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)

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;
}
```

```
int x = 10;
mov     dword ptr [x],0Ah

int y = x;
mov     eax,dword ptr [x]
mov     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;
mov     dword ptr [x], 0Ah
    int y = x;
mov     dword ptr [y], 0Ah
```

- 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

- 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

- Cx++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

- Cx++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;
      x = 10;
      return 5;
}
void main()
{
      int x[GetCount()];
}
```

```
error C3615: constexpr function 'GetCount' cannot result in a constant expression
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 **constexpre** 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

```
    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
```

- 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"

Constant expressions

```
    constexpr int x = SomeValue();
mov         dword ptr [x],5
    printf("%d", x);
push        5
push        offset string "%d"
call        printf
add         esp,8
```

- Let's analyze the following code:

App.cpp

```
constexpr int SomeValue() { return 5; }
int main()
{
    constexpr int x = SomeValue();
    printf("%d", 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 !)

Constant expressions

- Let's analyze the following code:

Normal variable	With constexpr	With const

`constexpr int SomeValue() {`
`  return 5;`
`}`
`int main()`
`{`
`  int x = SomeValue();`
`  printf("%d", x);`
`}`
	`constexpr int SomeValue() {`
	`  return 5;`
	`}`
	`int main()`
	`{`
	`  constexpr int x = SomeValue();`
	`  printf("%d", x);`
	`}`
		`constexpr int SomeValue() {`
		`  return 5;`
		`}`
		`int main()`
		`{`
		`  const int x = SomeValue();`
		`  printf("%d", x);`
		`}`

`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`
	`mov  dword ptr [x],5`
	`  printf("%d", x);`
	`push  5`
	`push  offset string "%d"`
	`call  printf`
	`add  esp,8`
		`call  SomeValue`
		`mov  dword ptr [x],eax`
		`  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 (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;
    printf("%d", x);
}
```

```
error C3892: 'x': you cannot assign to a variable that is const
```

- 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)

`class A`
`{`
`public:`
`  constexpr int x;`
`  A() : x(10) {}`
`};`
	`class A`
	`{`
	`public:`
	`  const int x;`
	`  A() : x(10) {}`
	`};`
		`class A`
		`{`
		`public:`
		`  static constexpr int x = 10;`
		`  A() {}`
		`}`
		`int main()`
		`{`
		`  printf("A::x = %d, sizeof(A) = %d", A::x,sizeof(A));`
		`  return 0;`
		`}`

- 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 **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.

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 alv
  expression. Let's conside

App.cpp

```
constexpr int cmmdc(int x, int
{
      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)

- 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 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 compiles 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 compiles 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 compiles 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 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)

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     dword ptr [x], 55
push    55
push    offset string "%d"
call    printf
add     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);
    printf("%d", x);
}
```

error C2131: expression did not evaluate to a constant

- 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;
}
```

```
mov        dword ptr [x],200000h
```

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

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_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).

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::initialized_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 know) 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 advanced 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 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;
}
```

- 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.

- Type inference

"auto" keyword

- 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.

"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 assign 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 v_1,v_2 …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 ca 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 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

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`

Static Polymorphism

- (CRTP)

Static Polymorphism

- Static polymorphism (also called **C**uriously **R**ecurring **T**emplate **P**attern or CRTP) 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).

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 excepted.

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 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<B> 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;
}
```

```
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

```
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;
}
```

Code will compile and run as expected.

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 {};

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:

App.cpp

```
class Base {};

class Derived : Base {
      Base b;
      int x;
};

void main()
{
      printf("SizeOf(Base) = %d\n", sizeof(Base));
      printf("SizeOf(Derived) = %d\n", sizeof(Derived));
}
```

- 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 if 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