, int g) { Name = n; Grade = g; }
    void IncrementGrade()
    {
        auto la = [this] () { this->Grade++; };
        la();
    }
};
int main(){
    Student s("Popescu", 8);
    s.IncrementGrade();
}
```

- ▶ After the call of `s.IncrementGrade` the value of `s.Grade` will be 9

Lambda expressions

- ▶ Lambdas can be used with classes and can capture `this` pointer

App.cpp

```
class Student{
private:
    const char *Name;
    int Grade;
public:
    Student(const char *n, int g) { Name = n; Grade = g; }
    void IncrementGrade()
    {
        auto la = [this]() { this->Grade++; };
        la();
    }
};
int main(){
    Student s("Popescu", 8);
    s.IncrementGrade();
}
```

- ▶ Keep in mind that lambdas work similar to a friend function. Even if data members are private they can still be accessed. This code will run and the value of field Grade will be incremented.

Lambda expressions

- ▶ Lambdas can be used with classes and can capture `this` pointer

App.cpp

```
class Student{
private:
    const char *Name;
    int Grade;
public:
    Student(const char *n, int g) { Name = n; Grade = g; }
    void IncrementGrade()
    {
        auto la = [=]() { this->Grade++; };
        la();
    }
};
int main(){
    Student s("Popescu", 8);
    s.IncrementGrade();
}
```

- ▶ The same happens if we capture `this` by using ‘=’

Lambda expressions

- ▶ Lambdas can be used with classes and can capture `this` pointer

App.cpp

```
class Student{
private:
    const char *Name;
    int Grade;
public:
    Student(const char *n, int g) { Name = n; Grade = g; }
    void IncrementGrade()
    {
        auto la = [&] () { Grade++; };
        la();
    }
};
int main(){
    Student s("Popescu", 8);
    s.IncrementGrade();
}
```

- ▶ The same happens if we capture `this` by using '&'. Also the use of “`this->`” pointer in lambda function is not required.

Lambda expressions

- ▶ Lambdas can be used with classes and can capture `this` pointer

App.cpp

```
class Student{
private:
    const char *Name;
    int Grade;
public:
    Student(const char *n, int g) { Name = n; Grade = g; }
    void IncrementGrade()
    {
        auto la = []() { this->Grade++; };
        la();
    }
};
int main(){
    Student s("Popescu", 8);
    s.IncrementGrade();
}
```

- ▶ However, this code will not work as `this` pointer is not captured.



Implicit conversion to a pointer to a function

Lambda expressions

- ▶ All lambdas with no capture have an implicit conversion to a function pointer. This is normal as having no capture means that “this” pointer for the lambda structure is unnecessary.
- ▶ The following code works because “f” has no capture.

App.cpp

```
void Sort(int *number, int count, bool(*Compare)(int n1, int n2)) { ... }
int main()
{
    int numbers[] = { 1, 4, 2, 6, 1, 3 };
    auto f = [] (int n1, int n2) { return n1 > n2; };
    Sort(numbers, sizeof(numbers) / sizeof(int), f);
    return 0;
}
```

Lambda expressions

- ▶ All lambdas with no capture have an implicit conversion to a function pointer. This is normal as having no capture means that “this” pointer for the lambda structure is unnecessary.
- ▶ The following code will not work as “f” captures local variable “a”

App.cpp

```
void Sort(int *number, int count, bool(*Compare)(int n1, int n2)) { ... }
int main()
{
    int a = 100;
    int numbers[] = { 1, 4, 2, 6, 1, 3 };
    auto f = [a](int n1, int n2) { return n1 > n2; };
    Sort(numbers, sizeof(numbers) / sizeof(int), f);
    return 0;
}
```

Lambda expressions

- ▶ All lambdas with no capture have an implicit conversion to a function pointer. This is normal as having no capture means that “this” pointer for the lambda structure is unnecessary.
- ▶ The following code will work. “f” captures all used local variables / parameters by making a copy of them, but as the body of the lambda does not use any of them, the lambda is actually without capture.

App.cpp

```
void Sort(int *number, int count, bool(*Compare)(int n1, int n2)) { ... }
int main()
{
    int a = 100;
    int numbers[] = { 1, 4, 2, 6, 1, 3 };
    auto f = [=](int n1, int n2) { return n1 > n2; };
    Sort(numbers, sizeof(numbers) / sizeof(int), f);
    return 0;
}
```

Lambda expressions

- ▶ All lambdas with no capture have an implicit conversion to a function pointer. This is normal as having no capture means that “this” pointer for the lambda structure is unnecessary.
- ▶ The following code will NOT work because “f” captures all local variables/parameters by value (making a copy of them) and the body actually uses one of them (“a”).

App.cpp

```
void Sort(int *number, int count, bool(*Compare)(int n1, int n2)) { ... }
int main()
{
    int a = 100;
    int numbers[] = { 1, 4, 2, 6, 1, 3 };
    auto f = [=] (int n1, int n2) { return n1 > (n2 + a); };
    Sort(numbers, sizeof(numbers) / sizeof(int), f);
    return 0;
}
```

Lambda expressions

- ▶ How does the compiler models this behavior

App.cpp

```
void Sort(int *number, int count, bool(*Compare)(int n1, int n2)) { ... }
int main()
{
    int numbers[] = { 1, 4, 2, 6, 1, 3 };
    auto f = [=] (int n1, int n2) { return n1 > n2 ; };
    bool res = f(1, 2);
    Sort(numbers, sizeof(numbers) / sizeof(int), f);
    return 0;
}
```

Lambda expressions

- ▶ How does the compiler models this behavior

Assembly code

```
bool res = f(1, 2);
push    2
push    1
lea     ecx,[f]
call    <lambda_d87c98bc2...>::operator() (02929E0h)
mov     byte ptr [res],al
Sort(numbers, sizeof(numbers) / sizeof(int), f);
lea     ecx,[f]
call    <lambda_d87c98bc2...>::operator bool (__cdecl*)(int,int) (0293270h)
push    eax
push    6
lea     eax,[numbers]
push    eax
call    Sort (02912F3h)
add    esp,0Ch
return 0;
xor    eax,eax
```

Lambda expressions

- ▶ How does the compiler models this behavior

Assembly code

```
bool res = f(1, 2);
push    2
push    1
lea     ecx,[f]
call   <lambda_d87c98bc2...>::operator()
mov    byte ptr [res],al
      Sort(numbers, sizeof(numbers) / sizeof(int),
lea     ecx,[f]
call   <lambda_d87c98bc2...>::operator bool (__cdecl*)(int,int) (0293270h)
push    eax
push    6
lea     eax,[numbers]
push    eax
call   Sort (02912F3h)
add    esp,0Ch
      return 0;
xor    eax,eax
```

The compiler creates a cast function (that can cast to a pointer to a function).

Lambda expressions

- ▶ How does it work?

Assembly

```
bool Sort(int, int)
{
    push    ebp
    mov     ebp,esp
    sub    esp,44h
    mov     dword ptr [this],ecx
    mov     eax,292B20h
    mov     esp,ebp
    pop    ebp
    ret

    Sort(numbers)
    lea    ecx,[numbers]
    call   <lambda_d87c98bc2...>::operator bool (__cdecl*)(int,int) (0293270h)
    push   eax
    push   6
    lea    eax,[numbers]
    push   eax
    call   Sort (02912F3h)
    add    esp,0Ch
    return 0;
    xor    eax,eax
```

Lambda expressions

- ▶ How does it work?

Assembly

```
bool Sort(int, int)
{
    push    eb]
    mov     eb]
    sub    esp, 4
    mov     dword ptr [this], ecx
    mov     eax, 292B20h
    push    esp, ebp
    push    ebp
    lea     ecx, [numbers]
    call    <lambda_d87c98bc2...>::operator bool (__cdecl*)(int,int) (0293270h)
    push    eax
    push    6
    lea     eax, [numbers]
    push    eax
    call    Sort (02912F3h)
    add    esp, 0Ch
    return 0;
    xor    eax, eax
```

Address of the actual
function that can compare
two integers.

Lambda expressions

- ▶ This means that this code translates as follows:

App.cpp

```
int main()
{
    int numbers[] = { 1, 4, 2, 6, 1, 3 };
    auto f = [=] (int n1, int n2) { return n1>n2; };
    bool res = f(1, 2);
    Sort(numbers, sizeof(numbers) / sizeof(int), f);
    return 0;
}
```

Translated code

```
bool lambda_d87c98bc2_function(int n1,int n2) {
    return n1>n2;
}

class lambda_d87c98bc2 {
public:
    bool operator() (int n1,int n2) {
        return n1>n2;
    }
    operator bool (*)(int,int) {
        return lambda_d87c98bc2_function;
    }
}

int main()
{
    int numbers[] = { 1, 4, 2, 6, 1, 3 };
    lambda_d87c98bc2 f;
    bool res = f(1, 2);
    Sort(numbers, sizeof(numbers) / sizeof(int), f);
    return 0;
}
```

Lambda expressions

- ▶ This means that this code translates as follows:

App.cpp

```
int main()
{
    int numbers[] = { 1, 4, 2, 6, 1, 3 };
    auto f = [=] (int n1, int n2) { return n1>n2; };
    bool res = f(1, 2);
    Sort(numbers, sizeof(numbers) / sizeof(int), f);
    return 0;
}
```

Translated code

```
bool lambda_d87c98bc2_function (int n1,int n2) {
    return n1>n2;
}

class lambda_d87c98bc2 {
public:
    bool operator() (int n1,int n2) {
        return n1>n2;
    }
    operator bool (*)(int,int) {
        return lambda_d87c98bc2_function;
    }
}

int main()
{
    int numbers[] = { 1, 4, 2, 6, 1, 3 };
    lambda_d87c98bc2 f;
    bool res = f(1, 2);
    Sort(numbers, sizeof(numbers) / sizeof(int), f);
    return 0;
}
```

```
auto ptrFunction = (bool (*)(int,int)) f;
Sort (numbers, sizeof(numbers), ptrFunction)
```

Lambda expressions

- ▶ This means that this code translates as follows:

App.cpp

```
int main()
{
    int numbers[] = { 1, 4, 2, 6, 1, 3 };
    auto f = [=] (int n1, int n2) { return n1>n2; };
    bool res = f(1, 2);
    Sort(numbers, sizeof(numbers) / sizeof(int), f);
    return 0;
}
```

Translated code

```
bool lambda_d87c98bc2_function (int n1,int n2) {
    return n1>n2;
}

class lambda_d87c98bc2 {
public:
    bool operator() (int n1,int n2) {
        return n1>n2;
    }
    operator bool (*)(int,int) {
        return lambda_d87c98bc2_function;
    }
}

int main()
{
    int numbers[] = { 1, 4, 2, 6, 1, 3 };
    lambda_d87c98bc2 f;
    bool res = f(1, 2);
    Sort(numbers, sizeof(numbers) / sizeof(int), f);
    return 0;
}
```

auto **ptrFunction** = (bool (*)(int,int)) f;
Sort (numbers, sizeof(numbers), ptrFunction)

The background features a large, abstract graphic on the left side composed of various shades of blue and dark blue triangles, creating a polygonal pattern that tapers towards the bottom. The rest of the slide is a solid dark blue.

► Lambdas and STL

Lambda expressions

- ▶ A lambda expression can be used with STL (algorithm templates). The following code prints all elements from the vector “v”

App.cpp

```
#include <vector>
#include <algorithm>
int main(){
    vector<int> v = { 1, 2, 3, 5, 6, 7 };
    std::for_each (v.begin(), v.end(), [](int value){ printf("%d,", value); });
}
```

- ▶ The following code doubles the value of all values from vector “v”

App.cpp

```
#include <vector>
#include <algorithm>
int main()
{
    vector<int> v = { 1, 2, 3, 5, 6, 7 };
    std::for_each (v.begin(), v.end(), [](int &value){ value *= 2; });
    std::for_each (v.begin(), v.end(), [](int value){ printf("%d,", value); });
}
```

Lambda expressions

- ▶ A lambda expression can be used with STL (algorithm templates). The following code prints all elements from the vector “v”

App.cpp

```
#include <vector>
#include <algorithm>
int main(){
    vector<int> v = { 1, 2, 3, 5, 6, 7 };
    std::for_each (v.begin(), v.end(), [](int value){ printf("%d,", value); });
}
```

- ▶ The following code doubles the value of all values from vector “v”

App.cpp

```
#include <vector>
#include <algorithm>
int main()
{
    vector<int> v = { 1, 2, 3, 5, 6, 7 };
    std::for_each (v.begin(), v.end(), [](int &value){ value *= 2; });
    std::for_each (v.begin(), v.end(), [](int value){ printf("%d,", value); });
}
```

This app will print: 2,4,6,10,12,14

Lambda expressions

- ▶ Compute the number of odd numbers from a list:

App.cpp

```
#include <vector>
#include <algorithm>
int main(){
    vector<int> v = { 1, 2, 3, 5, 6, 7 };
    int odd_numbers = std::count_if (v.begin(), v.end(), [](int value) { return value % 2 == 0; });
    printf("%d\n", odd_numbers); // 2
}
```

- ▶ The following code removes all odd numbers from a list:

App.cpp

```
#include <vector>
#include <algorithm>
int main(){
    vector<int> v = { 1, 2, 3, 5, 6, 7 };
    v.erase(
        std::remove_if (v.begin(), v.end(), [](int value) { return value % 2 == 0; }),
        v.end()
    );
    // v = 1,3,5,7
}
```

Lambda expressions

- ▶ A lambda expression can be used with STL (*std::function*) component to describe a function

App.cpp

```
#include <functional>
int main(){
    std::function<int(int, int)> fnc = [] (int a, int b) -> int { return a + b; };
    printf("%d\n", fnc(10, 20));
}
```

- ▶ Usually this type of code is meant to replace a pointer to a function:

App.cpp

```
#include <functional>
typedef int (TypeIntegerSum) (int ,int );

int main()
{
    std::function<TypeIntegerSum> fnc = [] (int a, int b) -> int { return a + b; };
    printf("%d\n", fnc(10, 20));
}
```

Lambda expressions

- ▶ Instead of using pointers to a function, one can replace them with:

App.cpp

```
#include <functional>
typedef int (TypeCompare) (int ,int );

void Sort(int *numbers, int count, std::function<TypeCompare> &compareFunction)
{
}

int main(){
    int n[] = { 1, 2, 3, 4 };
    std::function<TypeCompare> cmpFunction;
    cmpFunction = [](int a, int b)->int { return a > b ? 1 : (a < b?(-1):0); };
    Sort(n, 4, cmpFunction);
}
```

- ▶ In this case we replace the standard **int (*)(int,int)** function with **std::function**

Lambda expressions

- ▶ Instead of using pointers to a function, one can replace them with:

App.cpp

```
#include <functional>
typedef int (TypeCompare) (int ,int );
void Sort(int *numbers, int count, std::function<TypeCompare> &compareFunction) { ... }
int main(){
    int x = -1;
    int y = 1;
    int n[] = { 1, 2, 3, 4 };
    std::function<TypeCompare> cmpFunction;
    cmpFunction = [x, y] (int a, int b)->int { return a > b ? x : (a < b ? y:0); };
    Sort(n, 4, cmpFunction);
}
```

- ▶ The main advantage in this case is that you can pass lambdas that have a caption (while in case of a cast to a function pointer you can not).
- ▶ The disadvantage is that using **std::function** is slower than using the pointer to a function directly.

Lambda expressions

- ▶ A lambda function and `std::function` can also be used with classes:

App.cpp

```
#include <functional>
class Student {
private:
    const char *Name;
    int Grade;
public:
    Student(const char *n, int g) { Name = n; Grade = g; }
    std::function<void()> GetIncrementFunction()
    {
        auto la = [&]() { Grade++; };
        std::function<void()> fnc = la;
        return fnc;
    }
};
int main() {
    Student s("Popescu", 8);
    auto fnc = s.GetIncrementFunction();
    fnc();
    return 0;
}
```

- ▶ After the execution of this code, `s.Grade` will be 9

Lambda expressions

- ▶ Be careful when using `std::function` with lambdas that capture local variables by reference !

App.cpp

```
#include <functional>
std::function<void(int)> GetFunction()
{
    int a = 100;
    printf("Address of a = %p\n");
    std::function<void(int)> fnc = [&a](int value) {
        printf("Address of a (from lambda) = %p\n“, &a);
        a += value;
    };
    return fnc;
}
int main(){
    auto f = GetFunction();
    f(10);
    return 0;
}
```

Lambda expressions

- ▶ Be careful when using `std::function` with lambdas that capture local variables by reference !

App.cpp

```
#include <functional>
std::function<void(int)> GetFunction()
{
    int a = 100;
    printf("Address of a = %p\n");
    std::function<void(int)> fnc = [&a](int value) {
        printf("Address of a (from lambda) = %p\n“, &a);
        a += value;
    };
    return fnc;
}
int main(){
    auto f = GetFunction();
    f(10);
    return 0;
}
```

Address of a = 00DE12D5
Address of a (from lambda) = 00DE12D5

- ▶ In this case a value on the stack (from the `GetFunction` stack) is modified outside `GetFunction` !

Lambda expressions

- ▶ Be careful when using std::function with lambdas that capture local variables by reference !

App.cpp

```
#include <functional>
std::function<void(int)> GetFunction(){
    int a = 100;
    std::function<void(int)> fnc = [&a](int value) { a += value; };
    return fnc;
}
void Test(std::function<void(int)> &fnc){
    int b = 10;
    std::function<void(int)> temp_fnc = [&b](int value) {   };
    fnc(5);
    printf("b = %d", b);
}
int main(){
    auto f = GetFunction();
    Test(f);
    return 0;
}
```

```
/permissive- /GS- /analyze- /W3 /Zc:wchar_t /ZI /Gm- /Od /sdl
/Fd"Debug\vc141.pdb" /Zc:inline /fp:precise /D "WIN32" /D "_DEBUG"
/D "_CONSOLE" /D "_UNICODE" /D "UNICODE" /errorReport:prompt
/WX- /Zc:forScope /RTCu /arch:IA32 /Gd /Oy- /MDd /FC /Fa"Debug\
/nologo /Fo"Debug\" /Fp"Debug\TestCpp.pch" /diagnostics:classic
```

- ▶ What will be printed on the screen upon the execution of this code ?
- ▶ Tested with cl.exe (19.16.27030.1), VS 2017

Lambda expressions

- ▶ Be careful when using `std::function` with lambdas that capture local variables by reference !

App.cpp

```
#include <functional>
std::function<void(int)> GetFunction(){
    int a = 100;
    std::function<void(int)> fnc = [&a](int value) { a += value; };
    return fnc;
}
void Test(std::function<void(int)> &fnc){
    int b = 10;
    std::function<void(int)> temp_fnc = [&b](int value) {   };
    fnc(5);
    printf("b = %d", b);
}
int main(){
    auto f = GetFunction();
    Test(f);
    return 0;
}
```

- ▶ The code compiles and prints **15** on the screen (even if b is 10). Similar results are highly dependent on the stack alignment.

Lambda expressions

- ▶ Be careful when using `std::function` with lambdas that capture local variables by reference !

App.cpp

```
#include <functional>
std::function<void(int)> GetFunction(){
    int a = 100;
    std::function<void(int)> fnc = [&a](int value) { a += value; };
    return fnc;
}
void Test(std::function<void(int)> &fnc){
    int b = 10;
    std::function<void(int)> temp_fnc = [&b](int value) {   };
    fnc(5);
    printf("b = %d", b);
}
int main(){
    auto f = GetFunction();
    Test(f);
    return 0;
}
```

The same stack layout !

- ▶ The code compiles and prints 15 on the screen (even if b is 10). Similar results are highly dependent on the stack alignment.

Lambda expressions

- ▶ Be careful when using `std::function` with lambdas that capture local variables by reference !

App.cpp

```
#include <functional>
std::function<void(int)> GetFunction(){
    int a = 100;
    std::function<void(int)> fnc = [&a](int value)
        return fnc;
}
void Test(std::function<void(int)> &fnc){
    int b = 10;
    std::function<void(int)> temp_fnc = [&b](int value) { };
    fnc(5);
    printf("b = %d", b);
}
int main(){
    auto f = GetFunction();
    Test(f);
    return 0;
}
```

Notice that `temp_fnc` is
not used at all !

- ▶ The code compiles and prints 15 on the screen (even if b is 10). Similar results are highly dependent on the stack alignment.

Lambda expressions

- ▶ Be careful when using local variable reference !

App.cpp

```
#include <functional>
std::function<void(int)> GetFunction(){
    int a = 100;
    std::function<void(int)> fnc = [a](int value) { a += value; };
    return fnc;
}
void Test(std::function<void(int)> fnc, std::function<void(int)> &tnc){
    int b = 10;
    std::function<void(int)> temp_fnc = [&b](int value) { b += value; };
    fnc(5); // Red box highlights this line
    printf("b = %d", b);
}
int main(){
    auto f = GetFunction();
    Test(f);
    return 0;
}
```

As local variable “**b**” from **Test** function and local variable “**a**” from **GetFunction** function have the same location in the stack, calling the lambda function from **fnc** will affect both of them.

- ▶ The code compiles and prints 15 on the screen (even if **b** is 10). Similar results are highly dependent on the stack alignment.



Using lambda with templates

- ▶ (Generic lambdas)

Lambda expressions

- ▶ Starting from C++14, lambda expressions can be used with auto parameters creating a template lambda.

App.cpp

```
int main()
{
    auto f = [] (auto x, auto y) { return x + y; };
    printf ("%d\n", f(10, 20));
    printf ("%lf\n", f(10.5, 20.7));
    return 0;
}
```

- ▶ The code compiles and prints: 30 and 31.2 into the screen. It only works for the standard C++14 and above.
- ▶ In this case the auto parameters work as a template (**this is not however a template !**)

Lambda expressions

- ▶ Starting from C++20, lambda expressions can be used with a template parameter.

App.cpp

```
int main()
{
    auto f = []<typename T> (T v1, T v2) { return v1 + v2; };
    printf ("%d %lf\n", f(10, 20), f(1.2,4.3));
    return 0;
}
```

- ▶ The code is no different than using auto, the main difference being that we can force a specific type , or a template of a specific type to the lambda expression.
- ▶ This code works with g++, but will not compile for cl.exe (VS 2017)
- ▶ The code will print 30 and 5.5

Lambda expressions

- ▶ Starting from C++20, lambda expressions can be used with a template parameter.

App.cpp

```
int main()
{
    auto f = []<typename T> (T v1, T v2) { return v1 + v2; };
    printf ("%d \n", f(10, 20.5));
    return 0;
}
```

- ▶ The code will not compile. The compiler fails to deduce type T (it can either be *int* or *double*) for the call `f(10, 20.5)`

Lambda expressions

- ▶ Starting from C++20, lambda expressions can be used with a template parameter.

App.cpp

```
int main()
{
    auto f = []<typename T> (T v1, T v2) {
        T temp;
        temp = v1 + v2;
        return (int)temp;
    };
    printf("%d\n", f(1.5, 2.2));
    return 0;
}
```

- ▶ In this case, we can force the lambda expression to return an *int* value (regardless of the type *T*).
- ▶ The result will be 3 (3.7 converted to int).

Lambda expressions

- ▶ Starting from C++20, lambda expressions can be used with a template parameter.

App.cpp

```
int main()
{
    auto f = []<typename T, typename R> (T v1, T v2) -> R {
        T temp;
        temp = v1 + v2;
        return (int)temp;
    };
    printf("%d\n", f(1.5, 2.2));
    return 0;
}
```

- ▶ In this case, the code will not compile, as type *R* can not be deduced. The problem is located in the fact that we specify that the result type is *R* but the compiler can not deduce it !
- ▶ Currently, using an explicit template for *f* (ex: *f<double,int>*) is not supported !

- ▶ Mutable capture

Lambda expressions

- ▶ Let's assume the following code:

App.cpp

```
int main()
{
    int a = 0;
    auto f = [&a](int x, int y) { a = x + y; };
    f(10, 20);
    printf("a = %d\n", a);
    return 0;
}
```

- ▶ The code compiles and prints “a = 30” on the screen.
- ▶ The capture “a” is done via a reference and it can be modified.

Lambda expressions

- ▶ Let's assume the following code:

App.cpp

```
int main()
{
    int a = 0;
    auto f = [a](int x, int y) { a = x + y; };
    f(10, 20);
    printf("a = %d\n", a);
    return 0;
}
```

error C3491: 'a': a by copy capture cannot
be modified in a non-mutable lambda

- ▶ The code however, will NOT work.
- ▶ But, as “a” is copied in the capture of the lambda expression, it should work like a class member and therefor we should be able to modify it (even if this will NOT affect the local variable “a” from the main function).
- ▶ What happens ?

Lambda expressions

- ▶ Let's look at the translated code:

App.cpp

```
int main() {
    int a = 0;
    auto f = [a](int x, int y) { a = x + y; };
    f(10, 20);
    printf("a = %d\n", a);
}
```

Translated code

```
class lambda_3c006326 {
    int a;
public:
    lambda_3c006326(int &ref_a): a(ref_a) { }
    void operator() (int x,int y) const { a = x + y; }
};
int main() {
    int a, b;
    lambda_3c006326 f (a);
    f(10, 20);
    printf("a = %d\n", a);
}
```

Lambda expressions

- ▶ Let's look at the translated code:

App.cpp

```
int main() {
    int a = 0;
    auto f = [a](int x, int y) { a = x + y; };
    f(10, 20);
    printf("a = %d\n", a);
}
```

Translated code

```
class lambda_3c006326 {
    int a;
public:
    lambda_3c006326(int &ref_a): a(ref_a) {}
    void operator()(int x,int y) const { a = x + y; }
}
int main() {
    int a, b;
    lambda_3c006326 f (a);
    f(10, 20);
    printf("a = %d\n", a);
}
```

The problem lies in the way operator() is defined !

Because of the “*const*” operator from the end of the definition, operator() can not modify any of its data members.

Lambda expressions

- ▶ The solution is to use “**mutable**” keyword when defining the lambda:

App.cpp

```
int main() {
    int a = 0;
    auto f = [a](int x, int y) mutable { a = x + y; };
    f(10, 20);
    printf("a = %d\n", a);
}
```

Translated code

```
class lambda_3c006326 {
mutable int a;
public:
    lambda_3c006326(int &ref_a): a(ref_a) { }
    void operator() (int x,int y) const { a = x + y; }
}
int main() {
    int a, b;
    lambda_3c006326 f (a);
    f(10, 20);
    printf("a = %d\n", a);
}
```

As a result, the created class has the internal data members defined as **mutable** (meaning that even if **operator()** is **const**, those data members can still be modified).

Lambda expressions

- ▶ The solution is to use “**mutable**” keyword when defining the lambda:

App.cpp

```
int main() {
    int a = 0;
    auto f = [a](int x, int y) mutable { a = x + y; };
    f(10, 20);
    printf("a = %d\n", a);
    return 0;
}
```

- ▶ The code compiles and prints “a = 0” into the screen.
- ▶ Even if the “**mutable**” keyword is used, as “a” was captured by making a copy of it and only the copy is modified when calling **f(10,20)**

Lambda expressions

- ▶ The solution is to use “**mutable**” keyword when defining the lambda:

App.cpp

```
int main()
{
    int index = 0;
    auto counter = [index] () mutable { return index++; };
    for (int tr = 0; tr < 10; tr++)
    {
        printf("%d,", counter());
    }
    return 1;
}
```

- ▶ The code compiles and prints 0,1,2,3,4,5,6,7,8,9 into the screen.

Lambda expressions

- ▶ The solution is to use “**mutable**” keyword when defining the lambda:

App.cpp

```
int main()
{
    int index = 0;
    auto counter = [index] () mutable -> int { return index++; };
    for (int tr = 0; tr < 10; tr++)
    {
        printf("%d,", counter());
    }
    return 1;
}
```

- ▶ If we want to describe the type of la lambda, then the **mutable** keyword should be added before the type (after the lambdas parameters)

The background features a large, abstract graphic on the left side composed of overlapping blue triangles of varying shades of blue. It has a dark blue background with white text.

Initialized lambda

- ▶ capture

Lambda expressions

- ▶ C++14 standards allows to initialize the lambda capture

App.cpp

```
int main()
{
    int a = 10, b = 20;

    auto f1 = [var1 = a+b, var2 = a-b] (int x, int y) { return x + y + var1 + var2; };

    printf("%d\n", f1(1, 2));
    return 0;
}
```

- ▶ In this case lambda “f1” has two variable captured (var1 and var2). “var1” equals $10+20 = 30$, and “var2” equals $10-20 = -10$;
- ▶ The code compiles under C++14 standards and prints $1+2+30-10 = 23$

Lambda expressions

- ▶ C++14 standards allows to initialize the lambda capture

App.cpp

```
int main()
{
    auto f1 = [counter = 0] () mutable { return counter++; };
    for (int tr=0;tr<10;tr++)
    {
        printf("%d\n",f1());
    }
    return 0;
}
```

- ▶ This type of initialization can be used to create lambdas with their own parameters. In this example, “f1” has one member (counter) that is initialized with 0 and incremented each time f1() is called.
- ▶ The code prints the numbers from 0 to 9
- ▶ **mutable** specifier is a MUST for this initialization to work. Without it, the code will produce a compile error as *counter* can not be modified.

Lambda expressions

- ▶ C++14 standards allows to initialize the lambda capture

App.cpp

```
int main()
{
    auto f1 = [counter = 1]() mutable { counter*=2; return counter; };
    for (int tr=0;tr<10;tr++)
    {
        printf("%d\n",f1());
    }
    return 0;
}
```

- ▶ In this case the type of counter is deduced to be int.
- ▶ The code compiles and prints 2,4,8,16,32,64,128,256,512,1024

Lambda expressions

- ▶ C++14 standards allows to initialize the lambda capture

App.cpp

```
int main()
{
    auto f1 = [unsigned char counter = 1] () mutable { counter*=2; return counter; };
    for (int tr=0;tr<10;tr++)
    {
        printf("%d\n",f1());
    }
    return 0;
}
```

- ▶ This code will not work → it is not allowed to set the type of capture. Type of capture is deduced from the assignment.

Lambda expressions

- ▶ C++14 standards allows to initialize the lambda capture

App.cpp

```
int main()
{
    auto f1 = [counter = (unsigned char) 1]() mutable { counter*=2; return counter; };
    for (int tr=0;tr<10;tr++)
    {
        printf("%d\n",f1());
    }
    return 0;
}
```

- ▶ However, you can force the type of such assignments by forcing the type of the evaluated value (usually using a cast).
- ▶ The code will compile and will print: 2,4,8,16,32,64,128,0,0,0
- ▶ The last 3 zeros are because counter is of type **unsigned char** and once it reaches value 256 it overflows and becomes 0.

Lambda expressions

- ▶ C++14 standards allows to initialize the lambda capture

App.cpp

```
int main()
{
    auto random = [seed = 1U] (unsigned int maxNumber) mutable {
        seed = 22695477U * seed + 1;
        return seed % maxNumber;
    };

    for (int tr=0;tr<10;tr++)
        printf("Pseudo random number between 0 and 99: %d \n", random(100));
    return 0;
}
```

- ▶ This example generates a pseudo random function based on the Linear congruential generator algorithm.
- ▶ The function uses the **seed** internal variable to generate the next random number.

The background features a large, abstract graphic on the left side composed of overlapping blue and dark blue triangles and trapezoids, creating a sense of depth and motion.

New feature in
▶ C++17 and beyond

Lambda expressions

- ▶ C++17 standards allows to capture (*this) → by using its copy constructor

App.cpp

```
struct MyClass
{
    int a;
    MyClass(int value) { a = value; }
    MyClass(const MyClass & obj) { std::cout << "Copy ctor" << std::endl; a = obj.a; }
    auto GetLambda() { return [this]() { std::cout << a << std::endl; }; }
};

int main()
{
    MyClass c = 10;
    auto f = c.GetLambda();
    f();
    c.a += 10;
    f();
    return 0;
}
```

- ▶ This code creates a lambda functions that captures *this* (as a pointer). The execution will print 10 and 20 on the screen (as lambda captures a reference to object c).

Lambda expressions

- ▶ C++17 standards allows to capture (*this) → by using its copy constructor

App.cpp

```
struct MyClass
{
    int a;
    MyClass(int value) { a = value; }
    MyClass(const MyClass & obj) { std::cout << "Copy ctor" << std::endl; a = obj.a; }
    auto GetLambda() { return [ *this ]() { std::cout << a << std::endl; }; }
};
int main()
{
    MyClass c = 10;
    auto f = c.GetLambda();
    f();
    c.a += 10;
    f();
    return 0;
}
```

- ▶ This code only works on C++17 standard. In this case, lambda captures a copy of c object, and the results printed on the screen will be: “**Copy ctor**”, then “**10**” and then “**10**” again (only the local **c** object is modified, not its copy).

Lambda expressions

- ▶ C++17 standards also allows creating a **constexpr** lambda expression

App.cpp

```
int main()
{
    int a = 10;
    constexpr auto f = [](int v1, int v2) constexpr { return v1 + v2; };
    printf("%d\n", f(1, 2));
    return 0;
}
```

- ▶ This code only works on C++17 standard. However, the generated cod does not show that the **constexpr** optimization is indeed applied. Code was tested with cl.exe, version 19.16.27025.1 and 19.16.27030.1 for x86 architecture

Generated assembly code for “**printf**(“%d\n”, f(1,2))”

```
push  2
push  1
lea   ecx,[f]
call  <lambda_f97ecf0fb43f36dc04c4d496e5fe74ab>::operator()
push  eax
push  offset string "%d\n"
call  printf
add   esp,8
```

Lambda expressions

- ▶ C++17 standards also allows creating a **constexpr** lambda expression

App.cpp

```
int main() {
    int a = 10;
    constexpr auto f = [](int v1, int v2) constexpr { return v1 + v2; };
    int aa[f(1, 2)];
    aa[0] = 0;
    return 0;
}
```

- ▶ It however works for the previous case, and “aa” local variable is instantiated.
- ▶ It also works with **static_assert** like in the next example.

App.cpp

```
int main()
{
    constexpr auto f = [](int v1, int v2) constexpr { return v1 + v2; };
    static_assert(f(1, 2) == 3);
    return 0;
}
```

Q & A

OOP (C++): Exceptions

Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

Object Oriented Programming 2020/2021

- 1** About Errors
- 2** Exception in OOP
Case Study: Parsing Regular Expressions
- 3** Exceptions as part of function specification
- 4** Standard Exceptions
- 5** Handling Unexpected Exceptions

- 1 About Errors
- 2 Exception in OOP
Case Study: Parsing Regular Expressions
- 3 Exceptions as part of function specification
- 4 Standard Exceptions
- 5 Handling Unexpected Exceptions

What the Experts Say

- "... I realized that from now on a large part of my life would be spent finding and correcting my own mistakes."
[Maurice Wilkes](#), 1949 (EDSAC computer), Turing Award in 1967
- "My guess is that avoiding, finding, and correcting errors is 95% or more of the effort for serious software development."
"Error handling is a difficult task for which the programmer needs all the help that can be provided."
[Bjarne Stroustrup](#) (C++ creator)

This rate can be improved by using the right programming skills!

Three Main Requirements for a Program

- to produce the desired outputs, as specified, for legal inputs (**correctness**)
- to give reasonable error messages for illegal entries
- to allow termination in case of an error

A Taxonomy of Errors

- compilation errors
 - detected by the compiler
 - generally easy to fix
- linking errors
 - functions/methods not implemented
 - library access
- run-time errors
 - the source may be the computer, a component of a library, or the **program** itself
- logical errors
 - the program does not have the desired behavior
 - can be detected by the programmer (by testing)
... or by the user (customer)

Errors Must be Reported

```
int PolygonalLine::length() {  
    if (n <= 0)  
        return -1  
    else {  
        // ...  
    }  
}  
  
int p = L.length();  
if (p < 0)  
    printError("Bad computation of a polygonal line");  
// ...
```

How to Report an Error?

- error reporting must be uniform: same message for the same type of error (location information may differ)
- ... therefore an "error indicator" must be managed
- association of numbers for errors is a solution but it can be problematic
- OOP has the necessary means to smartly organize the detection and reporting of errors

Error vs Exception

- often the two terms are confused
- however, there is a (subtle) difference between the meanings of the two notions
- error = **abnormal behavior** detected during operation to be eliminated by repairing the program
- exception = **unforeseen behavior** that can occur in rare or very rare situations
- exceptions should be handled in the case of programs
- **an untreated exception is an error!**

In this lecture we discuss more about exceptions.

Plan

- 1 About Errors
- 2 Exception in OOP
- Case Study: Parsing Regular Expressions
- 3 Exceptions as part of function specification
- 4 Standard Exceptions
- 5 Handling Unexpected Exceptions

General Principles

- the exceptions in OOP were designed with the intention of separating the business logic from the mechanism of error transmission
- the purpose is to allow the handling of errors, which occur as exceptions, at an appropriate level that does not interfere with business logic

Plan

- 1 About Errors
- 2 Exception in OOP
Case Study: Parsing Regular Expressions
- 3 Exceptions as part of function specification
- 4 Standard Exceptions
- 5 Handling Unexpected Exceptions

Recall from Algorithm Design 1/3

Syntax for regular expressions:

```
expression ::= term ("+" term)*
term ::= maybeStar
        | maybeStar ("." maybeStar)*
maybeStar ::= factor ["*"]
            /* equiv to factor | factor "*" */
factor ::= 
    empty
    | Sigma
    | "(" expression ")"
Sigma ::= "a" | "b" | ...
```

Recall from Algorithm Design 2/3

Alg data structure for AST: $ast(e)$

- $empty: []$
- $\varepsilon: ["", \langle\rangle]$
- $a \in \Sigma: ["a", \langle\rangle]$
- $e_1 e_2 \dots: ["\cdot\cdot\cdot", \langle ast(e_1), ast(e_2), \dots \rangle]$
- $e_1 + e_2 + \dots: ["\cdot+\cdot", \langle ast(e_1), ast(e_2), \dots \rangle]$
- $e*: ["\cdot*", \langle ast(e) \rangle]$

Example: $(ab + b) * (ba)$

$["\cdot\cdot\cdot", \langle ["\cdot*", \langle ["\cdot+\cdot", \langle ["\cdot\cdot\cdot", \langle ["a", \langle\rangle], ["b", \langle\rangle] \rangle], ["b", \langle\rangle] \rangle \rangle], ["\cdot\cdot\cdot", \langle ["b", \langle\rangle], ["a", \langle\rangle] \rangle \rangle]$

Recall from Algorithm Design 3/3

Some functions from AST domain:

```
// number of children
chldNo(ast) {
    if (ast.size() > 0) return ast[1].size();
    return 0;
}

// the i-th child of an AST
chld(ast, i) {
    if (ast.size() > 0 && i < ast[1].size()) {
        return ast[1].at(i);
    }
}
```

From Alk to C++: Class for ASTs, First Attempt

```
class Ast {  
private:  
    string label;  
    vector<Ast*> children;  
  
public:  
    AstNonEmpty(string aLabel = "",  
                vector<Ast*> aList = vector<Ast*>())  
    {  
        label = aLabel;  
        children = aList;  
    }  
    ...  
};
```

We have a problem: how to represent the empty AST ("[]")?
In the above declaration, the default constructor builds the AST for
the empty string ϵ .

From Alk to C++: Class for ASTs, Second Attempt

```
class Ast {          //abstract class
public:
    virtual int chldNo() = 0;
    virtual Ast* child(int i) = 0;
    ...
};

class AstEmpty : public Ast {        //empty AST []
public:
    AstEmpty() {}           //empty object
    ...
};

class AstNonEmpty : public Ast {    //non empty AST ["...",<...>]
private:
    string label;
    vector<Ast*> children;
public:
    AstNonEmpty(string aLabel = "",
                vector<Ast*> aList = vector<Ast*>())
    {
        label = aLabel;
        children = aList;
    }
    ...
};
```

Implementation of AstEmpty (partial)

```
int AstEmpty::chldNo() {  
    #??  is it OK to return -1? ⇐ exception  
}  
Ast* AstEmpty::child(int i) {  
    #??  is it OK to return new AstEmpty? ⇐ exception  
}
```

Implementation of AstNonEmpty (partial)

```
Ast* AstNonEmpty::child(int i) {
    if (0 <= i and i < chldNo()) {
        return children[i];
    }
    #?? is it OK to return new AstEmpty? ⇐ exception
}
```

Recall from Algorithm Design: the Parser

Global variables:

`input` - the expression given as input

`sigma` - the alphabet

`index` - the current position in the input

- the current symbol:

```
sym() modifies index, input {
    if (index < input.size())
        return input.at(index);
    return "\0";
}
```

- next symbol

```
nextSym() modifies index, input {
    if (index < input.size()) {
        index++;
    } else
        error("nextsym: expected a symbol");
}
```

- ...

From Alk description to C++ 1/3

```
class Parser {  
private:  
    string input;  
    vector<char> sigma;  
    int index;  
public:  
    Parser(vector<char> alphabet = vector<char>()) {  
        sigma = alphabet;  
        input = "";  
        index = 0;  
    }  
    ...  
}
```

From Alk description to C++ 2/3

```
char Parser::sym() {
    if (index < input.size()) {
        return input[index];
    }
    return '\0';
}

void Parser::nextSym() {
    if (index < input.size()) {
        index++;
    }
    else {
        error("nextsym: expected a symbol"); // ←exception
    }
}
```

From Alk description to C++ 3/3

Alk

```
factor() {
    s = sym();
    if (acceptSigma()) {
        return [s,<>];
    } else if (accept("(")) {
        ast = expression();
        expect(")");
        return ast;
    }
}
```

C++

```
Ast* Parser::factor() {
    Ast* past;
    char s = sym();
    if (acceptSigma()) {
        past =
            new AstNonEmpty(string(1, s));
    } else if (accept('(')) {
        past = expression();
        expect(')');
    }
    // else ?? ⇐ exception
    return past;
}
```

Similar the other methods.

Testing

Test source:

```
int main() {
    vector<char> alph{'a', 'b', 'c'}; // c++11
    Parser parser(alph);
    parser.setInput("(a.b+");
    past = parser.expression();
    past->print();
    return 0;
}
```

Test Run:

```
$ g++ ast.cpp parser.cpp test-parser.cpp -std=c++11
$ ./a.out
Bus error: 10
```

try, throw and catch

From the manual:

- throw** expression signals that an exceptional condition—often, an error—has occurred in a try block. You can use an object of any type as the operand of a throw expression. Typically, this object is used to communicate information about the error
- try** block is used to enclose one or more statements that might throw an exception (business component)
- catch** blocks are implemented immediately following a try block. Each catch block specifies the type of exception it can handle (error handling component)

try – throw – catch play



try
(business logic)



throw
(exception thrown)



catch
(exception handling)

Ast Hierarchy with exceptions

Consider only problematic methods:

```
int AstEmpty::chldNo() {
    throw "Error: trying to access children of an empty AST";
}

Ast* AstEmpty::child(int i) {
    throw "Error: trying to access children of an empty AST";
}

Ast* AstNonEmpty::child(int i) {
if (0 <= i and i < chldNo()) {
    return children[i];
}
throw "Error: index out of bounds.";
}
```

Testing, w/o Try

Test source:

```
int main() {
    AstEmpty ast;
    cout << "ast.child(1)->print(): ";
    ast.child(1)->print();
    cout << endl;
    cout << "ast.chldNo(): ";
    cout << ast.chldNo();
    cout << endl;
    return 0;
}
```

Test Run:

```
$ g++ ast.cpp parser.cpp test-ast-empty.cpp -std=c++11
$ ./a.out
libc++abi.dylib: terminating with uncaught exception
    of type char const*
ast.child(1)->print(): Abort trap: 6
```

Testing, w/ Try

Test source:

```
try {      //business logic block
    AstEmpty ast;
    cout << "ast.child(1)->print(): ";
    ast.child(1)->print();
    cout << endl;
    cout << "ast.chldNo(): ";
    cout << ast.chldNo();
    cout << endl;
}
catch (char const* msg) { //exception handling block
    cout << msg << endl;
    cout << "Here the exceptions can be handled." << endl;
}
```

Test Run:

```
$ g++ ast.cpp parser.cpp test-ast-empty-w-try.cpp -std=c++11
$ ./a.out
ast.child(1)->print(): Error: trying to access children
    of an empty AST
Here the exceptions can be handled.
```

Parser Class with exceptions 1/2

Only problematic methods:

```
void Parser::nextSym() {
    if (index < input.size()) {
        index++;
    }
    else {
        throw "Error: index out of input size";
    }
}

//bool Parser::expect(char s) {
void Parser::expect(char s) {
    if (accept(s)) {
        // return true;
        return;
    }
    // error("unexpected symbol at position " + to_string(index)
    // return false;
    throw s;
}
```

Parser Class with exceptions 2/2

Only problematic methods:

```
Ast* Parser::factor() {
    Ast* past;
    char s = sym();
    if (acceptSigma()) {
        past = new AstNonEmpty(string(1, s));
    } else if (accept('(')) {
        past = expression();
        expect(')');
    }
    // else ???
    else
        throw "expected alphabet symbol or (.";
    return past;
}
```

Testing, w/ Try 1/3

Test source:

```
try{
    vector<char> alph{'a', 'b', 'c'}; // c++11
    Parser parser(alph);
    parser.setInput("(a.b+");
    // wrong expression
    Ast* past = parser.expression();
    past->print();
}
catch (char const* msg) {
    cout << msg << endl;
    cout << "Here the parsing exceptions can be handled."
}
```

Test Run:

```
$ g++ ast.cpp parser.cpp test-parser-exc2.cpp -std=c++11
$ ./a.out
expected alphabet symbol or (.  
Here the parsing exceptions can be handled.
```

Testing, w/ Try 2/3

Test source:

```
try{
    vector<char> alph{'a', 'b', 'c'}; // c++11
    Parser parser(alph);
    parser.setInput("(a.b+c"); // wrong expression
    Ast* past = parser.expression();
    past->print();
}
catch (char const* msg) {
    cout << msg << endl;
    cout << "Here the parsing exceptions can be handled." << e
}
```

Test Run:

```
$ g++ ast.cpp parser.cpp test-parser-exc2.cpp -std=c++11
$ ./a.out
libc++abi.dylib: terminating with uncaught exception
of type char
Abort trap: 6
```

Testing, w/ Try 3/3

Test source:

```
try{
    vector<char> alph{'a', 'b', 'c'}; // c++11
    Parser parser(alph);
    parser.setInput("(a.b+c"); // wrong expression
    Ast* past = parser.expression();
    past->print();
}
catch (char const* msg) { // catching C string exceptions
    cout << msg << endl;
    cout << "Here the parsing exceptions can be handled." << endl;
}
catch (char s) { // catching char exceptions
    cout << "expected character " << s << endl;
    cout << "Here the parsing exceptions can be handled." << endl;
}
```

Test Run:

```
$ g++ ast.cpp parser.cpp test-parser-exc2.cpp -std=c++11
$ ./a.out
expected character )
Here the parsing exceptions can be handled.
```

Hierarchy of Exceptions 1/3

It is recommended to organize the exceptions into a hierarchy.
AST exceptions I:

```
class MyException {  
public:  
    virtual void debugPrint() {  
        std::cout << "Exception: ";  
    }  
};  
  
class AstException : public MyException {  
public:  
    virtual void debugPrint() {  
        this->MyException::debugPrint();  
        std::cout << "AST: ";  
    }  
};  
  
class EmptyAstException : public AstException {  
public:  
    virtual void debugPrint() {  
        this->AstException::debugPrint();  
        std::cout << "trying to access an empty tree.";  
    }  
};
```

Hierarchy of Exceptions 2/3

AST exceptions II:

```
class NonEmptyAstException : public AstException {
private:
    int badIndex;
public:
    void setBadIndex(int j) {
        badIndex = j;
    }
    virtual void debugPrint() {
        this->AstException::debugPrint();
        std::cout << "index out of bounds (" << badIndex << ")";
    }
};
```

Hierarchy of Exceptions 3/3

Parser exceptions:

```
class ParserException : public MyException {
public:
private:
    int badPos;
    std::string errMsg;
public:
    void setBadPos(int j) {
        badPos = j;
    }
    void setErrMsg(std::string msg) {
        errMsg = msg;
    }
    virtual void debugPrint() {
        this->MyException::debugPrint();
        std::cout << "Parsing: " << errMsg
                  << " at input position " << badPos << ".\n";
    }
};
```

Ast Hierarchy with exceptions, revisited

Consider only problematic methods:

```
int AstEmpty::chldNo() {
    throw EmptyAstException();
}

Ast* AstEmpty::child(int i) {
    throw EmptyAstException();
}

Ast* AstNonEmpty::child(int i) {
    if (0 <= i and i < chldNo()) {
        return children[i];
    }
    NonEmptyAstException exc;
    exc.setBadIndex(i);
    throw exc;
}
```

Parser Class with exceptions, revisited 1/2

Only problematic methods:

```
void Parser::nextSym() {
    if (index < input.size()) {
        index++;
    }
    else {
        ParserException exc;
        exc.setBadPos(index);
        throw exc;
    }
}

void Parser::expect(char s) {
    if (accept(s)) {
        return;
    }
    ParserException exc;
    exc.setErrMsg(string("unexpected symbol"));
    exc.setBadPos(index);
    throw exc;
}
```

Parser Class with exceptions, revisited 2/2

Only problematic methods:

```
Ast* Parser::factor() {
    Ast* past;
    char s = sym();
    if (acceptSigma()) {
        past = new AstNonEmpty(string(1, s));
    } else if (accept('(')) {
        past = expression();
        expect(')');
    } // else ???
    else {
        ParserException exc;
        exc.setErrMsg(string("expected alphabet symbol or ("));
        exc.setBadPos(index);
        throw exc;
    }
    return past;
}
```

Menu function:

```
void printMenu() {  
    cout << "\n1. Empty AST exception.\n";  
    cout << "2. Nonempty AST exception.\n";  
    cout << "3. Parser exception.\n";  
    cout << "0. Exit.\n";  
    cout << "Option: ";  
}
```

Testing 2/3

Test source:

```
while (opt != 0) {  
    try{  
        printMenu();  
        cin >> opt;  
        switch (opt) {  
            case 1: {  
                AstEmpty* peast = new AstEmpty();  
                peast->child(0)->print();  
                break;  
            }  
            ...  
        }  
    }  
    catch (MyException& exc) {  
        exc.debugPrint();  
        cout << endl;  
        parser.setIndex(0);  
    }  
}
```

Test Run:

Testing 3/3

```
$ ./a.out
1. Empty AST exception.
2. Nonempty AST exception.
3. Parser exception.
0. Exit.
Option: 1
Exception: AST: trying to access an empty tree.
1. Empty AST exception.
2. Nonempty AST exception.
3. Parser exception.
0. Exit.
Option: 2
Exception: AST: index out of bounds (2).
1. Empty AST exception.
2. Nonempty AST exception.
3. Parser exception.
0. Exit.
Option: 3
Exception: Parsing: expected alphabet symbol or (at input position
7.
1. Empty AST exception.
2. Nonempty AST exception.
3. Parser exception.
0. Exit.
Option: 0
```

Plan

- 1 About Errors
- 2 Exception in OOP
Case Study: Parsing Regular Expressions
- 3 Exceptions as part of function specification
- 4 Standard Exceptions
- 5 Handling Unexpected Exceptions

Specify what exceptions are thrown

```
Ast* Parser::factor() throw(ParserException) {
    Ast* past;
    char s = sym();
    if (acceptSigma()) {
        past = new AstNonEmpty(string(1, s));
    } else if (accept('(')) {
        past = expression();
        expect(')');
    } else {
        ParserException* pexc = new ParserException();
        pexc->setErrMsg(string("expected alphabet symbol or ("));
        pexc->setBadPos(index);
        throw pexc;
    }
    return past;
}
```

Empty throw Specification

- C++ 2003

```
void f() throw();
```

- C++ 2011

```
void f() noexcept(true);
```

Example 1/2

Exceptions in destructors may lead to unpredictable behavior:

```
class A {
    public:
        ~A() {
            throw "Thrown by Destructor";
        }
};

int main() {
    try {
        A a;
    }
    catch(const char *exc) {
        std::cout << "Print " << exc;
    }
}
```

Output:

```
libc++abi.dylib: terminating with uncaught exception
of type char const*
Abort trap: 6
```

Example 2/2

Using `noexcept` spec we get predictable behavior:

```
class A {  
public:  
    ~A() noexcept(false) {  
        throw "Thrown by Destructor";  
    }  
};
```

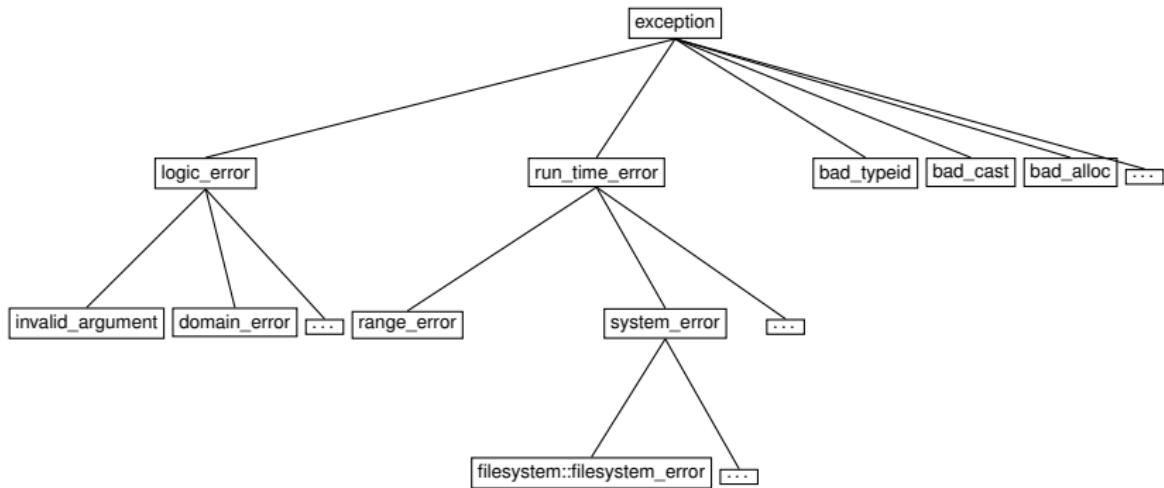
Output:

Print This Thrown by Destructor

Plan

- 1 About Errors
- 2 Exception in OOP
Case Study: Parsing Regular Expressions
- 3 Exceptions as part of function specification
- 4 Standard Exceptions
- 5 Handling Unexpected Exceptions

Standard Exceptions Hierarchy (partial)



Example: regex-error (<regex>)

Source code:

```
try {
    std::regex re("[a-z]*[a");
}
catch (const std::regex_error& e) {
    std::cout << "regex_error caught: " << e.what() << '\n';
    if (e.code() == std::regex_constants::error_brack) {
        std::cout << "The code was error_brack\n";
    }
}
```

Testing:

```
$ g++ regex-error.cpp -std=c++11
$ ./a.out
regex_error caught:
The expression contained mismatched [ and ].
The code was error_brack
```

Example: bad-function-call (<functional>)

Source code:

```
std::function<int ()> f = nullptr;
try {
    f();
} catch(const std::bad_function_call& e) {
    std::cout << "bad_function_call caught: " << e.what() << '\n';
}
return 0;
```

Testing: Mac OS

```
$ g++ bad-function-call.cpp -std=c++11
$ ./a.out
bad_function_call caught: std::exception
```

Windows:

```
> cl bad-function-call.cpp /std:c++14
> bad-function-call.exe
bad_function_call caught: bad function call
```

Example: bad-alloc (<new>) 1/3

Source code:

```
int negative = -1;
int small = 1;
int large = INT_MAX;
int opt = -1;
while (opt != 0) {
    try {
        printMenu();
        cin >> opt;
        switch (opt) {
            case 1: new int[negative]; break;
            case 2: new int[small]{1,2,3}; break;
            case 3: new int[large][1000000]; break;
            default: opt = 0;
        }
    } catch(const std::bad_array_new_length &e) {
        cout << "bad_array_new_length caught: " << e.what() << '\n';
    } catch(const std::bad_alloc &e) {
        cout << "bad_alloc caught: " << e.what() << '\n';
    } catch(...) {
        cout << "unknown exceptions caught.";
    }
}
```

Example: bad-alloc (<new>) 2/3

Testing Mac OS:

1. Negative.
2. Small.
3. Large.

Option: 1

```
bad_alloc caught: std::bad_alloc
```

1. Negative.
2. Small.
3. Large.

Option: 2

```
bad_alloc caught: std::bad_alloc
```

1. Negative.
2. Small.
3. Large.

Option: 3

```
a.out(21194,0x11e7b15c0) malloc: can't allocate region
*** mach_vm_map(size=8589934588002304) failed (error code=3)
a.out(21194,0x11e7b15c0) malloc: *** set a breakpoint in malloc_error_
bad_alloc caught: std::bad_alloc
```

1. Negative.
2. Small.
3. Large.

Option:

Example: bad-alloc (<new>) 3/3

Testing Windows:

1. Negative.
2. Small.
3. Large.

Option: 1

```
bad_array_new_length caught: bad array new length
```

1. Negative.
2. Small.
3. Large.

Option: 2

1. Negative.
2. Small.
3. Large.

Option: 3

```
bad_array_new_length caught: bad array new length
```

exception Class

```
class exception {  
public:  
    exception () throw();  
    exception (const exception&) throw();  
    exception& operator= (const exception&) throw();  
    virtual ~exception() throw();  
    virtual const char* what() const throw();  
    // ...  
}
```

Deriving from exception

```
class MyException : public exception {
public:
    const char * what () const throw ()
    {
        return "Division by zero. ";
    }
};

int safeDiv(int dividend, int divisor)
{
    if (divisor == 0)
        throw MyException();
    return dividend / divisor;
}
```

Testing

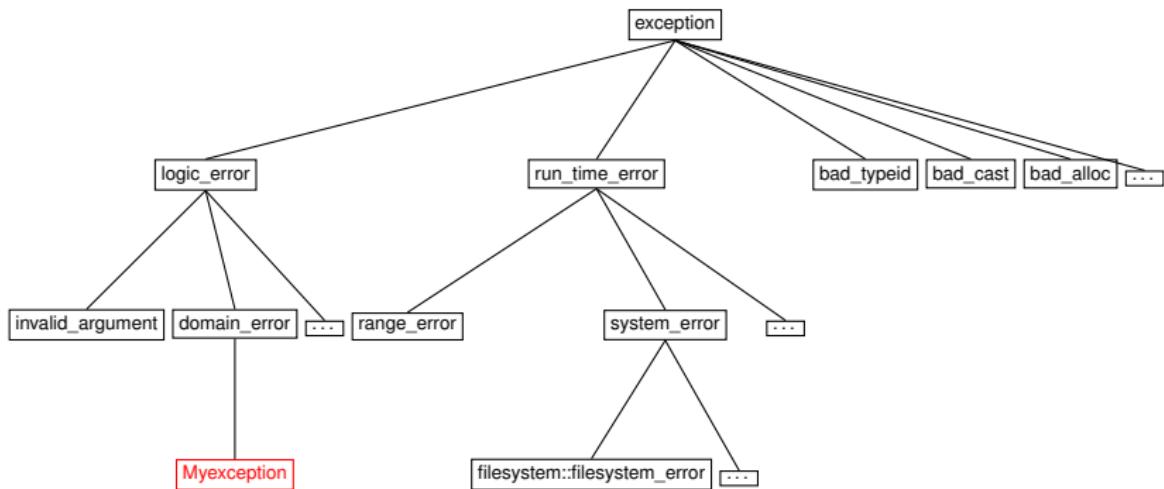
Source:

```
try {
    safeDiv(3, 0);
} catch(MyException& exc) {
    std::cout << "MyException caught" << std::endl;
    std::cout << exc.what() << std::endl;
} catch(std::exception& e) {
    //Other errors
}
```

Output:

```
$ g++ demo.cpp
$ ./a.out
MyException caught
Division by zero.
```

Deriving from Standard Exceptions



Plan

- 1 About Errors
- 2 Exception in OOP
Case Study: Parsing Regular Expressions
- 3 Exceptions as part of function specification
- 4 Standard Exceptions
- 5 Handling Unexpected Exceptions

Missing a Throw 1/2

```
class One : public exception { };
class Two : public exception { };
void g() {
    throw "Surprise.";
}
void fct(int x) throw (One, Two) {
    switch (x) {
    case 1: throw One();
    case 2: throw Two();
    }
    g();
}
```

Missing a Throw 2/2

Testing source:

```
cout << "Option (1-3): "; cin >> option;
try {
    fct(option);
} catch (One) {
    cout << "Exception one" << endl;
} catch (Two) {
    cout << "Exception two" << endl;
}
```

Output:

```
$ ./a.out
Option (1-3): 2
Exception two
$ ./a.out
Option (1-3): 3
libc++abi.dylib: terminating with unexpected exception
of type char const*
Abort trap: 6
```

Using set_unexpected

Write a function to be executed when an unexpected exception occurs:

```
void my_unexpected() {  
    cout << "My unexpected exception.\n";  
    exit(1);  
}
```

In the main program set it as such a function:

```
set_unexpected(my_unexpected);
```

Testing:

```
$ g++ missed-throw.cpp  
$ ./a.out  
Option (1-3): 3  
My unexpected exception.
```

noexcept () Operator

- tests whether or not an expression throws an exception
- returns `false` if there is any subexpression that can throw exceptions, i.e., if there is any expression that is not specified with `noexcept(true)` or `throw()`
- returns `true` if all subexpressions are specified with `noexcept(true)` or `throw()`
- very useful when moving objects (e.g., `reserve()` method)

noexcept () : Example

```
class A {  
public:  
    A() noexcept(false) {  
        throw 2;  
    }  
};  
  
class B {  
public:  
    B() noexcept(true) {}  
};
```

noexcept () :: Testing 1/3

Source:

```
template <typename T>
void f() {
    if (noexcept(T()))
        std::cout << "NO Exception in Constructor" << std::endl
    else
        std::cout << "Exception in Constructor" << std::endl
}

int main() {
    f<A>();
    f<B>();
}
```

Output:

```
$ g++ noexcept-operator.cpp -std=c++11
$ ./a.out
Exception in Constructor
NO Exception in Constructor
```

noexcept () :: Testing 2/3

Testing in a similar way the destructor does not work:

```
class C {
public:
    C() noexcept(true) {}
    ~C() noexcept(false) {}
};

template <typename T>
void g() {
    if (noexcept(~T()))
        std::cout << "NO Exception in Destructor\n";
    else
        std::cout << "Possible Exception in Destructor\n";
}

int main() {
    g<C>();
}
```

Output:

```
$ g++ -std=c++17 noexcept-declval.cpp
noexcept-declval.cpp:13:16: error: invalid argument type 'C' to
    if (noexcept(~T()))
           ^~~~
```

noexcept () :: Testing 3/3

```
Use std::declval<T>():

class C {
public:
    C() noexcept(true) {}
    ~C() noexcept(false) {}
};

template <typename T>
void g() {
    if (noexcept(std::declval<T>(); ~T()))
        std::cout << "NO Exception in Destructor\n";
    else
        std::cout << "Possible Exception in Destructor\n";
}

int main() {
    g<C>();
}
```

Output:

```
$ g++ -std=c++17 noexcept-declval.cpp
$ ./a.out
Possible Exception in Destructor
```

C++2017: noexcept is a Part of Function Type

Source:

```
void g() noexcept {}
int main()
{
    auto f = g;
    std::cout << std::boolalpha \
        << noexcept(f()) << std::endl;
}
```

Output:

```
$ g++ -std=c++17 exc-part-of-type.cpp
$ ./a.out
true
```

Recommendations

- always specify exceptions
- always start with standard exceptions
- declare the exceptions of a class within it
- uses exception hierarchies
- captures by references
- throw exceptions in constructors
- attention to memory release for partially created objects
- be careful how you test if an object has been created OK
- do not cause exceptions in destructors unless it is a must and then do it very carefully (preferably in C ++ 2011 or after)

Recommendations

Some reasons why it is not recommended to throw exceptions by constructors/destructors

- if a constructor throws an exception when the stack is in an unstable state, then program execution ends
- it is almost impossible to design predictable and accurate containers in the presence of exceptions in destructors
- certain pieces of C ++ code may have undefined behavior when destructors throw exceptions
- what happens to the object whose "destruction" failed (because the destructor method threw an exception)?

OOP (C++): Testing

Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

Object Oriented Programming 2020/20201

- ① CMake
- ② Unit Testing with Google Test
- ③ Test Fixtures
- ④ Test-driven development (TDD)
- ⑤ TDD = A Simple Case Study

Introduction

- In this lesson we learn how to use a unit-testing framework.
- The framework we use is Google Test
<https://github.com/google/googletest/>
- Usually, a unit-testing framework is used together with a build automation tool.
- We use CMake, a cross platform builder:

<https://cmake.org/>

Plan

- 1 CMake
- 2 Unit Testing with Google Test
- 3 Test Fixtures
- 4 Test-driven development (TDD)
- 5 TDD = A Simple Case Study

Brief Description

- A CMake-based buildsystem is organized as a set of high-level logical targets.
- Each target corresponds to an executable, or a library, or is a custom target that contains custom commands.
- Dependencies between targets are expressed in the generation system to determine the order of generation and the rules for regeneration when changes are made.
- CMake commands are written in **CMake Language** and included in **CMakeLists.txt** files or with the **.cmake** extension.

Resources

- An introduction:

<https://cmake.org/cmake/help/v3.20/manual/cmake.1.html>

- Detailed description:

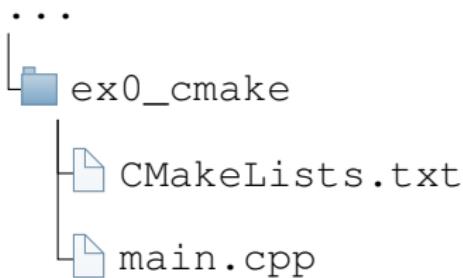
<https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html>

- CMake Language is described in

<https://cmake.org/cmake/help/latest/manual/cmake-language.7.html>

First Example: Folder Structure

This is the most simple example using CMake.



First Example: CMakeLists.txt

```
# CMake minimum version required
cmake_minimum_required(VERSION 3.16)

# the name of the project
project(ex0)

# C++ minimum version required
# GoogleTest requires at least C++11
set(CMAKE_CXX_STANDARD 14)

# add an executable target
add_executable(ex0_main main.cpp)
```

First Example: main.cpp

```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

First Example: Demo with CLion

First Example: Command line building

We work with a copy `ex0_cmake_cmd`

```
cd ex0_cmake_cmd // go to the example folder
cmake -H. -Bbuild // create the scripts in the "build" folder
cd build
cmake -build . // the building process
ls
CMakeCache.txt      Makefile           ex0_main
CMakeFiles          cmake_install.cmake
./ex0_main // run the generated binary
```

Third Example: Folder Structure

Let's elaborate a bit the folder structure.



Third Example: New commands in CMakeLists.txt

```
add_executable(ex2_date src/main.cpp src/Date.cpp)

# for additional headers available for including
#   from the sources of all targets
include_directories/includes)

# add a library target
add_library(ex2_library STATIC src/Date.cpp)

# linking the library
find_library(EX2_LIBRARY ex2_library lib)
target_link_libraries(ex2_main LINK_PUBLIC ${EX2_LIBRARY})
```

Third Example: Date.h

```
//...
class Date {
private:
    int year;
    int month;
    int day;
public:
    void setToday(int aYear, int aMonth, int aDay);
    string to_string();
    string today();
    string tomorrow();
private:
    static bool isLeapYear(int year);
    static int daysNo(Date& d);
};
//...
```

Third Example: Demo with CLion

Plan

- 1 CMake
- 2 Unit Testing with Google Test
- 3 Test Fixtures
- 4 Test-driven development (TDD)
- 5 TDD = A Simple Case Study

Testing Frameworks for C++

- **Boost Test Library** https://www.boost.org/doc/libs/1_66_0/libs/test/doc/html/index.html
- **CppUnit** http://cppunit.sourceforge.net/doc/cvs/cppunit_cookbook.html
- **CUTE (C++ Unit Testing Easier)**
<http://cute-test.com/>
- **Google C++ Mocking Framework**
<https://github.com/google/googletest/tree/master/googmock>
- **Google Test** <https://github.com/google/googletest/tree/master/googletest>
- **Microsoft Unit Testing Framework for C++**
<https://msdn.microsoft.com/en-us/library/hh598953.aspx>
- ...

On Google Test

From Google Test primer:

- ① Tests should be independent and repeatable.
- ② Tests should be well organized and reflect the structure of the tested code.
- ③ Tests should be portable and reusable.
- ④ When tests fail, they should provide as much information about the problem as possible.
- ⑤ The testing framework should liberate test writers from housekeeping chores and let them focus on the test content.
- ⑥ Tests should be fast.

Unit Testing Terminology

Assertion A statement that checks if a condition holds (is true).

Test A collection of assertions that check the behaviour of a piece of code.

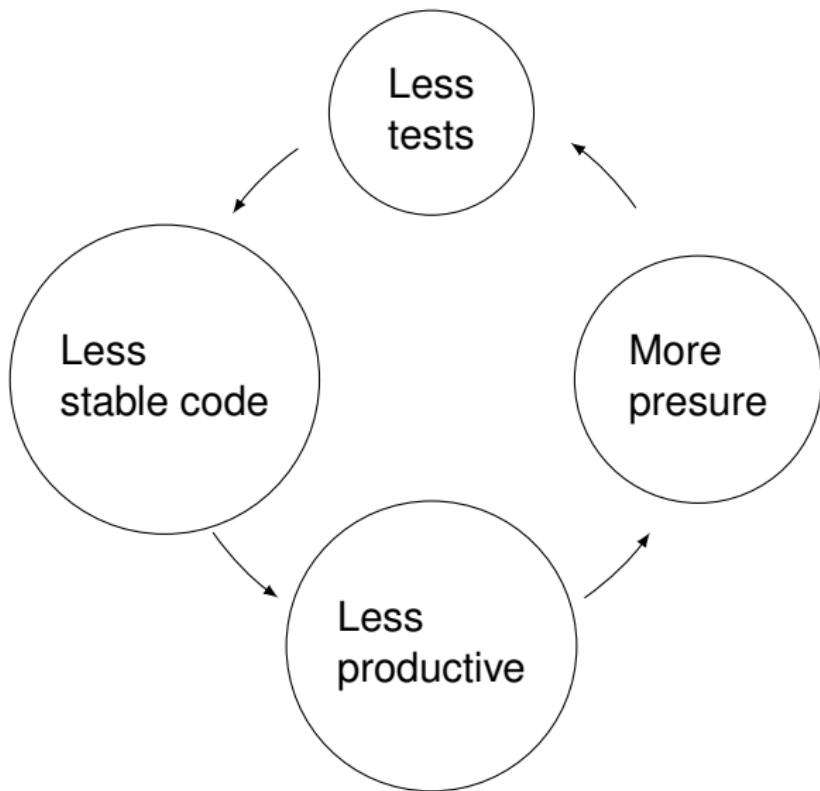
Test suite A group of test. The test suite should reflect the structure of the code.

Test program A set of test suites.

Advantages of using unit testing (UT)

- UT helps to fix bugs in early stages.
- UT helps to understand better the code testing.
- UT helps to understand better what functionality should be provided by a unit code.
- UT allows to test some parts of the code without waiting the other ones to be finished.
- UT helps to reuse the code.

Unit testing myth



The structure of a Google test

```
TEST(TestSuiteName, TestName) {  
    ... test body ...  
}
```

Six Steps to Follow

- Step 1: Initialize the project
- Step 2: Incorporate the Google Test project
- Step 3: Add the source files to the project
- Step 4: Add tests
- Step 5: Complete the files with the test configuration information
- Step 6: Build and execute the project

First Example with tests: Folder Structure

This is the most simple example using CMake.



First Example with tests: New commands in CMakeLists.txt

```
# fetch Google Test
include(FetchContent)
FetchContent_Declare(
    gtest
    URL https://github.com/google/googletest/archive/refs/tags/release-1.10.0.zip
)
# For Windows: Prevent overriding the parent project's compiler/linker settings
set(gtest_force_shared_crt ON CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(gtest)

enable_testing()

# add an executable test target
add_executable(
    hello_test
    hello_test.cpp
)

# link the Google Test library
target_link_libraries(
    hello_test
    gtest_main
)

# enable CMake's test runner to discover the tests included in the binary,
# using the GoogleTest CMake module
include(GoogleTest)
gtest_discover_tests(hello_test)
```

First Example with tests: hello_test.cpp

```
#include <gtest/gtest.h>

// Demonstrate some basic assertions.
TEST(HelloTest, BasicAssertions) {
    // Expect two strings not to be equal.
    EXPECT_STRNE("mine", "yours");
    // Expect equality.
    EXPECT_EQ(2 + 2, 4);
}
```

First Example with tests: Demo with CLion

First Example with tests: Command line building

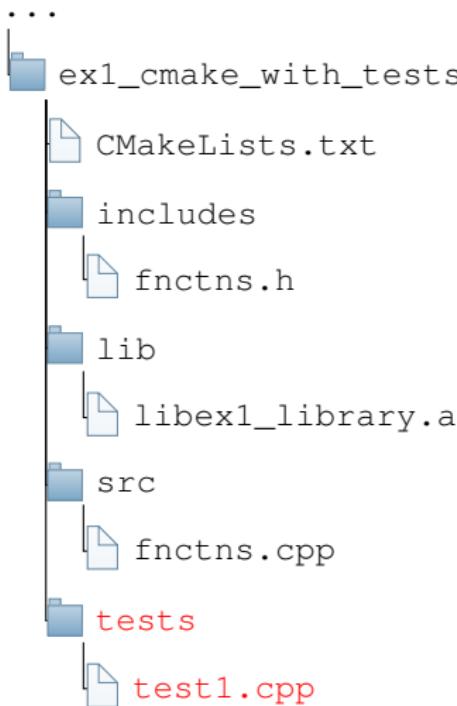
We work with a copy `ex0_cmake_cmd`

```
cd ex0_cmake_cmd // go to the example folder
cmake -H. -Bbuild // create the scripts in the "build" folder
cd build
cmake -build . // the building process
ls
CMakeCache.txt           cmake_install.cmake
CMakeFiles                  ex0_main
CTestTestfile.cmake        hello_test
Makefile                   hello_test[1]_include.cmake
_deps                      hello_test[1]_tests.cmake
bin                         lib
./hello_test // run the generated tests
...
[=====] Running 1 test from 1 test suite.
[-----] Global test environment set-up.
[-----] 1 test from HelloTest
[ RUN      ] HelloTest.BasicAssertions
[       OK ] HelloTest.BasicAssertions (0 ms)
[-----] 1 test from HelloTest (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
```

Second Example with tests: Folder Structure

This is the most simple example using CMake.



Second Example with tests: Specific commands in CMakeLists.txt

```
...
# add an executable target
add_executable(ex1_main src/main.cpp)
add_executable(ex1_fnctns src/main.cpp src/fnctns.cpp)
...
# add a library target
add_library(ex1_library STATIC src/fnctns.cpp)

# linking the library
find_library(EX1_LIBRARY ex1_library lib)
target_link_libraries(ex1_main LINK_PUBLIC $EX1_LIBRARY)
...
# fetch Google Test
...
enable_testing()

# add an executable test target
add_executable(ex1_test1 tests/test1.cpp src/fnctns.cpp)

# link the Google Test library
target_link_libraries(ex1_test1 gtest_main)

# enable CMake's test runner to discover the tests
...
gtest_discover_tests(ex1_test1)
```

Second Example with tests: test1.cpp

```
#include <gtest/gtest.h>
#include "../includes/fnctns.h"
#include <climits>

// Tests for inc2
TEST(Ex1Test, BasicAssertionsInc2) {
    EXPECT_EQ(5, inc2(3));
    ASSERT_TRUE(inc2(7) > 7);
}

// Tests for sqr
TEST(Ex1Test, BasicAssertionsSqr) {
    EXPECT_EQ(9, sqr(3));
    ASSERT_TRUE(sqr(-5) > 0);
}

// Tests for combinations of the two
TEST(Ex1Test, CombinedAssertions) {
    EXPECT_EQ(inc2(2), sqr(2));
    ASSERT_TRUE(sqr(7) > inc2(7));
    ASSERT_TRUE(inc2(1) > sqr(1));
}

// Tests for boundary cases
TEST(Ex1Test, BoundaryAssertions) {
    ASSERT_TRUE(inc2(INT_MAX) > INT_MAX);
    ASSERT_TRUE(sqr(INT_MAX/5) > INT_MAX/5);
}
```

Second Example with tests: Demo with CLion

Second Example with tests: Command line building

We work with a copy `ex1_cmake_with_tests_cmd`

```
cd ex1_cmake_with_tests_cmd // go to the example folder
cmake -H. -Bbuild // create the scripts in the "build" folder
cd build
cmake -build . // the building process
./ex1_test1
...
[=====] Running 4 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 4 tests from Ex1Test
[ RUN      ] Ex1Test.BasicAssertionsInc2
[ OK       ] Ex1Test.BasicAssertionsInc2 (0 ms)
[ RUN      ] Ex1Test.BasicAssertionsSqr
[ OK       ] Ex1Test.BasicAssertionsSqr (0 ms)
[ RUN      ] Ex1Test.CombinedAssertions
[ OK       ] Ex1Test.CombinedAssertions (0 ms)
[ RUN      ] Ex1Test.BoundaryAssertions
.../examples/ex1_cmake_with_tests_cmd/tests/test1.cpp:30: Failure
Value of: inc2(INT_MAX) > INT_MAX
  Actual: false
Expected: true
[ FAILED   ] Ex1Test.BoundaryAssertions (0 ms)
[-----] 4 tests from Ex1Test (0 ms total)

[-----] Global test environment tear-down
[=====] 4 tests from 1 test suite ran. (0 ms total)
[ PASSED   ] 3 tests.
[ FAILED    ] 1 test, listed below:
[ FAILED   ] Ex1Test.BoundaryAssertions

1 FAILED TEST
```

Plan

- 1 CMake
- 2 Unit Testing with Google Test
- 3 Test Fixtures
- 4 Test-driven development (TDD)
- 5 TDD = A Simple Case Study

What Is a Test Fixture

- Allows the use of the same object configuration for multiple tests
- It is recommended for testing classes
- The steps to follow are:
 - ① Derive a class from :: testing :: Test.
 - ② Inside the class, state any objects you intend to use.
 - ③ If necessary, write a default constructor or a **SetUp()** function to prepare the objects for each test. A common mistake is to write **SetUp()** as **Setup()**, with a small u - don't let this happen.
 - ④ If necessary, write a destructor or **TearDown()** function to free up the resources allocated in **SetUp()**.
 - ⑤ If necessary, define methods to allow test sharing.

Further Info

[https://github.com/google/googletest/blob/
master/googletest/docs/FAQ.md](https://github.com/google/googletest/blob/master/googletest/docs/FAQ.md)

TEST_F() Macro

- it must be used whenever we have "test fixture" type objects
- template:

```
TEST_F(test_case_name, test_name) {  
    ... test body ...  
}
```

- the first parameter must be the name of the "test fixture" class
- you must first define a test fixture class before using it in a TEST_F()

How TEST_F() Works

For each test defined with TEST_F(), Google Test will:

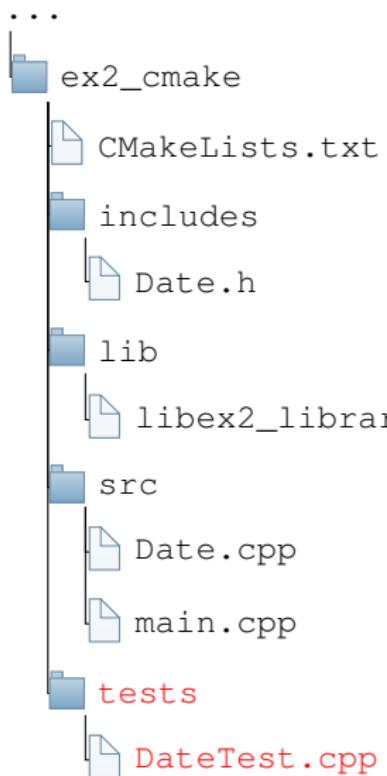
- ① create a new "test fixture" at runtime
- ② initialize with SetUp ()
- ③ run the test
- ④ free memory by calling TearDown()
- ⑤ delete "test fixture".

Keep in mind that different tests in the same test have different test objects, and Google Test always deletes one test item before creating the next one.

Google Test does not reuse the same "fixture test" for multiple tests. Any changes that a test makes to the "test fixture" do not affect other tests.

Third Example: Folder Structure

Let's elaborate a bit the folder structure.



Third Example: New commands in CMakeLists.txt

```
...
# fetch Google Test
...
# add an executable test target
add_executable(ex2_test1 tests/DateTest.cpp src/Date.cpp)

# link the Google Test library
target_link_libraries(ex2_test1 gtest_main)

# enable CMake's test runner to discover the tests included in
...
gtest_discover_tests(ex2_test1)
```

Third Example: DateTest.cpp

```
#include "gtest/gtest.h"
#include "Date.h"

class DateTest : public testing::Test {
protected: // You should make the members protected s.t. they can be
           // accessed from sub-classes.

    // virtual void SetUp() will be called before each test is run.  You
    // should define it if you need to initialize the variables.
    // Otherwise, this can be skipped.
    virtual void SetUp() {
        d1_.setToday(2021, 5, 10);
        d2_.setToday(2020, 2, 28);
        d3_.setToday(2020, 12, 31);
    }

    // virtual void TearDown() will be called after each test is run.
    // You should define it if there is cleanup work to do.  Otherwise,
    // you don't have to provide it.
    //
    // virtual void TearDown() {
    // }

    // Declares the variables your tests want to use.
    Date d1_, d2_, d3_;
};

// When you have a test fixture, you define a test using TEST_F
// instead of TEST.

// Tests for today().
TEST_F(DateTest, Today) {
    ASSERT_STREQ("10.5.2021", d1_.today().c_str());
}
```

What Happen When a Test is Executed

For the Today test, Google Test builds a DateTest object (let's call it t1). Then the following steps are performed.

- t1.SetUp () initializes t1.
- The first test (DefaultConstructor) is run over t1.
- t1.TearDown () cleans up after the test (if any).
- t1 is destroyed.

The above steps are repeated with another DateTest object, in this case the Tomorrow test.

Third Example: Demo with CLion

Third Example with tests: Command line building

We work with a copy `ex2_cmake_with_tests_cmd`

```
cd ex2_cmake_with_tests_cmd // go to the example folder
cmake -H. -Bbuild // create the scripts in the "build" folder
cd build
cmake -build . // the building process
./ex2_test1
...
[=====] Running 2 tests from 1 test suite.
[-----] Global test environment set-up.
[-----] 2 tests from DateTest
[ RUN      ] DateTest.Today
[       OK ] DateTest.Today (0 ms)
[ RUN      ] DateTest.Tomorrow
[       OK ] DateTest.Tomorrow (0 ms)
[-----] 2 tests from DateTest (0 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test suite ran. (0 ms total)
[  PASSED  ] 2 tests.
```

Plan

- 1 CMake
- 2 Unit Testing with Google Test
- 3 Test Fixtures
- 4 Test-driven development (TDD)
- 5 TDD = A Simple Case Study

A Giant's Dialog

From SOFTWARE ENGINEERING TECHNIQUES Report on a conference sponsored by the NATO SCIENCE COMMITTEE Rome, Italy, 27th to 31st October 1969:

Hoare: One can construct convincing proofs quite readily of the ultimate futility of exhaustive testing of a program and even of testing by sampling. So how can one proceed? **The role of testing, in theory, is to establish the base propositions of an inductive proof.** You should convince yourself, or other people, as firmly as possible that if the program works a certain number of times on specified data, then it will always work on any data.

...

Dijkstra: **Testing shows the presence, not the absence of bugs.**

On TDD

“One of the primary reasons I switched to TDD is for improved test coverage, which leads to 40%-80% fewer bugs in production. This is my favorite benefit of TDD. It’s like a giant weight lifting off your shoulders.”

(Eric Elliott, TDD Changed My Life)

Test-driven development (TDD): Definition

“Test-driven development” refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).

It can be succinctly described by the following set of rules:

- write a “single” unit test describing an aspect of the program
- run the test, which should fail because the program lacks that feature
- write “just enough” code, the simplest possible, to make the test pass
- “refactor” the code until it conforms to the simplicity criteria
- repeat, “accumulating” unit tests over time

TDD: Expected Benefits

- many teams report significant reductions in defect rates, at the cost of a moderate increase in initial development effort
- the same teams tend to report that these overheads are more than offset by a reduction in effort in projects' final phases
- although empirical research has so far failed to confirm this, veteran practitioners report that TDD leads to improved design qualities in the code, and more generally a higher degree of “internal” or technical quality, for instance improving the metrics of cohesion and coupling

<https://www.agilealliance.org/>

TDD: Common Pitfalls 1/2

Typical individual mistakes include:

- forgetting to run tests frequently
- writing too many tests at once
- writing tests that are too large or coarse-grained
- writing overly trivial tests, for instance omitting assertions
- writing tests for trivial code, for instance accessors

<https://www.agilealliance.org/>

TDD: Common Pitfalls 2/2

Typical individual mistakes include:

- partial adoption – only a few developers on the team use TDD
- poor maintenance of the test suite – most commonly leading to a test suite with a prohibitively long running time
- abandoned test suite (i.e. seldom or never run) – sometimes as a result of poor maintenance, sometimes as a result of team turnover

<https://www.agilealliance.org/>

TDD: Signs of Use

- “code coverage” is a common approach to evidencing the use of TDD; while high coverage does not guarantee appropriate use of TDD, coverage below 80% is likely to indicate deficiencies in a team’s mastery of TDD
- version control logs should show that test code is checked in each time product code is checked in, in roughly comparable amounts

<https://www.agilealliance.org/>

Skill Levels: Beginner

- able to write a unit test prior to writing the corresponding code
- able to write code sufficient to make a failing test pass

<https://www.agilealliance.org/>

Skill Levels: Intermediate

- practices “test driven bug fixing”: when a defect is found, writes a test exposing the defect before correction
- able to decompose a compound program feature into a sequence of several unit tests to be written
- knows and can name a number of tactics to guide the writing of tests (for instance “when testing a recursive algorithm, first write a test for the recursion terminating case”)
- able to factor out reusable elements from existing unit tests, yielding situation-specific testing tools

<https://www.agilealliance.org/>

Skill Levels: Advanced

- able to formulate a “roadmap” of planned unit tests for a macroscopic features (and to revise it as necessary)
- able to “test drive” a variety of design paradigms: object-oriented, functional, event-drive
- able to “test drive” a variety of technical domains: computation, user interfaces, persistent data access...

<https://www.agilealliance.org/>

Plan

- 1 CMake
- 2 Unit Testing with Google Test
- 3 Test Fixtures
- 4 Test-driven development (TDD)
- 5 TDD = A Simple Case Study

First Step 1/3

- From Wikipedia:

In mathematics, the greatest common divisor (gcd) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers. For example, the gcd of 8 and 12 is 4.

- ➊ The definition already gives us a test, which we include in the file [./tests/gcd-unittest.cc](#)

```
TEST(GcdTest, Positive) {  
    EXPECT_EQ(4, Gcd(8,12));  
}
```

- ➋ add the file [gcd-unittest.cc](#) in [./tests/CMakeList.txt](#)
- ➌ add in [./src/myfsttstprj.h](#) the prototype of Gcd:

```
int Gcd(int a, int b);
```

First Step 2/3

- ④ add to the `./src/` folder the `gcd.cc` file in which we write the empty GCD function:

```
int Gcd(int a, int b) {  
    // nothing  
}
```

- ⑤ add the file `gcd.cc` in `./src/CMakeList.txt` and comment `isprime_unittest.cc` (for optimisation)
- ⑥ build the binaries
- ⑦ Ignore the warning

First Step 2/3

⑧ test:

```
Running main() from gtest_main.cc
[=====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from GcdTest
[ RUN      ] GcdTest.Positive
.../myfstattstprj/tests/gcd_unittest.cc:88: Failure
Expected equality of these values:
        4
Gcd(8,12)
    Which is: 1
[ FAILED  ] GcdTest.Positive (0 ms)
[-----] 1 test from GcdTest (0 ms total)

[-----] Global test environment tear-down
[=====] 1 test from 1 test case ran. (1 ms total)
[ PASSED ] 0 tests.
[ FAILED  ] 1 test, listed below:
[ FAILED ] GcdTest.Positive

1 FAILED TEST
```

First Refinement 1/3

- ① add the Euclid algorithm:

```
int Gcd(int a, int b) {  
    while (a != b)  
        if (a > b)  
            a = a - b;  
        else  
            b = b - a;  
    return a;  
}
```

First Refinement 2/3

② build and test:

```
Running main() from gtest_main.cc
[=====] Running 1 test from 1 test case.
[-----] Global test environment set-up.
[-----] 1 test from GcdTest
[ RUN      ] GcdTest.Positive
[       OK ] GcdTest.Positive (0 ms)
[-----] 1 test from GcdTest (0 ms total)
[-----] Global test environment tear-down
[=====] 1 test from 1 test case ran. (0 ms total)
[ PASSED ] 1 test.
```

First Refinement 3/3

- ③ add more positive tests:

```
TEST(GcdTest, Positive) {  
    EXPECT_EQ(4, Gcd(8,12));  
    EXPECT_EQ(7, Gcd(28,21));  
    EXPECT_EQ(1, Gcd(23,31));  
    EXPECT_EQ(1, Gcd(48,17));  
}
```

- ④ build and test:

```
Running main() from gtest_main.cc  
[=====] Running 1 test from 1 test case.  
[-----] Global test environment set-up.  
[-----] 1 test from GcdTest  
[ RUN ] GcdTest.Positive  
[ OK ] GcdTest.Positive (0 ms)  
[-----] 1 test from GcdTest (0 ms total)  
  
[-----] Global test environment tear-down  
[=====] 1 test from 1 test case ran. (0 ms total)  
[ PASSED ] 1 test.
```

Second Refinement 1/3

① add trivial tests

```
TEST(GcdTest, Trivial) {
    EXPECT_EQ(18, Gcd(18,0));
    EXPECT_EQ(24, Gcd(0,24));
    EXPECT_EQ(19, Gcd(19,19));
    EXPECT_EQ(0, Gcd(0,0)); // undefined
}
```

② build and generate

```
Running main() from gtest_main.cc
[==========] Running 2 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 2 tests from GcdTest
[ RUN      ] GcdTest.Positive
[       OK ] GcdTest.Positive (0 ms)
[ RUN      ] GcdTest.Trivial
^C
```

Oooops! The execution does not terminate ...

Second Refinement 2/3

- ③ proceed by elimination and see that $\text{Gcd}(18,0)$ is the problem
- ④ analyze the problem:

```
int Gcd(int a, int b) {  
    while (a != b)  
        if (a > b)  
            a = a - b; // if b == 0, a is not modified!!!  
        else  
            b = b - a;  
    return a;  
}
```

- ⑤ fix it:

```
int Gcd(int a, int b) {  
    if (b == 0) return a;  
    if (a == 0) return b;  
    while (a != b)  
        if (a > b)  
            a = a - b;  
        else  
            b = b - a;  
    return a;  
}
```

Second Refinement 3/3

⑥ build and test it:

```
Running main() from gtest_main.cc
[=====] Running 2 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 2 tests from GcdTest
[RUN      ] GcdTest.Positive
[OK      ] GcdTest.Positive (0 ms)
[RUN      ] GcdTest.Trivial
[OK      ] GcdTest.Trivial (0 ms)
[-----] 2 tests from GcdTest (0 ms total)

[-----] Global test environment tear-down
[=====] 2 tests from 1 test case ran. (1 ms total)
[PASSED ] 2 tests.
```

Third Refinement 1/3

① add negative tests

```
TEST(GcdTest, Negative) {  
    EXPECT_EQ(6, Gcd(18, -12));  
    EXPECT_EQ(4, Gcd(-28, 32));  
    EXPECT_EQ(1, Gcd(-29, -37));  
}
```

② build and generate

```
Running main() from gtest_main.cc  
[=====] Running 3 tests from 1 test case.  
...  
.../gcd_unittest.cc:104: Failure  
Expected equality of these values:  
 6  
Gcd(18, -12)  
  Which is: -2147483638  
^C
```

Oooops! The execution does not terminate again ...

Third Refinement 2/3

Back to the documentation:

" $d \mid a$ if and only if $d \mid -a$. Thus, the fact that a number is negative does not change its list of positive divisors relative to its positive counterpart. ... Therefore, $\text{GCD}(a, b) = \text{GCD}(|a|, |b|)$ for any integers a and b , at least one of which is nonzero."

- ③ include the case when the numbers are negative:

```
int Gcd(int a, int b) {
    if (b == 0) return a;
    if (a == 0) return b;
    if (a < 0) a = -a;
    if (b < 0) b = -b;
    while (a != b)
        if (a > b)
            a = a - b;
        else
            b = b - a;
    return a;
}
```

Third Refinement 3/3

④ build and test it:

```
Running main() from gtest_main.cc
[=====] Running 3 tests from 1 test case.
[-----] Global test environment set-up.
[-----] 3 tests from GcdTest
[ RUN      ] GcdTest.Positive
[         OK ] GcdTest.Positive (0 ms)
[ RUN      ] GcdTest.Trivial
[         OK ] GcdTest.Trivial (0 ms)
[ RUN      ] GcdTest.Negative
[         OK ] GcdTest.Negative (0 ms)
[-----] 3 tests from GcdTest (0 ms total)

[-----] Global test environment tear-down
[=====] 3 tests from 1 test case ran. (0 ms total)
[ PASSED ] 3 tests.
```

Conclusion

- TDD, like any other software development concept or method, requires practice.
- The more you use TDD, the easier TDD becomes.
- Don't forget to keep the tests simple.
- The tests, which are simple, are easy to understand and easy to maintain.
- Tools like Google Test, Google Mock, CppUnit, JustCode, JustMock, NUnit and Ninject are important and help facilitate the practice of TDD.
- But it is good to know that TDD is a practice and a philosophy that goes beyond the use of tools.
- Experience with tools and frameworks is important, the skills acquired will inspire confidence in the application developer.
- But it is good to know that the tools should not be the center of attention.
- Equal time must be given to the idea of "test first".

OOP (C++): OO Modeling & Design

Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

Object Oriented Programming 2020/2021

1 Models

2 Modeling with UML Class Diagram

3 Object Oriented Design Principles

The Single-Responsibility Principle (SRP)

The Open-Closed Principle (CPP)

The Liskov Substitution Principle

Dependency Inversion Principle

4 Conclusion

Plan

1 Models

2 Modeling with UML

Class Diagram

3 Object Oriented Design Principles

The Single-Responsibility Principle (SRP)

The Open-Closed Principle (CPP)

The Liskov Substitution Principle

Dependency Inversion Principle

4 Conclusion

A Kind of Definition

A **model** is an abstraction of the software system to be developed that makes the implementation (programming) of the system easier.

A model has three main characteristics (Herbert Stachowiak, 1973):

- *Mapping*: a model is always an image (mapping) of something.
- *Reduction*: a model does not capture all attributes of the original.
- *Pragmatism*: a model should be useful.

Quality of a Model

It is determined by the following characteristics (Bran Selic, 2003):

- Abstraction
- Understandability
- Accuracy
- Predictiveness
- Cost-effectiveness

UML

- To write a model, you need a modeling language.
- UML (Unified Modeling Language) is a language and modeling technique suitable for object-oriented programming UML is used to view, specify, build, and document object-oriented systems
- In this course we will use UML elements to explain the concepts and laws of OOP.
- Free soft tools: BOUML, Argouml (open source), Visual Paradigm UML (Online Express Edition),...

Plan

1 Models

2 Modeling with UML

Class Diagram

3 Object Oriented Design Principles

The Single-Responsibility Principle (SRP)

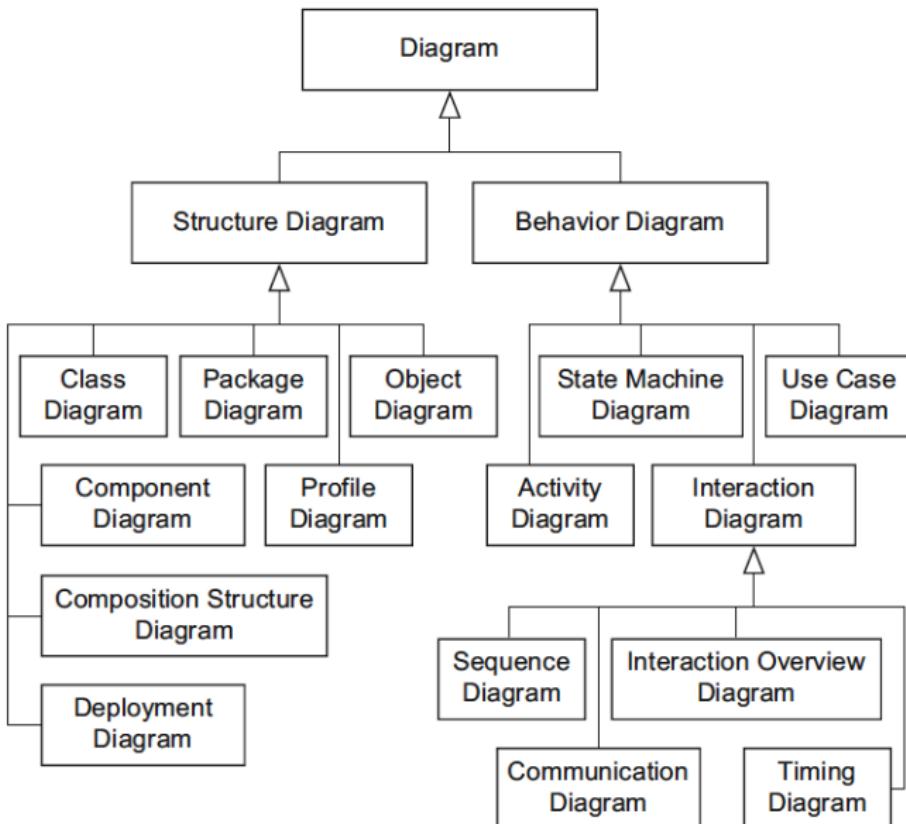
The Open-Closed Principle (CPP)

The Liskov Substitution Principle

Dependency Inversion Principle

4 Conclusion

UML Diagrams



Plan

1 Models

2 Modeling with UML
Class Diagram

3 Object Oriented Design Principles
The Single-Responsibility Principle (SRP)
The Open-Closed Principle (CPP)
The Liskov Substitution Principle
Dependency Inversion Principle

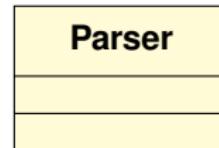
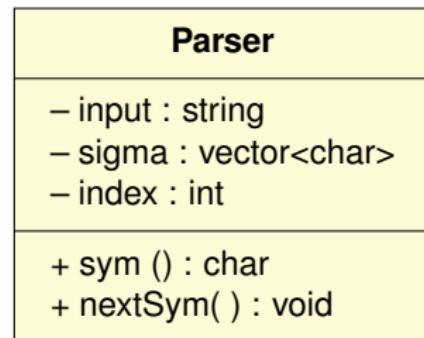
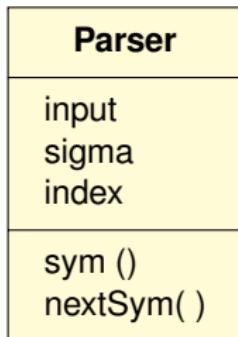
4 Conclusion

Specification of a Class in UML

```
class Parser {  
private:  
    string input;  
    vector<char> sigma;  
    int index;  
public:  
    char sym();  
    void nextSym();  
    ...  
};
```

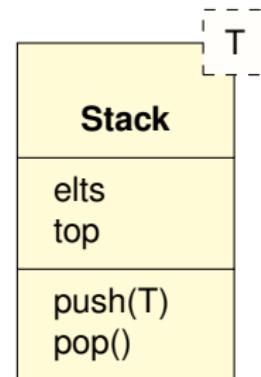
Parser
– input : string – sigma : vector<char> – index : int
+ sym () : char + nextSym() : void

Notation



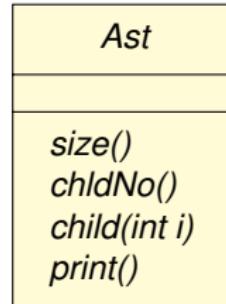
Parametrized Class

```
template<class T>
class Stack {
private:
    T* elts;
    int top;
public:
    void push(T);
    void pop();
    ...
};
```



Abstract Class

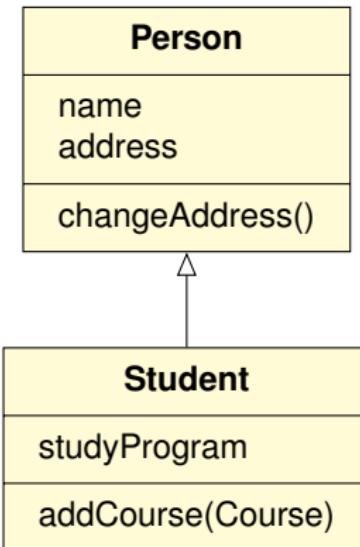
```
class Ast {  
public:  
    virtual int size() = 0;  
    virtual int chldNo() = 0;  
    virtual Ast* child(int i) = 0;  
    virtual void print() = 0;  
};
```



Note the use of the italic font.

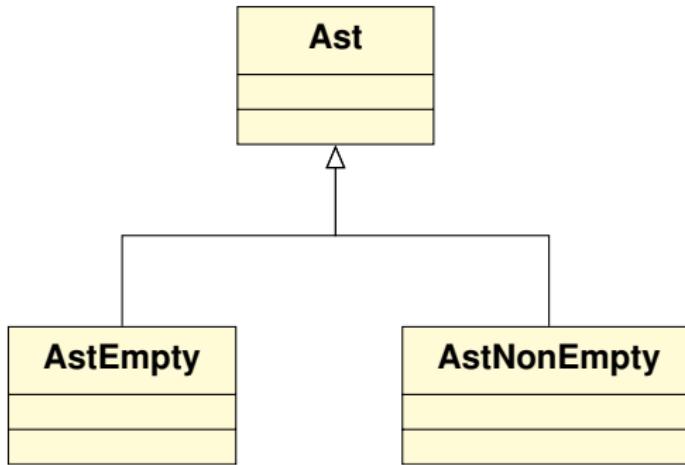
Generalization: Inheritance

```
class Person {  
    string name;  
    string address;  
public:  
    void changeAddress();  
};  
class Student : public Person {  
    list<Course> studyProgram;  
public:  
    void addCourse(Course);  
};
```



Generalization: Classification

```
class AstEmpty : public Ast {  
    ...  
};  
class AstNonEmpty : public Ast {  
    ...  
};
```

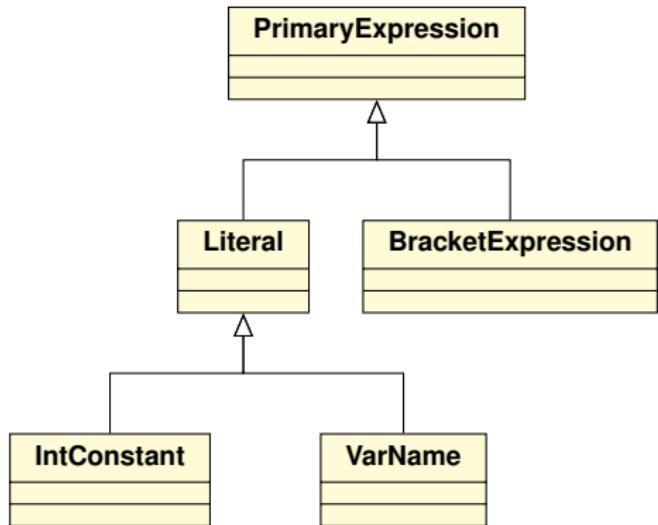


Classification: A Simple Case Study

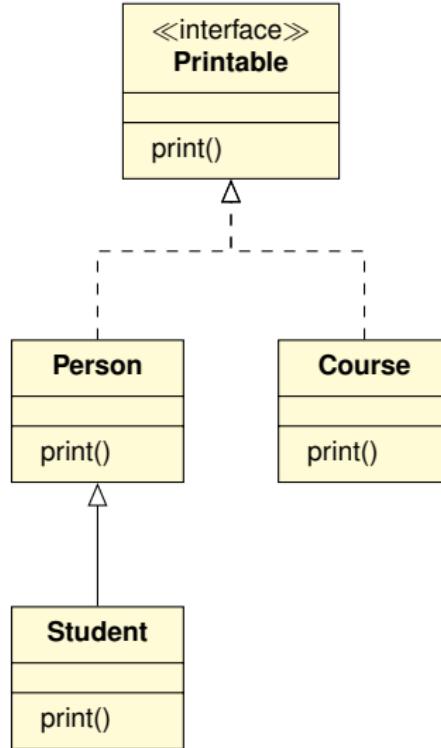
Frm C++ Grammar (simplified)
literal:

```
int-constant  
var-name
```

primary-expression:
literal
(expression)



Interface



Interfaces in C++

Are described using abstract classes:

```
class Printable {  
public: virtual void print() = 0;  
};  
  
class Person : public Printable {  
public: virtual void print();  
}  
...  
void Person::print() {  
    ...  
}
```

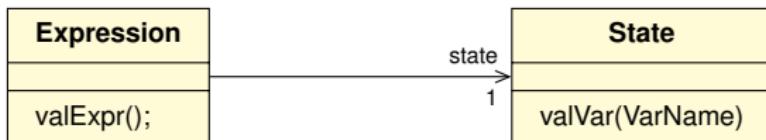
Bidirectional Binary Associations



```
class Professor {  
    private: list<Course*> courses;  
    ...  
}  
class Course {  
    private: Profesor* lecturers[2];  
    ...  
}
```

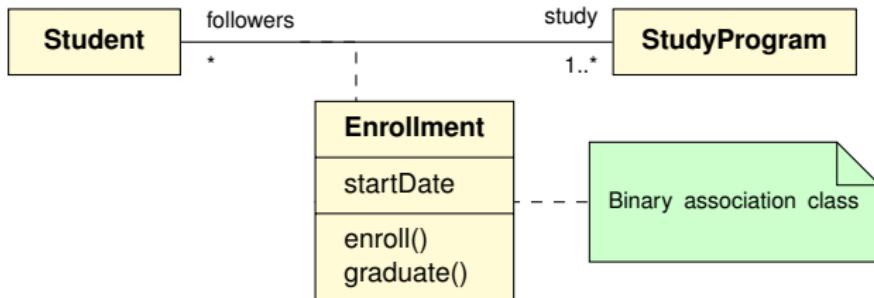
Unidirectional Binary Associations

- an expression is evaluated in a state (the expression knows the state it is evaluated)
- the state does not know for which expression it is used for evaluation



```
class Expression {  
private: State* state;  
public: int valExpr(); // uses state->valVar()  
...  
}  
class State {  
public : int valVar(VarName vn);  
// returns the value of the variable vn  
...  
}
```

Association Class



```
class Enrollment {  
    // class invariant: study != empty  
private:  
    list<Student*> followers;  
    list<StudyProgram*> study;  
    Date statDate;  
public:  
    void enroll(Student, StudyProgram);  
    void graduate(Student, StudyProgram);  
    ...  
};
```

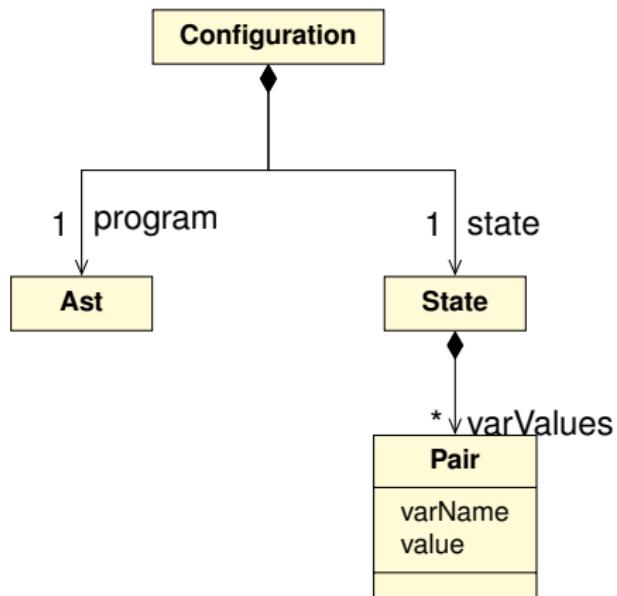
Shared Aggregation



```
class StudyProgram {  
    private: vector<Course*> courses;  
    ...  
};  
class Course {  
    // class invariant: includedIn != empty  
    private: list<StudyProgram*> curriculum;  
    ...  
};
```

Strong Aggregation (Composition)

configuration = ⟨ program, state ⟩
state = var1 ↠ val1, var2 ↠ val2, ...



```
class Configuration {  
    private:  
        Ast program;  
        State state;  
        ...  
};  
class State {  
    private:  
        map<VarName, Value> varValues;  
        ...  
};
```

Plan

1 Models

2 Modeling with UML
Class Diagram

3 Object Oriented Design Principles

The Single-Responsibility Principle (SRP)
The Open-Closed Principle (CPP)
The Liskov Substitution Principle
Dependency Inversion Principle

4 Conclusion

Plan

1 Models

2 Modeling with UML
Class Diagram

3 Object Oriented Design Principles

The Single-Responsibility Principle (SRP)

The Open-Closed Principle (CPP)

The Liskov Substitution Principle

Dependency Inversion Principle

4 Conclusion

Definition

Single-Responsibility Principle

A class should have one and only one reason to change, meaning that a class should have only one job.

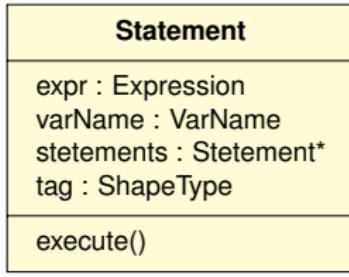
Robert C. Martin. Principles of Object Oriented Design.

https://drive.google.com/file/d/0ByOwmqah_nuGNHEtcU5OekdDMkk/view

From C++ grammar (modified)

statement:

```
var-name = expression ;      //asignement
{ statement }                // compound statetement
(statement)*                 //sequential composition
if ( expression ) statement  // if
if ( expression ) statement else statement //if
...
```



A green callout box containing a snippet of C-like pseudocode. The code handles different tags for execution:

```
if (tag == assign)
    execAssign()
else if (tag == compound)
    execCompound()
else if (tag == seq)
    ...

```

Too many responsibilities.

From C++ grammar (simplified)

statement:

- assign-statement
- compound-statement
- statement-seq
- selection-statement

assign-statement:

- var-name = expression ;

compound-statement:

- { statement }

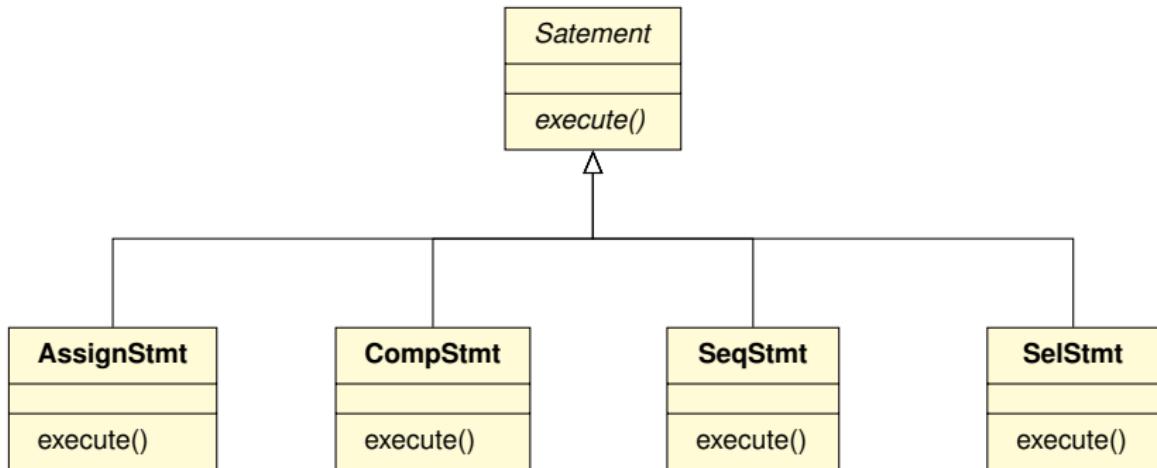
statement-seq:

- statement
- statement-seq statement

selection-statement:

- if (condition) statement
- if (condition) statement else statement

SRP 2/2



Plan

1 Models

2 Modeling with UML
Class Diagram

3 Object Oriented Design Principles

The Single-Responsibility Principle (SRP)

The Open-Closed Principle (CPP)

The Liskov Substitution Principle

Dependency Inversion Principle

4 Conclusion

Definition

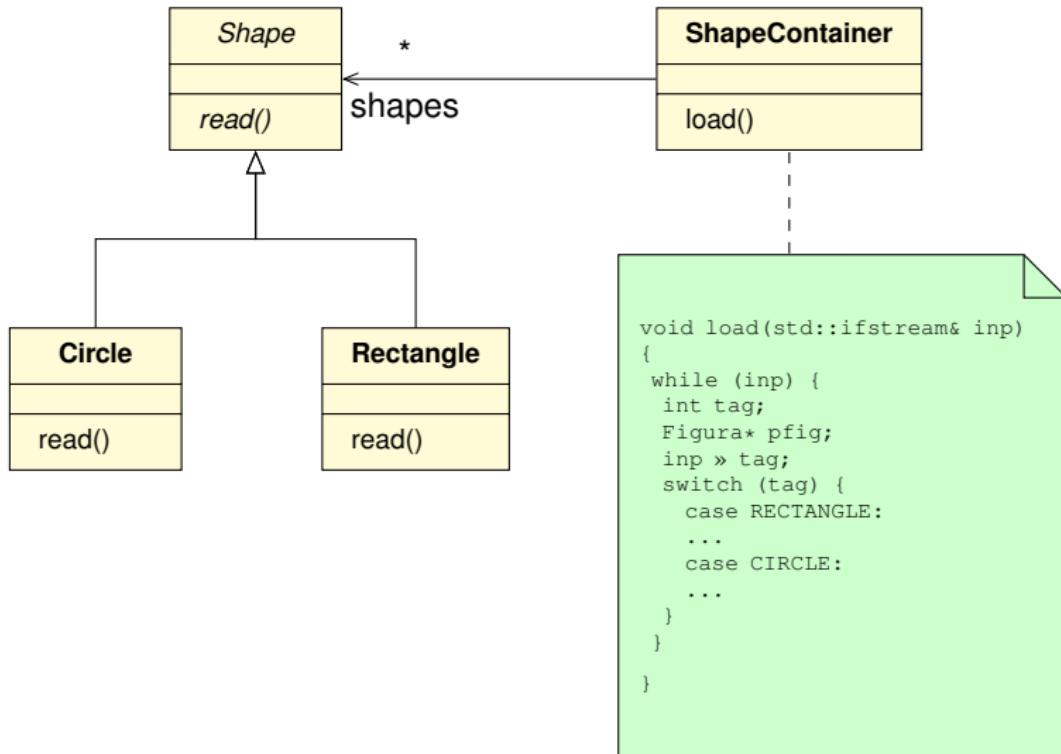
Open-Closed Principle

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Robert C. Martin. Principles of Object Oriented Design.

<https://drive.google.com/file/d/>

0BwhCYaYDn8EgN2M5MTkwM2EtNWFkZC00ZTI3LWFjZTUtNTFhZGZiYmUzODc1/view



If we add more shapes, we have to modify `ShapeContainer::load()`.

OCP

The previous example can be fixed to respect OCP using Object Factory Pattern (next lecture).

Plan

1 Models

2 Modeling with UML
Class Diagram

3 Object Oriented Design Principles

The Single-Responsibility Principle (SRP)

The Open-Closed Principle (CPP)

The Liskov Substitution Principle

Dependency Inversion Principle

4 Conclusion

Definition

Liskov¹ Substitution Principle

Let $P(x)$ be a property provable about objects x of type T .

Then $P(y)$ should be true for objects y of type S , where S is a subtype of T .

In other words, S is a subtype of T if anywhere you can use a T , you could also use an S (a T can be substituted by an S).

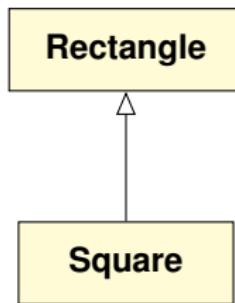
NB Here the notion of "subtype" should be meant as "behavioural subtype" (we will explain later the difference).

Barbara Liskov, Jeannette M. Wing: A Behavioral Notion of Subtyping. ACM Trans. Program. Lang. Syst. 16(6): 1811-1841 (1994)

¹Turing Award, 2008, for contributions to programming languages development, especially OO languages.

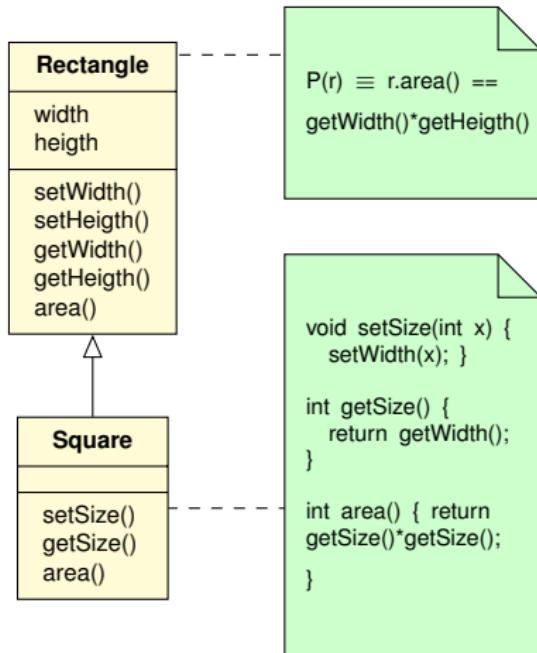
A Counter-Example 1/4

A rushed modeling of the relation "Any square is a rectangle" (this is an instance of the "is a" relation):



A Counter-Example 2/4

Let's refine it:



A Counter-Example 3/4

Consider the code:

```
Square s;  
s.setWidth(5);           // inherited  
s.setHeight(10);        // inherited
```

Assume that `s` plays the role of a `Rectangle`.

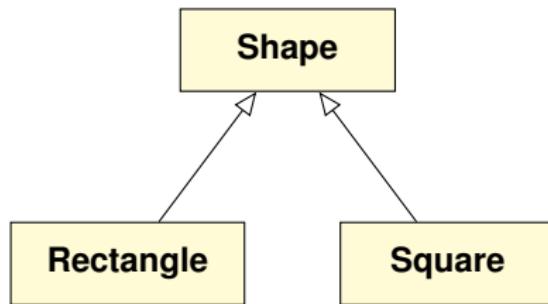
Does it satisfy the property $P(s)$? No.

Question

What is the right relationship between "subclass" and "subtype"?

A Counter-Example 4/4

A correct hierarchy:



Question

It seems that the **subclass** relation, given by
(child-class, parent-class)

is not the same with that of **subtype**.

So,

What is the right relationship between "subclass" and "subtype"?

Types, Types, Types, ... 1/3

From (online) manuals² (partial):

Objects, references, functions including function template specializations, and expressions have a property called type, which both restricts the operations that are permitted for those entities and provides semantic meaning to the otherwise generic sequences of bits.

Type classification

The C++ type system consists of the following types:

fundamental types

void

signed integer types int, long int, ...

floating-point types float, double, ...

...

compound type

reference type

pointer type

array types

function types

class types

...

²<https://en.cppreference.com/w/cpp/language/type> ▶ ⏪ ⏩ ⏴ ⏵ ⏵ ⏵

Types, Types, Types, ... 2/3

So, a class is a type?

No, not completely true.

A class includes two orthogonal things:

static/behavioral type
implementation

Types, Types, Types, ... 3/3

Static types (abstract view):

Declaration of X	Type associated to X	Remark
<code>int X;</code>	<code>int</code>	
<code>struct X {</code> <code> int a;</code> <code> double b;</code> }	<code>int × double</code>	product type
<code>int X(double a);</code>	<code>double → int</code>	function type
<code>class X {</code> <code> int size;</code> <code>public:</code> <code> int getSize();</code> <code> void setSize(int a);</code> }	<code>void → X</code> <code>X → int</code> <code>X × int → X</code>	<code>constructor X()</code> <code>getSize</code> <code>setSize</code>

For classes, only the types of the public members (interface) are considered.

Static Subtype 1/2

A type T consists of:

- a set of possible values which a variable/expression can possess during program execution
- a set of operations/functions that can apply values

S is a subtype of T , $S <: T$ if:

- S values "are" T values
- the operations/functions applied to T can also be applied to S preserving semantics

Static Subtype 2/2

Examples: `int <: long int`

We do not have `int <: float` because `_/_` on `int` has a different semantics from that of `float`.

For classes definition becomes more complex.

On Contravariance/Covariance/Invariance

Within a type system, the variance refers how the subtype relation is promoted to compound types.

- a rule is covariant if preserves the ordering of types $<:$
- a rule is contravariant if it reverses this ordering $<:$
- a rule is invariant if it is not neither covariant nor cotravariant

E.g., the rule for function types is contravariant on arguments and covariant on result:

$$\frac{A' <: A \quad B <: B'}{(A \rightarrow B) <: (A' \rightarrow B')}$$

Contravariance/Covariance/Invariance in C++

C++ for overridden functions is invariant on parameters type and covariant on result type.

From the manual:

The return type of an overriding function shall be either identical to the return type of the overridden function or covariant with the classes of the functions. If a function D::f overrides a function B::f, the return types of the functions are covariant if they satisfy the following criteria:

- (8.1) — both are pointers to classes, both are lvalue references to classes, or both are rvalue references to classes
- (8.2) — the class in the return type of B::f is the same class as the class in the return type of D::f, or is an unambiguous and accessible direct or indirect base class of the class in the return type of D::f
- (8.3) — both pointers or references have the same cv-qualification and the class type in the return type of D::f has the same cv-qualification as or less cv-qualification than the class type in the return type of B::f.

C++: Return Covariant Example 1

Not really:

```
class Base
{
public:
    Base() = default;
    virtual Base* clone() const { return new Base; }
    virtual ~Base() { std::cout << "~Base()\n"; }
};

class Derived : public Base
{
public:
    Derived () = default;
    Derived* clone() const override { return new Derived; }
    ~Derived() override { std::cout << "~Derived()\n"; }
};
```

C++: Return Covariant Example 2

```
class Animal {
public:
    virtual void makeNoise() = 0;
    virtual ~Animal() = default;
};

class Dog : public Animal {
public:
    void makeNoise() override { std::cout << "ham!\n"; }
};

class AnimalBreeder {
public:
    virtual Animal* produce() = 0;
};

class DogBreeder : public AnimalBreeder {
public:
    Dog* produce() override { return new Dog(); }
};
```

C++: Argument Contravariant (Contra)Example

```
class Animal {
public:
    virtual void makeNoise() = 0;
    virtual ~Animal() = default;
};

class Dog : public Animal {
public:
    void makeNoise() override { std::cout << "ham!\n"; }
};

class DogDoctor {
public:
    virtual void treat(Dog *dog) { std::cout << "Dog treated!\n"; }
};

class AnimalDoctor : public DogDoctor {
public:
    void treat(Animal * animal) override { std::cout << "Animal treated!\n"; } // error
};
```

Remember that C++ is invariant on arguments.

Behavioral Type of a Class 1/3

Consider the following simple example:

```
class Square {
    int size;
public:
    Square(int s = 0);
    int getSize();
    int area() throw(overflow_error);
    void halve();
}
```

Behavioral Type of a Class 2/3

Includes also the behavioral interface of the objects and consists of:

- a description of the value space

- * class invariants

Example: `getSize() ≥ 0`

- * for each constructor c

- its signature

Example: `Square : int → Square`

- precondition

Example: $s \geq 0$

- postcondition

Example: `getSize() ≥ 0`

- the exceptions it signals

Behavioral Type of a Class 3/3

- for each public method m :

- * its signature

Example: `area : Square → int`

`halve : Square → Square`

- * precondition

Example: `even(getSize())` (for `halve`)

- * postcondition

Example: `area() == 0.25 * old(area())` (for `halve`)

- * the exceptions it signals

- for each public attribute a :

- * its type

A Possible Specification for the Behavioral Type

```
type Square {
    invariant getSize() >= 0
    Square(int s = 0)
        requires s >= 0
        ensures true
        noexcept(true);
    int getSize()
        requires true
        ensures true
        noexcept(true);
    int area()
        requires true
        ensures result == getSize()*getSize()
        throw(overflow_error)
    void halve();
        requires even(getSize())
        ensures area() == 0.25*old(area())
        noexcept(true);
}
```

The Substitution Principle Explained using Behavioral Types 1/2

A child class (subclass) B is a behavioral subtype of the parent class (superclass) A , $B <: A$, if:

- the invariants of the superclass are preserved by the subclass (B values are A values): the conjunction of the B invariants implies conjunction of the A invariants, i.e.,

$$\bigwedge_{\phi \in Inv(B)} \phi \implies \bigwedge_{\phi \in Inv(A)} \phi$$

- for each overridden method m
 - $B::m()$ and $A::m()$ have the same number of arguments
 - contravariance of arguments**: if the i -th argument of $B::m()$ is β_i and the i -th argument of $A::m()$ is α_i , then $\alpha_i <: \beta_i$
 - covariance of result**: if the result type of $A::m()$ is α and the result type of $B::m()$ is β , then $\beta <: \alpha$

The Substitution Principle Explained using Behavioral Types 2/2

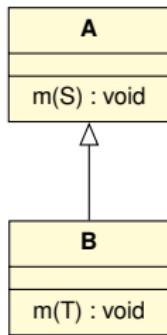
- exception rule: the set of exceptions signaled by $B::m()$ is included in the set of exceptions signaled by $A::m()$
- semantics is preserved:
 - the precondition of $A::m()$ implies the precondition of $B::m()$
 - the postcondition of $B::m()$ implies the postcondition of $A::m()$

N.B. This is a simplified version of the original definition.

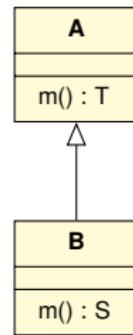
Exercise. Show that $T <: T$ ($<:$ is reflexive).

Contravariance/Covariance Rule with Diagrams

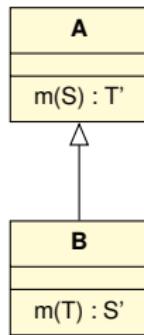
$S <: T$
 $S' <: T'$



Contravariance
of arguments

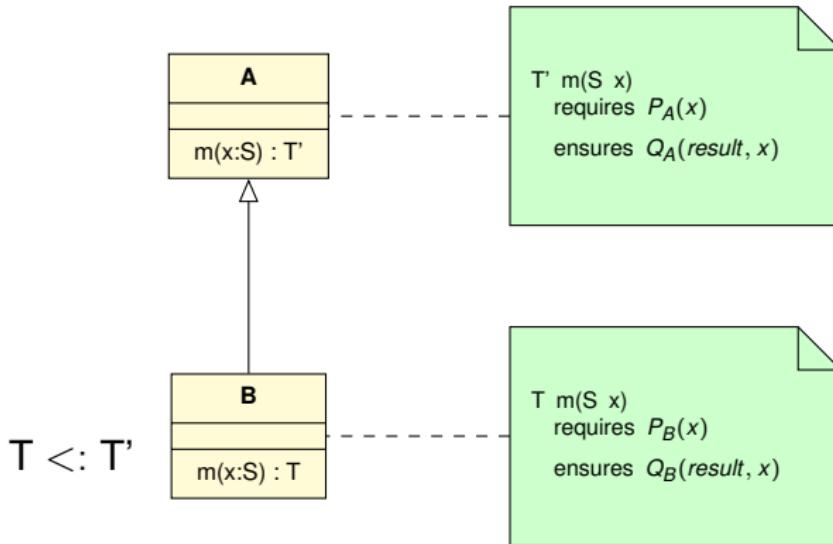


Covariance
of result



Contravariance
of arguments and
Covariance
of result

Semantics Rule with Diagrams



$$P_A \implies P_B$$

$$Q_B \implies Q_A$$

Contra-Example Revisited

The example "any square is a rectangle" violates several rules:

- the method `Square::setHeigth()` does not preserve the `Square` invariant `getWidth() = getHeigth()`
- the postcondition of `Square::area()` does not imply the postcondition of `Rectangle::area()`

Plan

1 Models

2 Modeling with UML
Class Diagram

3 Object Oriented Design Principles

The Single-Responsibility Principle (SRP)

The Open-Closed Principle (CPP)

The Liskov Substitution Principle

Dependency Inversion Principle

4 Conclusion

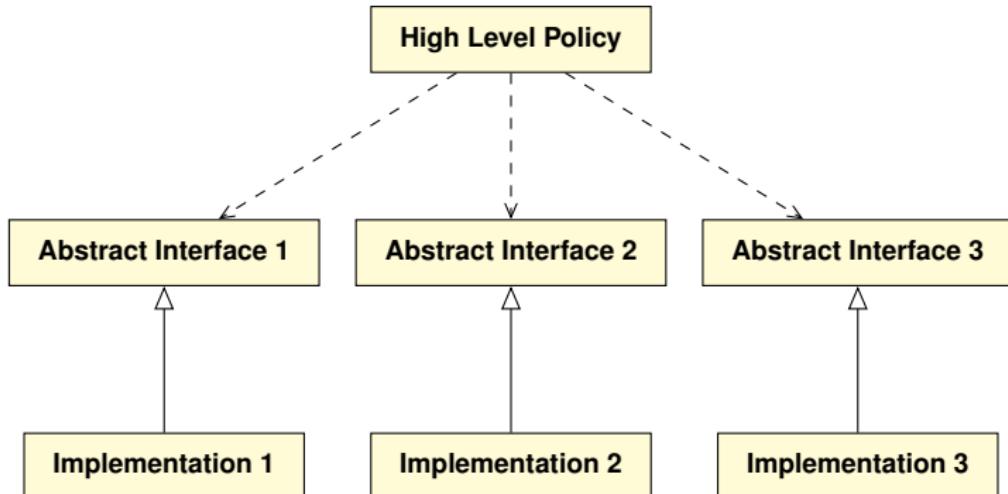
Definition

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

Robert C. Martin. Principles of Object Oriented Design.

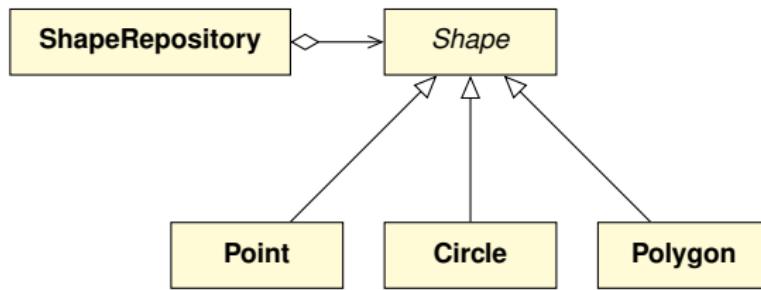
https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf

Abstract View

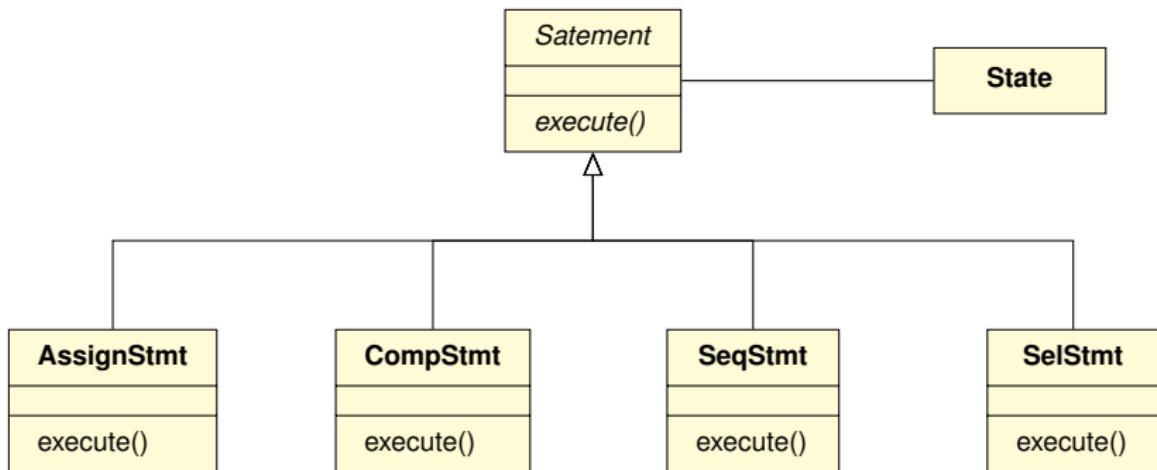


Note the notational difference between the dependence and inheritance relations. Inheritance also implies dependency.

Example



Example



Plan

1 Models

2 Modeling with UML
Class Diagram

3 Object Oriented Design Principles
The Single-Responsibility Principle (SRP)
The Open-Closed Principle (CPP)
The Liskov Substitution Principle
Dependency Inversion Principle

4 Conclusion

Conclusion

- Models first.
- Apply the right OO Design Principle wherever such a principle is applicable
- Revise your applications from previous laboratories and design their models and identify the places where principles are applicable.

OOP (C++): Design Patterns

Dorel Lucanu

Faculty of Computer Science
Alexandru Ioan Cuza University, Iași, Romania
dlucanu@info.uaic.ro

Object Oriented Programming 2020/2021

1 On Design Patterns

2 Singleton

3 Composite

Case Study: Expressions

4 Visitor

Combining Composite and Visitor

5 Object Factory

- 1 On Design Patterns
- 2 Singleton
- 3 Composite
Case Study: Expressions
- 4 Visitor
Combining Composite and Visitor
- 5 Object Factory

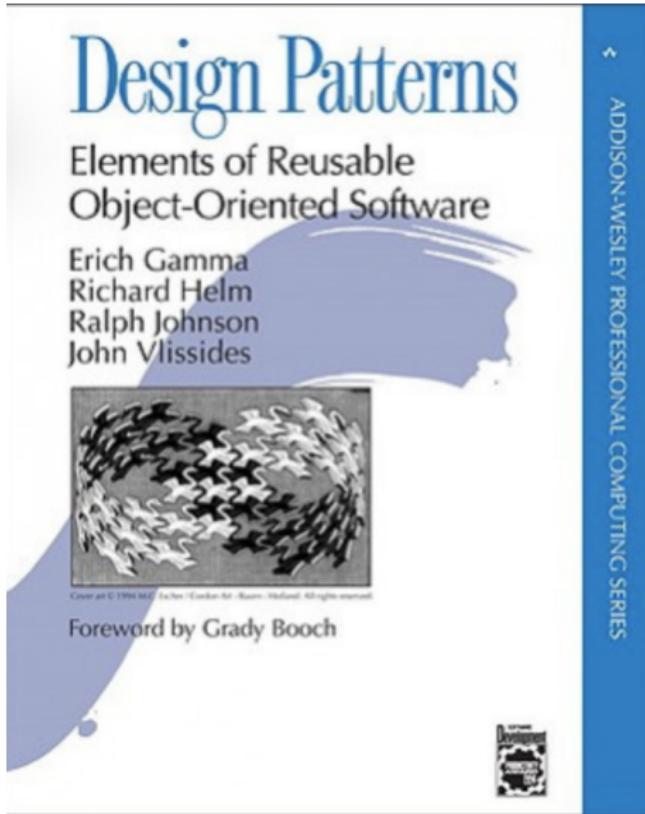
Alexander's Definition

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"¹

- applied first in urbanism architecture
- the first contributions in software: prima contributie in software: 1987, Kent Beck (creator of Extreme Programming) & Ward Cunningham (wrote the first wicki)

¹C. Alexander. A Pattern Language. 1977

GoF Book



includes 23 design patterns

Full Template for a Pattern

- name and classification
- intention
- known as
- motivation
- applicability
- structure
- participants
- collaborations
- consequences
- implementation
- code
- known use cases
- related patterns

We will use a simplified template.

Classification

- **creational**
used to create complex/specific objects
- **structural**
used to define the structure of the classes and objects
- **behavioral**
describe how the classes and their objects interact in order to distribute the responsibilities

Plan

1 On Design Patterns

2 Singleton

3 Composite

Case Study: Expressions

4 Visitor

Combining Composite and Visitor

5 Object Factory

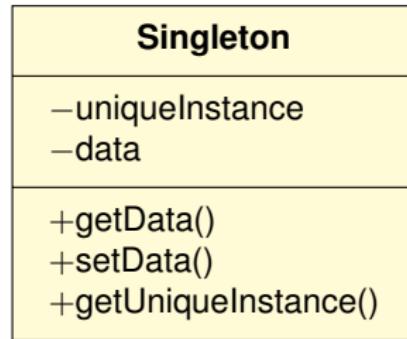
Motivation

- Classification:
creational
- Intention:
designing a class with a single object (a single instance)
- Motivation:
in an operating system:
 - there is a file system
 - there is only one window managerin a website: there is only one web page manager
- Application:
when there must be exactly one instance
class clients must have access to the instance from any well-defined point

Consequences

- controlled access to the single instance
- namespace reduction (global variable elimination)
- allows refinement of operations and representation
- allows a fixed number of instants (Doubleton, Tripleton, . . .)
- more flexible than class-level operations (static functions)

Structure



Implementation (version 1, \geq C++2011)

```
template <typename Data>
class Singleton {
public:
    static Singleton<Data>& getUniqueInstance() {
        return uniqueInstance;
    }
    Data getData();
    void setData(Data x);
    void operator=(Singleton&) = delete;
    Singleton(const Singleton&) = delete;
protected:
    Data data; // object state
    Singleton() { }
private:
    static Singleton<Data> uniqueInstance;
};

...
template <typename Data>
Singleton<Data> Singleton<Data>::uniqueInstance;
```

Testing

```
class SingletonTest : public testing::Test {
protected:
    virtual void SetUp() {
        s2.setValue(9);
        // s3 = s1; // compiling error
        s3.setValue(77);
    }
    // Declares the variables your tests want to use.
    Singleton &s1 = Singleton::instance();
    Singleton &s2 = Singleton::instance();
    Singleton &s3 = s2;
    // Singleton s4 = s2; // compiling error
};
// Tests the uniqueness.
TEST_F(SingletonTest, Uniqueness) {
    ASSERT_EQ(s1.getValue(), s2.getValue());
    ASSERT_EQ(s1.getValue(), s3.getValue());
}
```

What about the move constructor/assignment-operator?

From the manual (12.8):

"10 If the definition of a class X does not explicitly declare a move constructor, one will be implicitly declared as defaulted if and only if

- X does not have a user-declared copy constructor,
- X does not have a user-declared copy assignment operator,
- X does not have a user-declared move assignment operator,
- X does not have a user-declared destructor, and
- the move constructor would not be implicitly defined as deleted."

Does it make sense to declare a move constructor/assignment-operator?

Implementation (version 2)

```
template <typename Data>
class Singleton {
public:
    static Singleton* getInstance() {
        if (uniqueInstance == 0) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
    ...
protected:
    Data data;
    Singleton() { }
private:
    static Singleton<Data>* uniqueInstance;
};
```

Testing

```
class SingletonTest : public testing::Test {
protected:
    virtual void SetUp() {
        s1 = Singleton::instance();
        s1->setValue(47);
        s2 = Singleton::instance();
        s2->setValue(9);
        s3 = s1;
        *s3 = *s2;
        s3->setValue(75);
    }
    // Declares the variables your tests want to use.
    Singleton *s1;
    Singleton *s2;
    Singleton *s3;
};

// Tests the uniqueness.
TEST_F(SingletonTest, Uniqueness) {
    ASSERT_EQ(s1->getValue(), s2->getValue());
    ASSERT_EQ(s1->getValue(), s3->getValue());
}
```

Testing

```
[=====] Running 1 test from 1 test suite.  
[-----] Global test environment set-up.  
[-----] 1 test from SingletonTest  
[ RUN ] SingletonTest.Uniqueness  
[ OK ] SingletonTest.Uniqueness (0 ms)  
[-----] 1 test from SingletonTest (0 ms total)  
  
[-----] Global test environment tear-down  
[=====] 1 test from 1 test suite ran. (0 ms total)  
[ PASSED ] 1 test.
```

Process finished with exit code 0

Plan

1 On Design Patterns

2 Singleton

3 Composite

Case Study: Expressions

4 Visitor

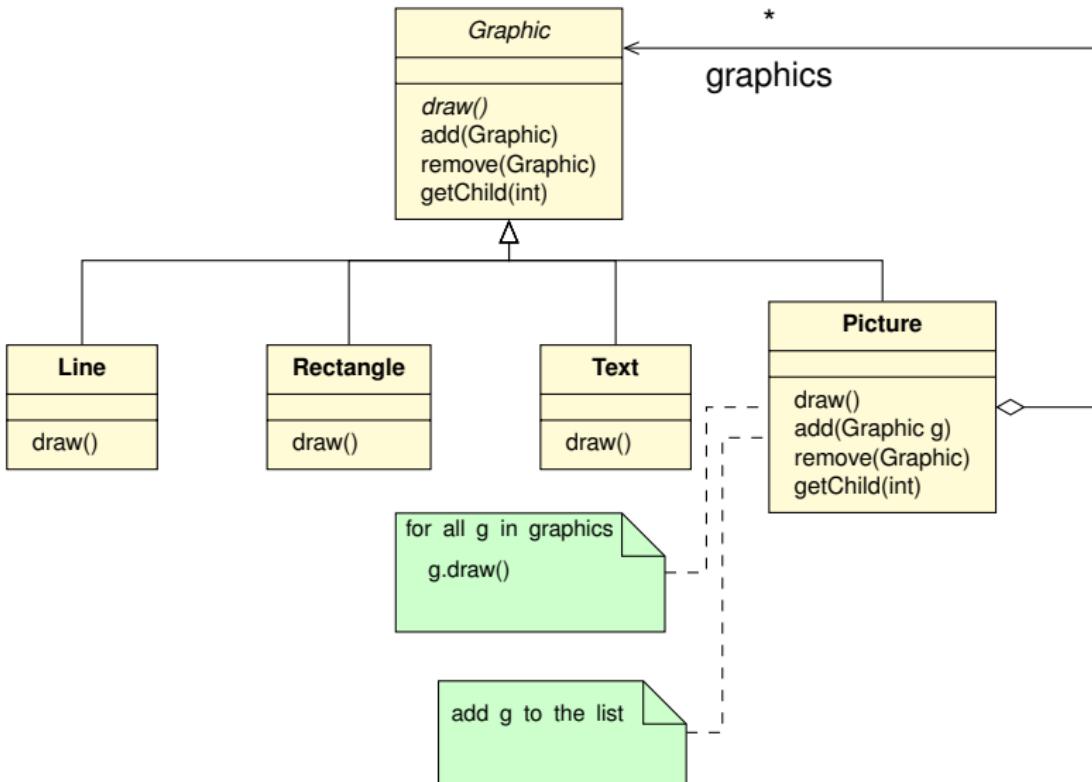
Combining Composite and Visitor

5 Object Factory

Intention

- it is a structural pattern
- composes objects in a tree structure to represent a part-whole hierarchy
- let the clients (of the structure) treat the individual and compound objects in a uniform way

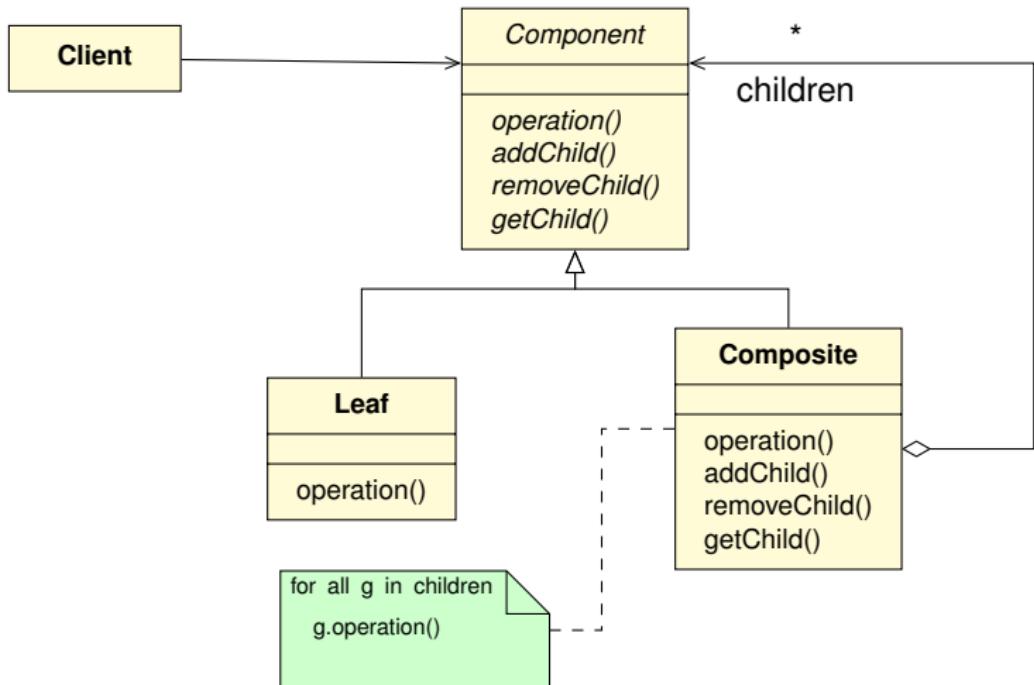
Motivation



It is a Recursive Definition

- any (object) line is a graphic object
- any (object) rectangle is a graphic object
- any text (object) is a graphic object
- a picture made up of several graphic objects is a graphic object

Structure



Participants 1/2

- Component (e.g., Graphic)
 - declares the interface for the objects in the composition
 - implements the default behavior for the common interface of all classes
 - declares an interface for accessing and managing child components
 - (optional) defines an interface for accessing parent components in the recursive structure
- Leaf (e.g., Rectangle, Line, Text, etc.)
 - represents primitive objects
 - a leaf has no children
 - defines the behavior of primitive objects

Participants 2/2

- Composite (e.g., Picture)
 - defines the behavior of components with children
 - memorizes child components
 - implements operations related to copies of the Component interface
- Client
 - handles the objects in the composition through the Component interface

Collaborations

- clients use the Component interface class to interact with objects in the structure
- if the container is a Leaf instance, then the request is resolved directly
- if the container is a Composite instance, then the request is forwarded to the child components; other additional operations are possible before or after forwarding

Consequences 1/2

- defines a hierarchy of classes consisting of primitive and compound objects
- primitive objects can be composed of more complex objects, which in turn can be composed of other more complex objects, etc. (recursion)
- whenever a client expects a primitive object, he can also take a composite object
- for the client it is very simple; it treats primitive and composite objects uniformly
- the client does not care if it has to do with a primitive or composite object (avoiding the use of switch-case structures)

Consequences 2/2

- it is easy to add new types of Leaf or Composite components; the new subclasses work automatically with the existing structure and the customer code. The customer does not change anything.
- makes the design very general
- drawback: it is difficult to restrict which components can appear in a composite object (a solution could be to check during execution)

Implementation: Decisions to Make

- explicit references to parents?
- shared components?
- maximize the interface? safety or transparency?
Transparency could lead to violation of the SRP!
Safety requires to convert a Component into a Composite!
- where to implement the operations handling children?

Plan

- 1 On Design Patterns
- 2 Singleton
- 3 Composite
Case Study: Expressions
- 4 Visitor
Combining Composite and Visitor
- 5 Object Factory

Expressions in Programming Languages

- arithmetic expressions: $a + b * 2 - c$
- relational expressions: $a + 2 < b * 3$
- Boolean expressions: $a < 3 \&\& (b < 0 \mid\mid a < b)$

Arithmetic Expressions: Syntax

```
PrimaryExpression ::=  
    IntConstant  
    | VarName  
    | "(" Expression ")"  
  
IntConstant ::=  
    Digit+  
  
Digit ::=  
    "0" | "1" | ... | "9"  
  
VarName ::=  
    "a" | "b" | ... | "z"  
  
MultExpression ::=  
    PrimaryExpression (( "*" | "/" | "%" ) PrimaryExpression)*  
  
ArithExpression ::=  
    MultExpression (( "+" | "-" ) MultExpression)*  
  
Expression ::= ArithExpression
```

AST Classes

IntConstant

VarName

PrimaryExpression

MultExpression

ArithExpression

Expression

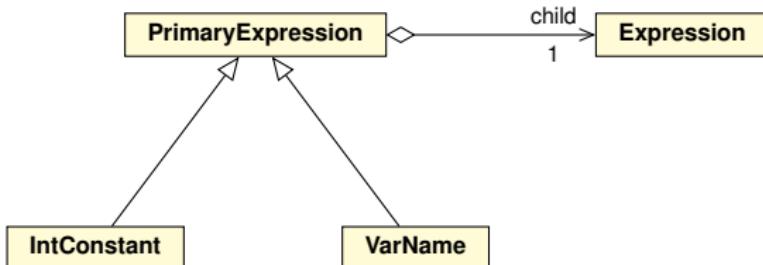
Relationship between Classes 1/3

```
PrimaryExpression ::=  
    IntConstant      =  
    | VarName  
    | "(" Expression ")"
```

```
PrimaryExpression ::=  
    IntConstant  
    | VarName
```

+

```
PrimaryExpression ::=  
    "(" Expression ")"
```



Relationship between Classes 2/3

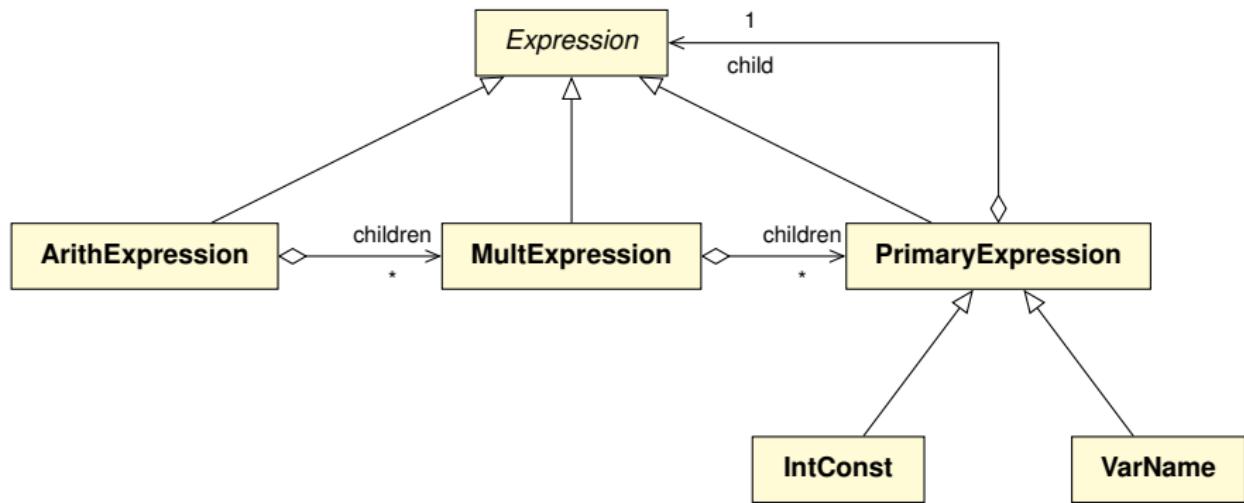
```
MultExpression ::=  
    PrimaryExpression (( "*" | "/" | "%" ) PrimaryExpression) *
```



```
ArithExpression ::=  
    MultExpression (( "+" | "-" ) MultExpression) *
```



Relationship between Classes 3/3



Ugly!

Question. What OO design principles are violated?

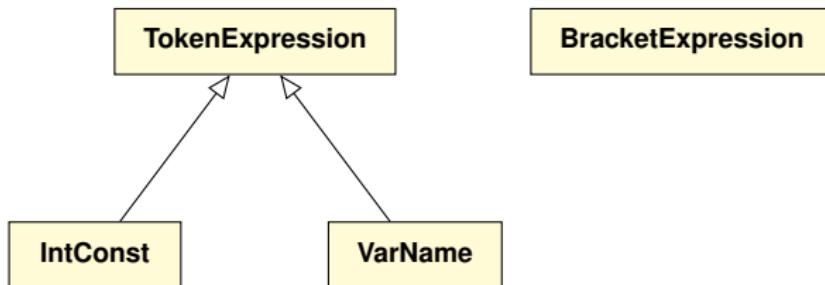
Using Composite 1/2

Leafs: IntConst, VarName

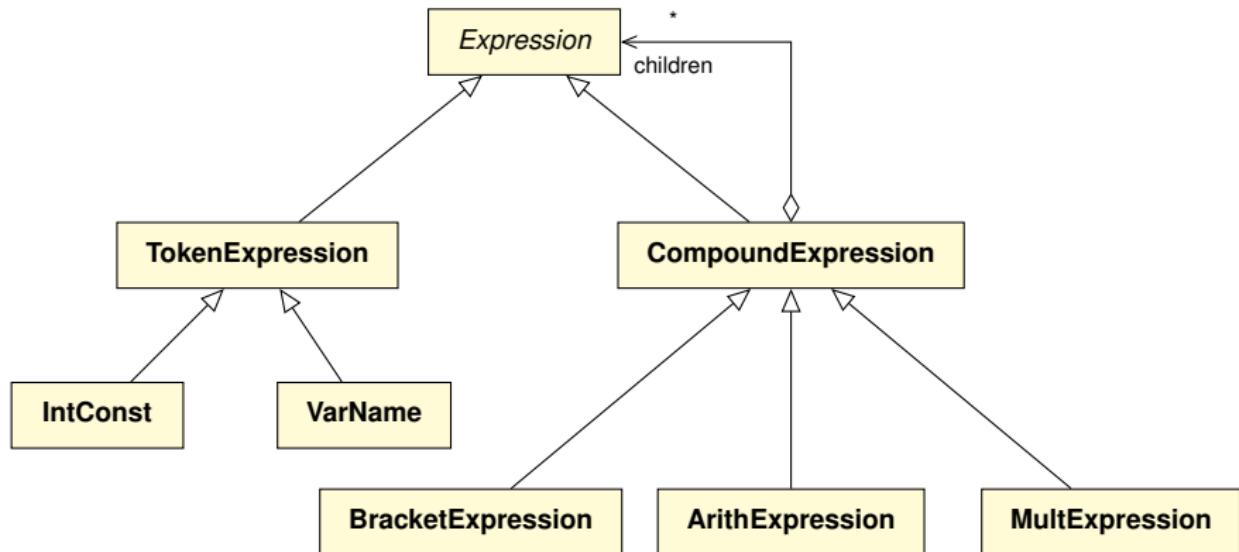
Composites: ArithExpression, MultExpression, Expression

What about PrimaryExpression? It is both

It is both, therefore we split it:



Using Composite 2/2



Class Expression in C++

```
class Expression {  
public: virtual list<string> getLabel();  
public: virtual void addLabel(string str);  
public: virtual list<string> getLabel();  
public: virtual string toString();  
protected: list<string> label;  
};
```

- we opted for safety

Class CompoundExpression in C++

```
class CompoundExpression : public Expression {  
public: void addChild(Expression* pe);  
public: string toString();  
public: list<Expression*> getChildren();  
protected: list<Expression*> children;  
};  
  
class ArithExpression : public CompoundExpression {  
};  
  
class MultExpression : public CompoundExpression {  
};  
...
```

Expressions Parser

See the appendix.

The implementation can be found in the folder
[examples/interpreter/cpp/expressions/parser-composite](#)

Plan

- 1 On Design Patterns
- 2 Singleton
- 3 Composite
Case Study: Expressions
- 4 Visitor
Combining Composite and Visitor
- 5 Object Factory

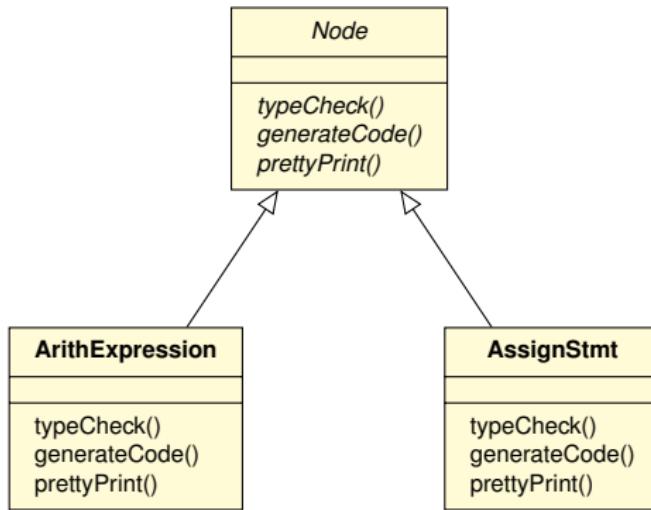
Intention

- it is a behavioral pattern
- models an operation (a set of operations) that runs over the elements of an object structure
- allows the definition of new operations without changing the classes of the elements over which the operations are executed

Motivation

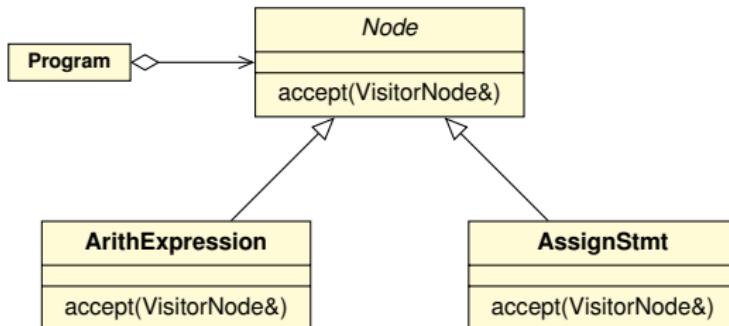
- A compiler is a program like an abstract syntactic tree (AST). This syntactic tree is used both for static semantics (e.g., type checking) and for code generation, code optimization, display.
- These operations differ from one type of instruction to another. For example, a node representing an assignment differs from a node representing an expression and consequently the operations on them will be different.
- These operations should be performed without changing the structure of the AST.
- Even if the structure of the AST differs from one language to another, the ways in which the operations are performed are similar.

Motivation: Polluting Solution

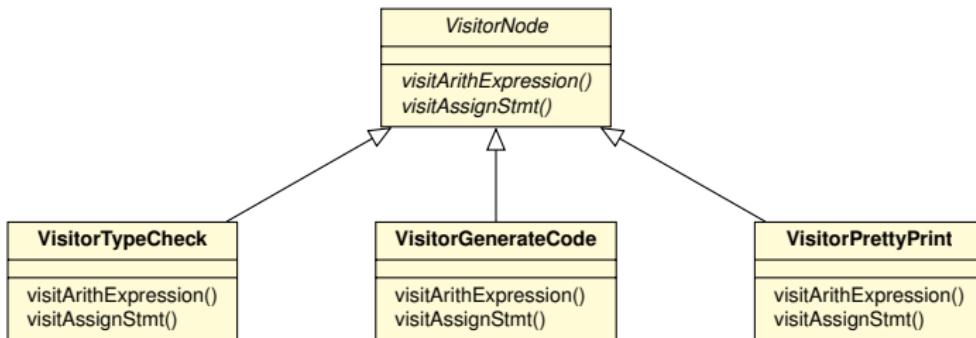


Motivation: Solution with Visitors

Hierarchy:

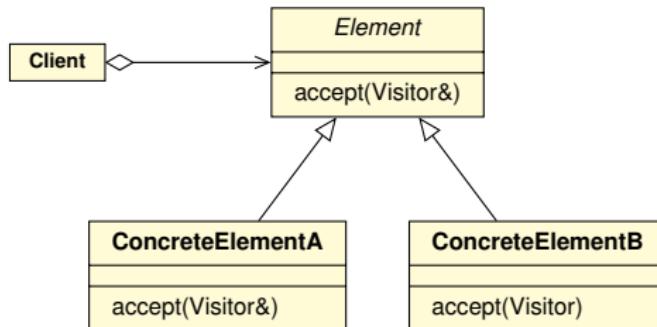


Visitors:

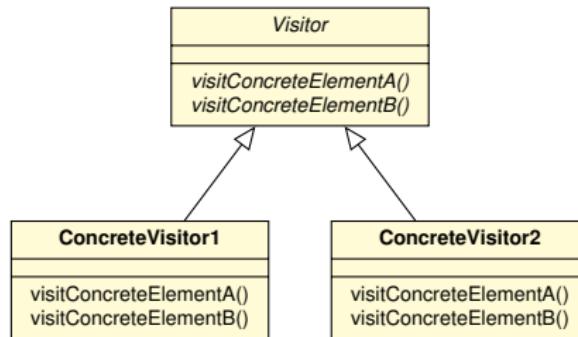


Structure

Hierarchy:



Visitors:



Participants 1/3

- Visitor (e.g., NodeVisitor)
 - declares a visit operation for each ConcreteElement class in the structure
 - the name of the operation and the signature identify the class that sends the visit request to the visitor
 - this allows the visitor to identify the specific element they are visiting
 - then, the visitor can visit the item through its interface

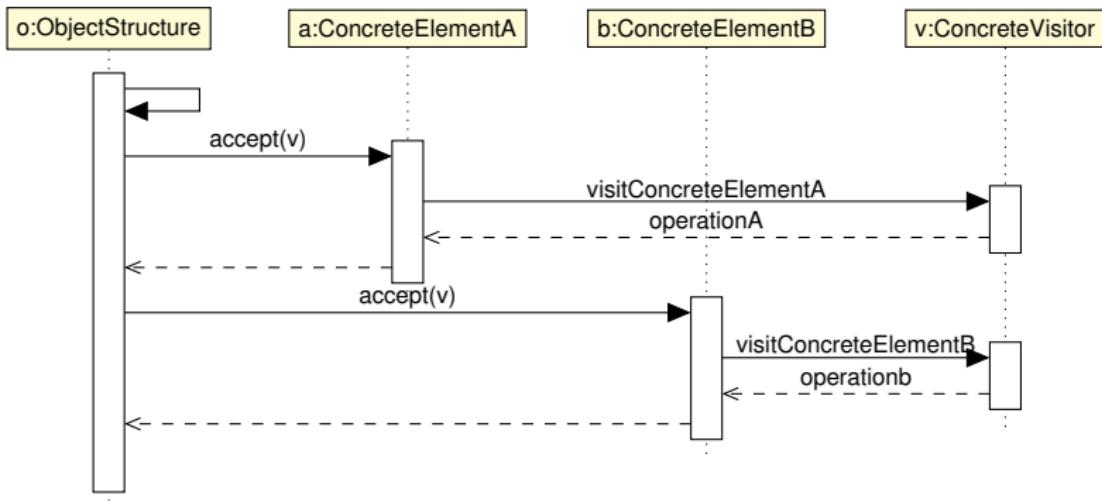
Participants 2/3

- ConcreteVisitor (e.g., TypeCheckingVisitor)
 - implements each operation declared by the visitor
 - each operation implements a fragment of the visit algorithm that corresponds to the element in the structure visited
 - it memorizes the state of the visiting algorithm, which often accumulates the results obtained while visiting the elements in the structure

Participants 3/3

- Element (Node)
 - defines accepting operations, which have a visitor as an argument
- ConcreteElement (e.g., AssignmentNode, VariableRefNode)
 - implements accepting operations
- ObjectStructure (e.g., Program)
 - it can list its elements
 - it can provide a high-level interface for a visitor visiting its elements
 - it can be a "composite"

Collaboration (Sequence Diagram)



Explanation

- after some internal computations, **o** sends the message `accept(v)` to **a** (in C++ this means that **o** calls `a.accept(v)`)
- then **a** sends the message `visitConcreteElementA` to **v** (i.e., **a** calls `v.visitConcreteElementA(this)`, a kind of " **v**, please visit me")
- then **v** "visits" **a** by executing `a.operationA()`
- then a similar scenario with **o** and **b** and **v**

Consequences 1/2

- Visitor makes adding new operations easy
- a visitor gathers the related operations and separates the unrelated ones
- adding new ConcreteElement classes to the structure is difficult
 - causes changes in the interfaces of all visitors
 - sometimes a default implementation in the Visitor abstract class can make the job easier

Consequences 2/2

- unlike iterators, a visitor can traverse multiple class hierarchies
- allows the calculation of cumulative states. Otherwise, the cumulative state must be transmitted as a parameter
- it could destroy the encapsulation
 - the concrete elements must have a strong interface capable of providing all the information requested by the visitor

Implementation 1/2

```
class Visitor {  
public:  
    virtual void visitElementA(ElementA*);  
    virtual void visitElementB(ElementB*);  
    // and so on for other concrete elements  
protected:  
    Visitor();  
};  
class Element {  
public:  
    virtual ~Element();  
    virtual void accept(Visitor&) = 0;  
protected:  
    Element();  
};
```

Implementation 2/2

```
class ElementA : public Element {  
public:  
    ElementA();  
    virtual void accept(Visitor& v) {  
        v.visitElementA(this);  
    }  
};  
class ElementB : public Element {  
public:  
    ElementB();  
    virtual void accept(Visitor& v) {  
        v.visitElementB(this);  
    }  
};
```

Simple/Double Dispatch

- **Simple dispatch.** The operation that makes a request depends on two criteria: the name of the request and the type of receiver. For example, `generateCode()` depends on the type of node.
- **Double dispatch.** The operation that performs the request depends on the types of two receivers. For example, an `accept()` call depends on both the component and the visitor.

Who Traverses Object Structure?

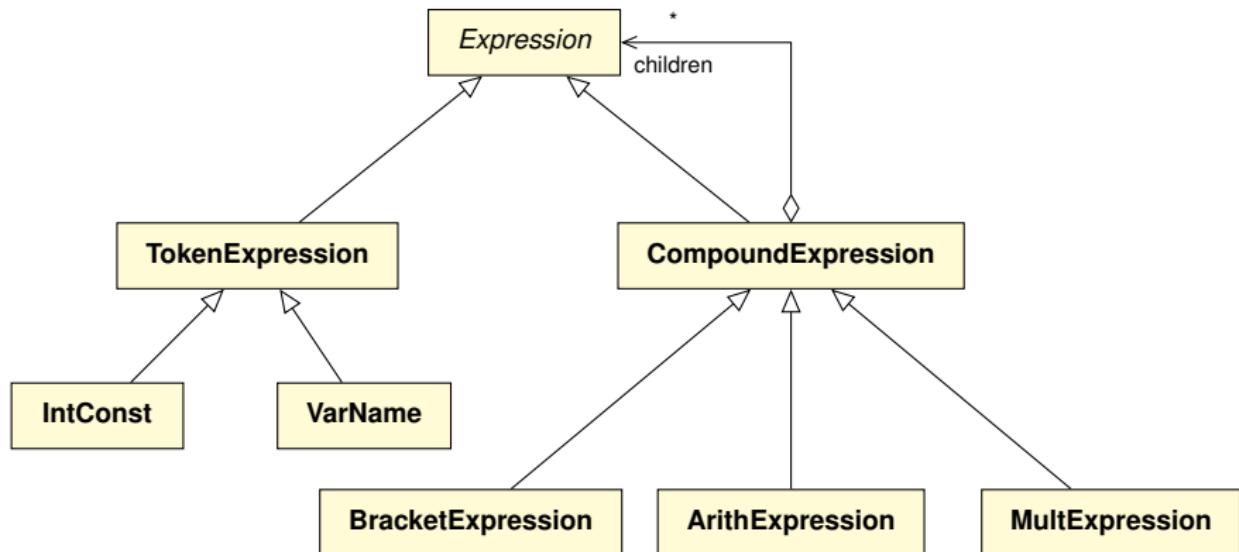
There are several options:

- the object structure itself
- the visitor
- an iterator

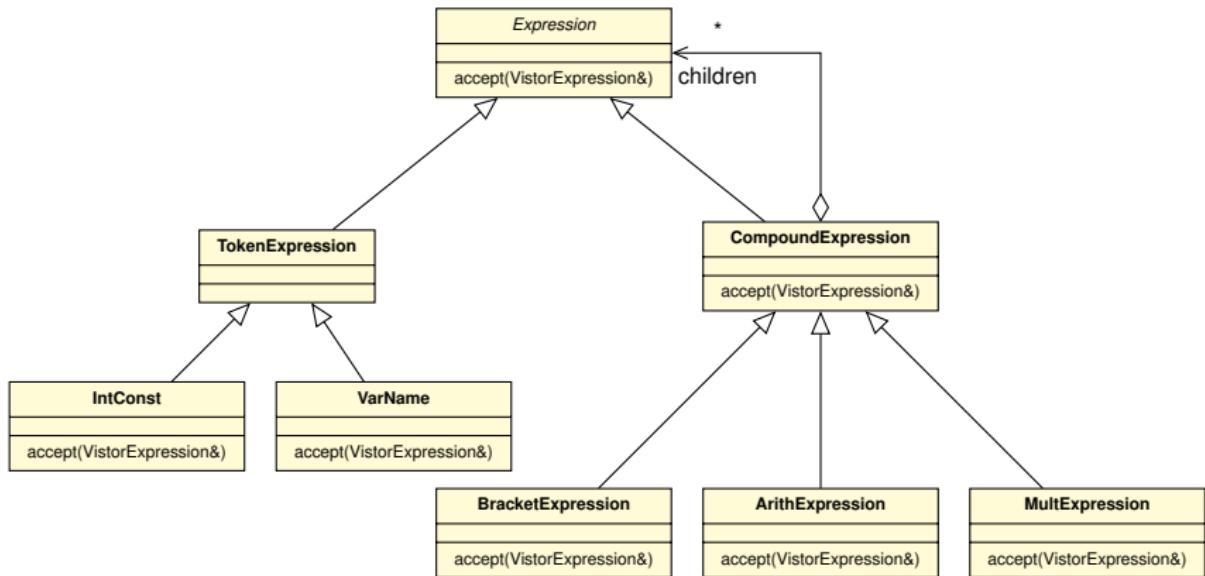
Plan

- 1 On Design Patterns
- 2 Singleton
- 3 Composite
Case Study: Expressions
- 4 Visitor
Combining Composite and Visitor
- 5 Object Factory

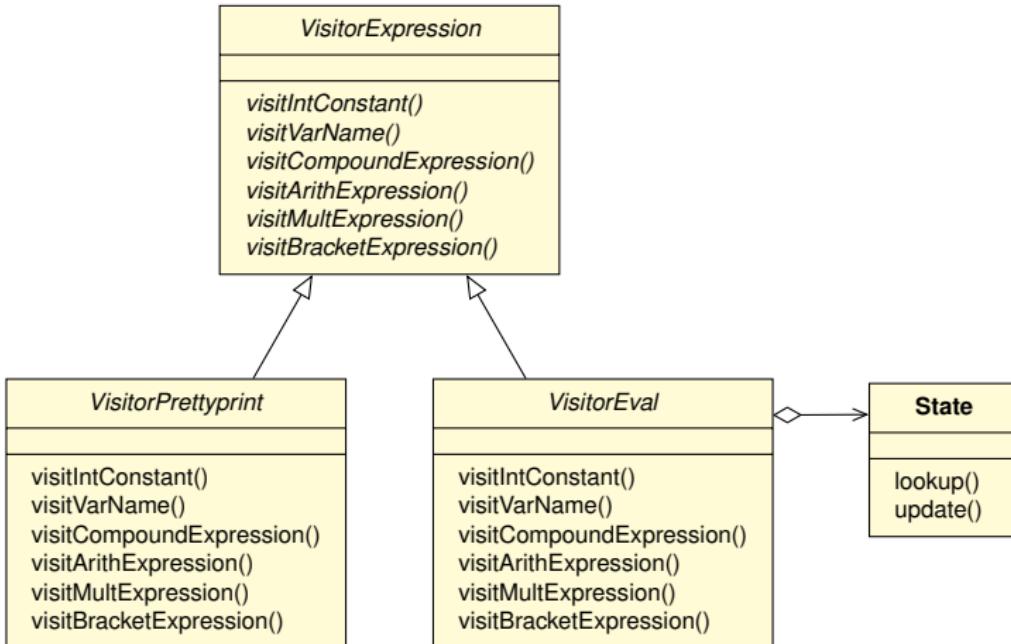
Recall Composite Diagram for Expression



Adding accept ()



Visitors for Expression



Implementation 1/2

The implementation can be found in the folder
examples/interpreter/cpp/expressions//visitor

main.cpp:

```
std:: cout << "Input: ";
std::cin.getline(str, 80);
Parser p(str);
std::optional<Expression*> ae = p.expression();
if (ae.has_value()) std::cout << ae.value()->toString() << "\n";
else std::cout << "nothing\n";
State st;
VarName a("a"), b("b");
st.update(a, 10);
st.update(b, 5);
st.print();
VisitorEval visitorEval1(st);
if (ae.has_value()) {
    ae.value()->accept(visitorEval1);
    cout << "ae = " << visitorEval1.getCumulateVal() << endl;
}
```

Implementation 2/2

Running:

```
$ g++ *.cpp ../../*.cpp -std=c++17 -o demo.exe
$ ./demo.exe
Input: a+b-2
(a + b - 2)
a |-> 10
b |-> 5
ae = 13
```

Plan

- 1 On Design Patterns
- 2 Singleton
- 3 Composite
Case Study: Expressions
- 4 Visitor
Combining Composite and Visitor
- 5 Object Factory

Intention

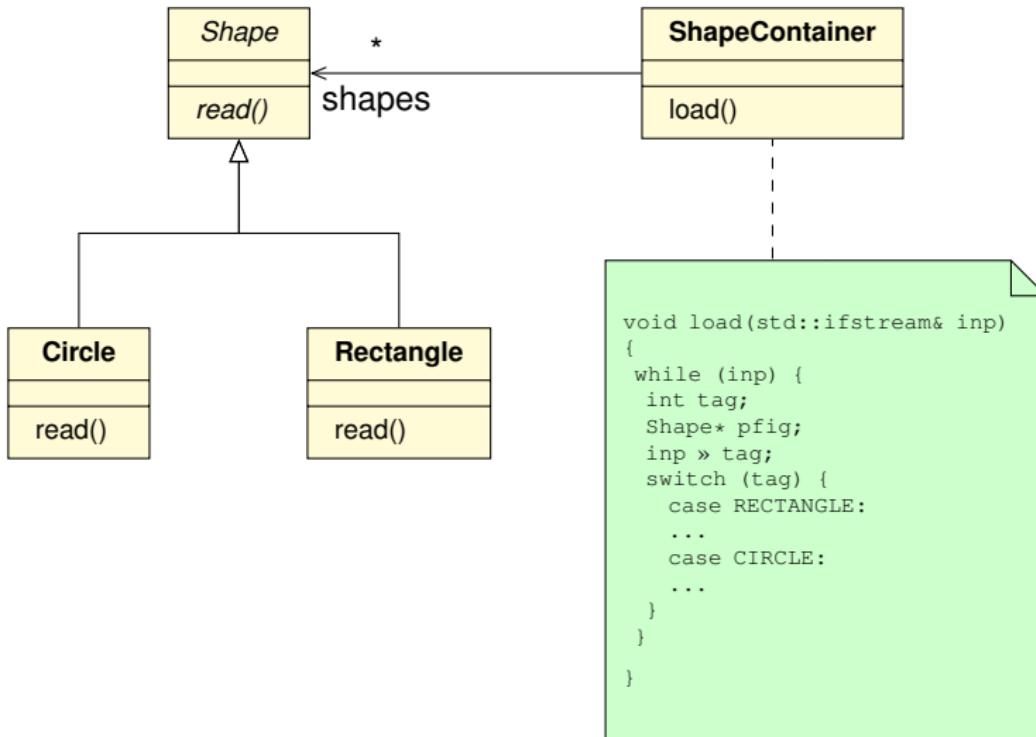
- it is a creational pattern
- to provide an interface for creating a family of intercorrelated or dependent objects without specifying their specific class

Applicability

- a system should be independent of how the products are created, composed or represented
- a system would be configured with multiple product families
- a family of intercorrelated objects is designed so that the objects can be used together
- you want to provide a product library and you want only the interface to be accessible, not the implementation

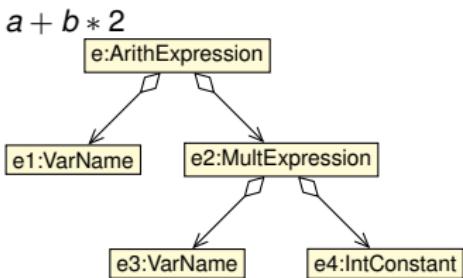
Motivation 1/2

Recall the counter-example from OCP:



If we add more shapes, we have to modify `ShapeContainer::load()`.

Motivation 2/2: (De)Serialization of Expressions



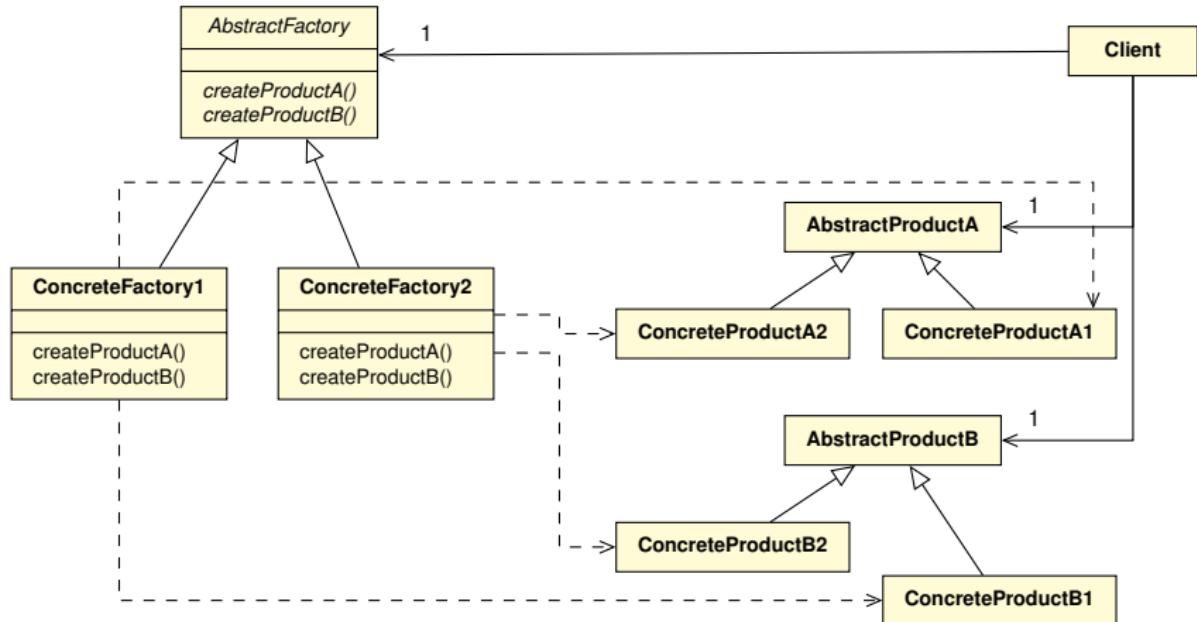
Deserialization

Serialization

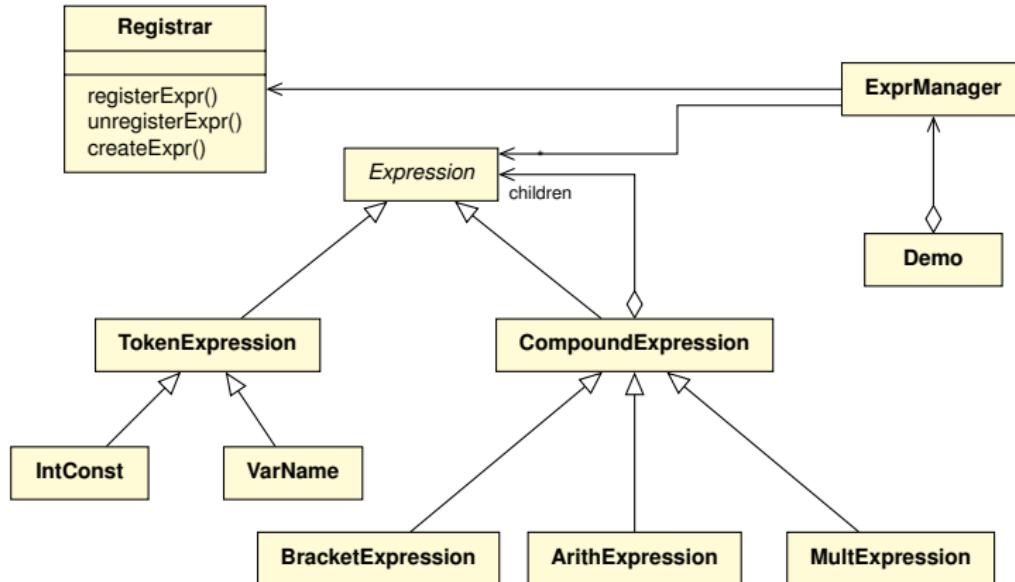
```
<arith>
  <label> [+] </label>
  <varName> a </varName>
  <mult>
    <label> [*] </label>
    <varName> b </varName>
    <intConstant> 2 </intConstant>
  </mult>
</arith>
```

XML notation

Structure



Expression Factory



Corresponds with the Standard Structure

- AbstractProductA = Expression
- AbstractProductB = Statements (not implemented yet)
- ConcreteFactory = Registrar (of expressions and statements)
- Client = ExprManager (in charge with (de)serialization)

Consequences

- isolates concrete classes
- simplifies the exchange of the product family
- promotes consistency among products
- supports new product families easily
- respects the open / closed principle

Participants: Registrar

- it is a class that manages the types of expressions
- registers a new type of expression (called whenever a new derived class is defined)
- delete a registered expression type (delete a derived class)
- creates expression objects
 - at the implementation level we use pairs (tag, createExprFn)
 - ... and callback functions (see next slide)
- Singleton template can be used to have a single factory (register)

Callback Functions

- a callback function is a function that is not explicitly invoked
- the responsibility for the call is delegated to another function that receives as parameter the address of the callback function
- the object factory uses callback functions to create objects: for each type there is a callback function that creates objects of that type
- for the "expression factory" we declare an alias for the type of Expression object creation functions:

```
typedef Expression* (*CreateExprFn)();
```

Implementation: Registrar 1/2

```
class Registrar
{
public:
    bool registerExpr(string tag,
                      CreateExprFn createExprFn )
    {
        return catalog.insert(
            std::pair<string, CreateExprFn>(
                tag, createExprFn)
            ).second;
    }
};
```

Implementation: Registrar 2/2

```
void unregisterExpr(string tag)
{
    catalog.erase(tag);
}

Expression* createExpr(string tag)
{
    map<string, CreateExprFn>::iterator i;
    i = catalog.find(tag);
    if ( i == catalog.end() )
        throw string("Unknown expression tag");
    return (i->second)();
}

protected:
    map<string, CreateExprFn> catalog;
};
```

Implementation: Object Creation

```
Expression* createIntConstant() {  
    return new IntConstant();  
}  
  
Expression* createVarName() {  
    return new VarName();  
}  
  
Expression* createMultExpression() {  
    return new MultExpression();  
}  
...
```

Implementation: ExprManager Constructor

```
ExprManager::ExprManager() {
    reg = new Registrar();
    reg->registerExpr("<intConstant>",
                       createIntConstant);
    reg->registerExpr("<varName>",
                       createVarName);
    reg->registerExpr("<mult>",
                       createMultExpression);
    reg->registerExpr("<arith>",
                       createArithExpression);
}
```

Full Implementation

The full implementation can be found in the folder
[examples/interpreter/cpp/expressions//factory](#)

Running:

```
$ g++ *.cpp -std=c++17 -o demo.exe
$ ./demo.exe
Input: a+b-2
```

test.xml file created:

```
<arith>
  <label> [+,-] </label>
  <mult>
    <label> [] </label>
    <varName> a </varName>
  </mult>
  <mult>
    <label> [] </label>
    <varName> b </varName>
  </mult>
  <mult>
    <label> [] </label>
    <intConstant> 2 </intConstant>
  </mult>
</arith>
```

Conclusion

- design patterns are a way to learn how to build your programs
- a design pattern is a tip that comes from people who have distilled their most common solutions into simple, digestible and suggestive advices
- OOP and design patterns are distinct topics
- OOP teaches you how to program, is it a programming methodology or a programming concept
- design patterns teach you
 - how to think about programs,
 - suggest methods for building classes / objects to solve a certain scenario in a program,
 - proven methods to succeed
- many other useful patterns are found in GoF