

Summary

- Constant expressions
- For each (Range-based for loop)
- Type inference
- Structured binding (destructuring)
- Static Polymorphism (CRTP)
- Plain Old Data (POD)

- 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:

Арр.срр	
void main()	
$\begin{cases} \\ int y = 10; \end{cases}$	
x = 10, int $y = x$:	
}	

- 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:



- 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:



However, adding a "const" declaration in front of "x" makes the compiler change the way if creates "y" (now the compiler will directly assign the value 10 (the constant value of "x" to "y")

- 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:



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 are in particular important when declaring arrays:

Арр.срр	
<pre>int GetCount() {</pre>	
return 5;	
void main()	
{ int x[GetCount()]: error C2131: expression did not evaluate to a constant note: failung was caused by call of undefined function on	one not declared 'consteven'
note: see usage of 'GetCount	one not declared constexpr

- ► 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 are in particular important when declaring arrays:

Арр.срр	
<pre>const int GetCount()</pre>	
return 5;	
J void main()	
{ error C2131: expression did not evaluate to a co	onstant
<pre>int x[GetCount()]; note: failure was caused by call of undefined fu } note: see usage of 'GetCount</pre>	unction or one not declared 'constexpr'

- 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).

Cx++11 adds a new keyword: "constexpr" that tells the compiler that a specific expression should be considered constant.

Арр.срр
constexpr int GetCount()
{
return 5;
}
void main()
{
<pre>int x[GetCount()];</pre>
}

Now the code will compile.

Cx++11 adds a new keyword: "constexpr" that tells the compiler that a specific expression should be considered constant.

Арр.срр		
<pre>constexpr int GetCount() {</pre>		
<pre>return 5; } void main()</pre>	push ebp mov ebp,esp	
<pre>{ int x[GetCount()]; }</pre>	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)

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

- Using "constexpr" comes with some limitations:
 - □ A **constexpr** function should not have any local variables uninitialized

Арр.срр		
<pre>constexpr int GetCount() { int x; x = 10; noturn 5;</pre>	error C3615: constexpr function 'GetCount' cannot result in note: failure was caused by an uninitialized variable decla note: see usage of 'x'	a constant expression ration
<pre>return 5; } void main() { int x[GetCount()]; }</pre>		

► 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.

- Using "constexpr" comes with some limitations:
 - □ A **constexpr** function should not have any local variables uninitialized. You can have, however multiple constant variable defined !



- 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.

Арр.срр
constexpr int GetCount(int x)
{
return x+x;
}
void main()
<pre>{ int x[GetCount(10)];</pre>
}

- 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.

- 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.

Арр.срр	
constexpr int Get	ount(int x)
{ if (x>10) r	turn 5; else return 6;
<pre>} void main()</pre>	
{ int x[GetCo	nt(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.

int x =	SomeValue();
call	SomeValue
mov	dword ptr [x],eax
printf("	'%d", x);
mov	<pre>eax,dword ptr [x]</pre>
push	eax
push	offset string "%d"
call	printf
add	esp,8

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"



- 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 <u>completely identical</u> as declaring "x" as a const ! (if we would have used a const specifier SomeValue function would still be called !)

Let's analyze the following code:

Normal	variable	With constexpr		With const	
<pre>constexpr i return } int main() { int x = printf(}</pre>	<pre>nt SomeValue() { 5; SomeValue(); "%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>	
call mov	SomeValue dword ptr [x],eax	mov	dword ptr [x],5	call mov	SomeValue dword ptr [x],eax
printf("%d", x);	printf('	'%d", x);	printf(("%d", x);
mov push push call	eax,dword ptr [x] eax offset string "%d" printf	push push call add	5 offset string "%d" printf esp,8	push push call add	5 offset string "%d" printf esp,8
call add	printf esp,8	add	esp,8	add	esp,8

Code was compiled with VS 2017, with C++17 Standards enabled (debug mode)



- 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 !

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.

- As a general consent, consider constexptr 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.



- generate the following assembly code.
- Code was compiled with VS 2017, with C++17 Standards enabled.

Let's analyze the following code:



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*

Let's analyze the following code:

App.cpp class A public: int x; dword ptr [a.x], 25 constexpr A(int value): x(value*value) {} mov }; int main() eax, dword ptr [a.x] mov push eax constexpr A a(5); offset string "%d" printf("%d", a.x); push 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.

Арр.срр	
<pre>class A { public: int x; constexpr A(int value): x(value*value) { printf("ctor"); } error C361 'A::A' canno { constexpr A a(5); printf("%d", a.x); 'A::A' canno constant ex con</pre>	5: constexpr function ot result in a xpression
}	

- 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.



- 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.



- 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

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.

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);
    }
}
```

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.

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.



- 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.

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));
}</pre>
```

Code will still compiles and run correctly (even if we made *cmmdc* function more complex). We have also used a temporary variable (*tr*) but with **constant** values.

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));
}</pre>
```

Code will still compiles and run correctly (even if we made *cmmdc* function more complex). We have also used a temporary variable (*tr*) but with **constant** values.

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));
}</pre>
```

Code will still compiles and run correctly. This time we have changed a constant value with another constant value (y) !!!

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));
}</pre>
```

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 produce 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)



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)



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.



Code was compiled with VS 2017, with C++17 Standards enabled.

- 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_expresion) 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).

Examples:

App.cpp

void	main(()						
L	int	v[3]	_	ſ	1	2	з	٦·
	fon	^[J]	- -	ι.	, ד עו	ر 2	5	ور
	TOP	(IIIC	т т.	· - c	X) (110/.		<u>.</u> \	
		pri	Lητ	.т(~~(, 1	1));
ł								

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

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::initialized_list object and iterates in it.

IOV	dword ptr [ebp-38h],1
IOV	dword ptr [ebp-34h],2
IOV	dword ptr [ebp-30h],3
IOV	dword ptr [ebp-2Ch],4
IOV	dword ptr [ebp-28h],5
.ea	eax,[ebp-24h]
oush	eax
.ea	ecx,[ebp-38h]
oush	ecx
.ea	ecx,[ebp-1Ch]
all	constructor for initializer_list

(int>

In case of a normal array (where size is know) the compiler simulates a for loop:



for (int i : x)

```
printf("%d", i);
```

► This code will not compile → the compiler can not know, in advanced how many elements are stored in "x" array.

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]"

Арр.срр
void main()
{
int x[][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
for (int i : x)
<pre>printf("%d", i);</pre>
}

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++)</pre>
                   printf("%d", i[index]);
```

References can also be used. In this case, the content of that loop can be modified accordingly.

A	pp.cpp
vo:	id main()
l	int x[] = { 1, 2, 3 };
	for (int &i : x) i *= 2;
}	<pre>for (int i : x) printf("%d,", i);</pre>

The output will be 2,4,6 (as the elements from x have been modified in the first for loop.

References can also be used. In this case, the content of that loop can be modified accordingly.



This code will not work because x is a const vector. The compiler can't assign a "const int &" to a "int &"

References can also be used. In this case, the content of that loop can be modified accordingly.



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 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);
}</pre>
```

This code will not compile as no "begin()" and "end()" functions are available for class MyVector.

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);
}</pre>
```

Be careful when using references. MyVector::x is a private field. However, it can be access 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;
}</pre>
```

The code works and v::x will be modified.

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;
}</pre>
```

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")

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

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.

Type inference

- C++11 introduces a new keyword : "auto" that can be use 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 discuss later.

► Examples:

C++11	Translation
<pre>void main()</pre>	void main()
<pre>i auto x = 10; auto y = 10.0f; auto z = 10.0; auto b = true; auto c = "test"; auto l = 100L; auto ll = 100L; auto ul = 100UL; auto ul = 100UL; auto ull = 100UL; auto ch = 'x'; auto wch = L'x';</pre>	<pre>{ 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 = 100UL; unsigned long ul = 100UL; unsigned long long ull = 100ULL; char ch = 'x'; wchar_t wch = L'x'; </pre>
auto d = NULL;	<pre>int d = NULL;</pre>
auto p = nullptr;	<pre>void* p = nullptr;</pre>
}	}

Examples:



"auto" can be forced if a casting occurs during initialization.



- However, the code will still crashes as "x" point to a const char* value.
- Using "new" operator also forces a cast.

C++11	Translation
<pre>void main() { auto x = new char[10] x[0] = 0; }</pre>	<pre>void main() { char* x= new char[10]; x[0] = 0; }</pre>

In this case the code works properly (x will be a char*)

"auto" can be used with user defined classes as well:

C++11	Translation
class Test	class Test
{	{
public:	public:
int x, y;	<pre>int x, y;</pre>
};	};
<pre>void main()</pre>	void main()
{	{
auto x = new Test();	Test* x = new Test();
}	}

"auto" can be used with "const" keyword

C++11	Translation
<pre>void main() {</pre>	<pre>void main() {</pre>
<pre>const auto x = 5; }</pre>	<pre>const int x = 5; }</pre>

"auto" can be used with another variable / expression.

C++11	Translation
<pre>void main()</pre>	void main()
{	{
auto $x = 5;$	int $x = 5;$
auto y = x;	int $y = x;$
auto &z = x;	int $\&z = x;$
auto *ptr = &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
<pre>void main()</pre>	<pre>void main()</pre>
{	{
auto $x = 5;$	int $x = 5;$
auto y = x * 1.5;	double $y = x * 1.5$;
auto $z = x > 100;$	bool $z = x > 100;$
}	}

"auto" can also be used to create pointer to a function:

C++11	Translation
<pre>int sum(int x, int y, int z)</pre>	int sum(int x, int y, int z)
{	{
return $x + y + z;$	return x + y + z;
}	}
<pre>void main()</pre>	void main()
{	{
auto f = sum;	<pre>int (*f)(int,int,int) = sum;</pre>
auto result = f(1, 2, 3);	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" is also useful when dealing with templates:

C++11	Translation
<pre>using namespace std; #include <vector></vector></pre>	using namespace std; #include <vector></vector>
<pre>void main() {</pre>	<pre>void main() {</pre>
<pre>vector<int> v;</int></pre>	vector <int> v;</int>
<pre>auto it = v.begin(); }</pre>	<pre>vector<int>::iterator it = v.begin(); }</int></pre>

In this case, it is much easier to declare something as "auto" than to write the entire declaration as a template.

"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;
```

In this example we two variables defined ("it" and "range").

"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>.



"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);
```

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 can be used with constants as well:

Cpp code

void	main()
	decl + vne(10) v
	docltype(10) X,
	decitype(10.21) y,
	decltype(hulipth) 2;
	decitype(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 can be used with arrays:

Срр	code			
void m {	ain()			
t	int v[10]; int w[10][20];			
	<pre>decltype(v) x; decltype(w) y;</pre>			
}				

In this example:

- □ "x" will be of type int[10] \rightarrow just like "v" is
- □ "y" will be of type int[10][20] \rightarrow just like "w" is

decltype can be used with elements from an array - but the result will be a reference of that type.

Cpp code	
<pre>void main() { int v[10]; decltype(v[0]) x; } error C2530: 'x': references m }</pre>	nust be initialized
This code will NOT compile because "x" is of type "int 8 initialized. For this a reference must be added to the ir	t" and it is not tit is not a state the term term term term term term term ter
Cpp code	
<pre>void main() { int v[10]; decltype(v[0]) x = v[0]; }</pre>	
Now the code compiles and "x" is a reference to the firm	st element from "v"

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	
<pre>void main() {</pre>	
<pre>decltype("Te") x = *(&"CC"); }</pre>	

▶ In this case, "x" will be of type "const char[3] &"

Structured binding (destructuring)

- 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 assign to a variable.
- The concept is related to what other languages (like Python) have \rightarrow 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 $[v_1, v_2, ..., v_n]$ = expression auto& $[v_1, v_2, ..., v_n]$ = expression

where $v_1, v_2 \dots v_n$ are variables that are going to be binded.

```
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
	<pre>auto [x, y] = a;</pre>	
lea	eax,[a]	
mov	dword ptr	<pre>[temp_ptr_to_a],eax</pre>
mov	eax,4	
imul	ecx,eax,0	
mov	edx,dword	<pre>ptr [temp_ptr_to_a]</pre>
mov	eax,dword	ptr [edx+ecx]
mov	dword ptr	[x],eax
mov	eax,4	
shl	eax,0	
mov	ecx,dword	<pre>ptr [temp_ptr_to_a]</pre>
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

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

Let's analyze the following code:



- 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

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 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

Ann cnn

Structured bindings takes into account the access specifier.

Арр.срр	
class A	
<pre>{ int x, y, z; public: A(int value) : x(value), y(value * 2), z(value * 4) {}; }; int main() { A a(1); puto[x, y, z] = 2; }</pre>	<pre>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</pre>
<pre>auto[x, y, z] = a, printf("x=%d, y=%d, z=%d", x, y, z); return 0; }</pre>	<pre>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'</pre>
and the binding is NOT possible.	note: see declaration of 'A::z' note: see declaration of 'A'

► This code will not compile !

You ca not bind only some data members - you have to bind all of them.



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 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 bindings are often used with for-each loops and STL, especially for maps where access to both components (key and value) can be obtains 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

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).

Арр.срр		
<pre>struct Student { const char * Name; int Grade;</pre>		
<pre>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); }</pre>	This method is however effective as it implied your own local variad translation function class (something that a std::tuple	ver not that es creating bles and a within the can return e)
return 0; }		

This code will compile and will print upon execution the following: Name:Popescu, Grade:10

Static Polymorphism (CRTP)

Static polymorphism (also called <u>C</u>uriously <u>Recurring Template Pattern or CRTP</u>) is a technique that takes advantage that a template is not instantiated (constructed) when it is written - but when it's 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).

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**

Let's see an example:

t.PrintName(); d.PrintName(); return 1;

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;
}
```

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 ***

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 excepted.

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

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=""></typename>
class Base
{
T& SomeMethod() {
return static_cast<t&>(*this);</t&>
}
};
<pre>class Derived: public Base<derived> {};</derived></pre>

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().Print();
   Float f; f.Value = 1.5;
   f.Inc().Print().Dec().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; }</pre>
};
struct Integer : public Comparable<Integer> {
    int Value;
    Integer(int v): Value(v) {}
    int CompareWith(const Integer& obj) const {
        if (Value < obj.Value) return -1;</pre>
        if (Value > obj.Value) return 1;
        return 0;
};
void main() {
                                                                 In this case, Integer class has both
    Integer i1(10);
                                                              operator== and operator< defined and as</pre>
    Integer i2(20);
                                                             such a syntax like (if (i1<i2)) will compile.
    if (i1 < i2) printf("i1 is smaller than i2");</pre>
```

- **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 vpftr 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)

Differences between static polymorphism and dynamic polymorphism.

Static polymorphism

struct Base { }; template <typename T> struct Base { }; class A : public Base<A> { }; 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 *'

Dynamic polymorphism

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;



Code will compile and run as expected.

Plain Old Data(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.

- Trivial types means that:
 - $_{\circ}$ Has a default constructor (that is not provided by the programmer)
 - $_{\odot}$ $\,$ 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)
 - \circ 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.

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;
}</pre>
```

```
This code will print "true" for TypeA and "false" for TypeB (because it has a user defined constructor)
```



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;
}</pre>
```

► This code will print "true" for TypeC

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;
}</pre>
```

► This code will print "false" for TypeD (because it is using a initialization function → it will be discuss in the Initialization list chapter)

- Standard layout types means that:
 - $_{\odot}$ $\,$ 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 {};
class Derived : Base {
    int x;
};
void main()
{
    printf("SizeOf(Base) = %d\n", sizeof(Base));
    printf("SizeOf(Derived) = %d\n", sizeof(Derived));
}
```

- 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:

Арр.срр
class Base {};
<pre>class Derived : Base { Base b; int x;</pre>
};
<pre>void main() {</pre>
<pre>printf("SizeOf(Base) = %d\n", sizeof(Base)); printf("SizeOf(Derived) = %d\n", sizeof(Derived)); }</pre>

- 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.

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 if has both public and private members.

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.

