OOP

Lecture-7

Gavrilut Dragos

New things in C++

- Constant expressions
- NULL pointer
- Plain Old Data (POD)
- Initialization lists
- For each (range-based for loop) support
- "auto" keyword (type inference)

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()];
}
```

- 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()];
}
```

- 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. However, this keyword is not implemented in VS 2013 (only in g++) .

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 **constexpre** 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 **constexpre** function should not have any local variables defined

App.cpp

```
constexpr int GetCount()
{
      int x;
      return 5;
}
void main()
{
      int x[GetCount()];
}
```

- The compiler assumes that if any variable is defined the purpose of that function might not be to return a constant value. In this case, even if “x” from GetCount() is not used, the code will not compile.

Constant expressions

- Using “**constexpr**” comes with some limitations:
  - If **constexpre** 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 **constexpre** function in C++11 must have only one return statement. This is subject to a change.

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 Cx++14 standards (g++). The compiler evaluates that GetCount(10) can actually be replaced with 6 without changing the logic behind the construction.

Constant expressions

- However, some things can not be evaluated:

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);
	printf("x = %d\n",x);
}
```

- While “x” is clearly 6, the compiler can not pre-compute the value of the function and replace the value of “x” with 6. This code compiled with g++, with both “-std=c++17” and “-std=c++14” will compile but it will also create code for cmmdc function. If compiled with “-std=c++11” it will produce an error.

NULL pointer

NULL pointer

- Let’s consider the following code:

App.cpp

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

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

	Print(NULL);
}
```

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

NULL pointer

- Let’s consider the following code:

App.cpp

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

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

	Print(NULL);
}
```

Output

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

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

NULL pointer

- Let’s consider the following code:

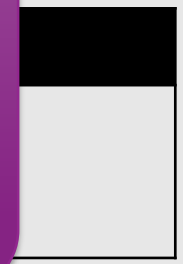
App.cpp

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

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

	Print(NULL);
}
```

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



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

NULL pointer

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

App.cpp

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

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

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

NULL pointer

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

App.cpp

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

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

App.cpp

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

NULL pointer

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

App.cpp

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

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

App.cpp

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

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

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

NULL pointer

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

App.cpp

```
void Print(bool value) { … }

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

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

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

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 **memcopy** 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()
{
	cout << std::boolalpha << std::is_trivial<TypeA>::value << endl;
	cout << std::boolalpha << std::is_trivial<TypeB>::value << 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 time 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 MyTest has a private member “Base::xx”

Initialization lists

Initialization lists

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

App.cpp

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

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

Assembly code generated

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

Initialization lists

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

App.cpp

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

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

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

Initialization lists

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

App.cpp

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

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

App.cpp

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

Initialization lists

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

App.cpp

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

```
push      0Ch
call      operator new
add       esp,4

mov       dword ptr [ebp-0D4h],eax
cmp       dword ptr [ebp-0D4h],0
je        _error_allocate_memory_

mov       eax,dword ptr [ebp-0D4h]
mov       dword ptr [eax],1
mov       ecx,dword ptr [ebp-0D4h]
mov       dword ptr [ecx+4],2
mov       edx,dword ptr [ebp-0D4h]
mov       dword ptr [edx+8],3
```

Initialization lists

- The same can be applied for classes and/or structures:

App.cpp

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

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

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

Initialization lists

- The same can be applied for structures and/or classes:

App.cpp

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

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

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

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

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

Initialization lists

- The same can be applied for structures and/or classes:

App.cpp

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

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

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

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

Initialization lists

- The same can be applied for structures and/or classes:

App.cpp

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

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

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

- This code will not work as “x” is not public ! If a class has at least one member that is not public, these assignments will not be possible.

Initialization lists

- The same can be applied for structures and/or classes:

App.cpp

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

	Data(int xx, char tt, const char * mm) : x(xx), t(tt), m(mm) {};
};
void main()
{
	Data d1{ 10, 'A', "test" };

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

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

Initialization lists

- The same can be applied for structures and/or classes:

App.cpp

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

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

Initialization lists

- The same can be applied for structures and/or classes:

App.cpp

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

```
push    0Ch
lea     ecx,[d1]
call    Data::__autoclassinit2
```

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

Initialization lists

- The same can be applied for structures and/or classes:

App.cpp

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

```
push    0Ch
lea     ecx,[d1]
call    Data::__autoclassinit2
```

The size of Data class (12 bytes)

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

Initialization lists

- The same can be applied for structures and/or classes:

App.cpp

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

```
push	0Ch
lea	ecx,[d1]
call	Data::__autoclassinit2
```

```
push	ebp
mov	ebp,esp

mov	dword ptr [this],ecx
mov	eax,dword ptr [ebp+8]
push	eax
push	0
mov	ecx,dword ptr [this]
push	ecx
call	_memset
add	esp,0Ch

mov	esp,ebp
pop	ebp
ret	4
```

- This code will compile. “d1” object will have t
execution: d1.x = 0, d1.t = ‘\0’ and d1.m = NU

Initialization lists

- The same can be applied for structures and/or classes:

App.cpp

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

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

- The compiler creates a initialization function that will fill the entire content of Data with 0 values. The result will be that x will be 0, t will be the character with ASCII code 0 and m will be NULL (also defined as 0).

Initialization lists

- The same can be applied for structures and/or classes:

App.cpp

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

      Data(int xx, char tt, const char * mm) :
                        x(xx), t(tt), m(mm) {};
};
void main()
{
      Data d1{ 10, 'A', "test" };

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

```
push      0Ch
lea       ecx,[d1]
call      Data::__autoclassinit2
push      0E58A8h
push      41h
push      0Ah
lea       ecx,[d1]
call      Data::Data

push      0Ch
lea       ecx,[d2]
call      Data::__autoclassinit2
push      0E58B0h
push      42h
push      5
lea       ecx,[d2]
call      Data::Data
```

Initialization lists

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

App.cpp

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

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

- The previous code will NOT compile on VS 2013 (as that version does not implement the full specification of Cx++11). However, it can be tested on g++ (g++ (Ubuntu 5.4.0-6ubuntu1~16.04.2) 5.4.0 20160609) or later by compiling with the following command: **g++ <file.cpp> -std=c++11**

Initialization lists

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

App.cpp

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

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

- The same applies in the this case as well (VS 2013 will not support this kind of initialization, g++ will, provided that you use **-std=c++11** as a compiling parameter.

Initialization lists

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

App.cpp

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

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

- This code will be compiled on VS 2013. The result will be a new default constructor defined that will initialized “x”, “y” and “t” with the specified values.

Initialization lists

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

App.cpp

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

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

```
lea	ecx,[d]
call	Data::Data
```

- This code will be compiled on VS 2013. The result will be a new default constructor defined that will initialized “x”, “y” and “t” with the specified values.

Initialization lists

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

App.cpp

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

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

```
lea	ecx,[d]
call	Data::Data
```

```
push	ebp
mov	ebp,esp

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

mov	esp,ebp
pop	ebp
ret
```

- This code will be compiled on VS 2013. T
constructor defined that will initialized
values.

Initialization lists

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

App.cpp

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

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

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

Initialization lists

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

App.cpp

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

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

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

nop
nop

mov     eax,dword ptr [this]
```

- Adding a constructor will force … to integrate the default initializa…

Initialization lists

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

App.cpp

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

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

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

Initialization lists

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

App.cpp

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

	int* p = NULL;
};

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

- However, using a pointer in the initialization list makes the VS 2013 compiler to create another function (a memset function) that will initialized all values to 0 before calling the constructor,

Initialization lists

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

App.cpp

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

	int* p = NULL;
};

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

```
push     10h
lea      ecx,[d]
call     Data::__autoclassinit2

lea      ecx,[d]
call     Data::Data
```

- However, using a pointer in the initialization list makes the VS 2013 compiler to create another function (a **memset** function) that will initialized all values to 0 before calling the constructor,

Initialization lists

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

App.cpp

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

	int* p = NULL;
};

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

```
push	10h
lea	ecx,[d]
call	Data::__autoclassinit2

lea	ecx,[d]
call	Data::Data
```

A **memset** function that fill the content of an object of type Data with value 0.

- However, using a pointer in the initialization list makes the VS 2013 compiler to create another function (a **memset** function) that will initialized all values to 0 before calling the constructor,

Initialization lists

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

App.cpp

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

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

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

Initialization lists

- Initialization lists work with STL as well:

App.cpp

```
using namespace std;
#include <vector>
#include <map>

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

void main()
{
	vector<int> v = { 1, 2, 3 };
	vector<string> s = { "POO", "C++" };
	vector<Student> st = { { "Popescu", 10 }, { "Ionescu", 9 } };
	map<const char*, int> st2 = { { "Popescu", 10 }, { "Ionescu", 9 } };
}
```

- The code complies ok and all objects are initialized properly.

Initialization lists

- Initialization lists work with STL as well:

App.cpp

```
using namespace std;
#include <vector>
#include <map>

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

void main()
{
	vector<int> v = { 1, 2, 3 };
	vector<string> s = { "POO", "C++" };
	vector<Student> st = { { "Popescu", 10 }, { "Ionescu", 9 } };
	map<const char*, int> st2 = { { "Popescu", 10 }, { "Ionescu", 9 } };

	map<const char*, int> st3 = { { "Popescu", 10 }, { "Popescu", 9 } };
}
```

In this case, “st3” will contain only one item (Popescu with the grade 10) the first one that was added. This is because the <map> template allows only one item for each key.

Initialization lists

- STL also provide a special container (called **std::initializer_list** that can be used to pass a initialization list to a function).

App.cpp

```
using namespace std;
#include <initializer_list>

int Sum(std::initializer_list<int> a)
{
	int result = 0;
	std::initializer_list<int>::iterator it;
	for (it = a.begin(); it != a.end(); it++)
		result += (*it);
	return result;
}

void main()
{
	printf("Sum = %d\n", Sum( { 1, 2, 3, 4, 5 } ));
}
```

- This code will compile and will print to the screen the value 15.

Initialization lists

- If **std::initializer_list** is used in a constructor the following expression for initializing an object can be used:

App.cpp

```
using namespace std;
#include <initializer_list>

class Data
{
	int v[10];
public:
	Data(std::initializer_list<int> a)
	{
		int index = 0;
		std::initializer_list<int>::iterator it;
		for (it = a.begin(); (it != a.end()) && (index<10); it++,index++)
			v[index] = (*it);
	}
};
void main()
{
	Data d1({ 1, 2, 3, 4, 5 });
	Data d2 = { 1, 2, 3, 4, 5 };
}
```

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 in advanced how many elements "x" has

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.

"auto" keyword (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);


}
```

- Easier, right ?

"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);
}
```

"auto" 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”.

"auto" 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)

"auto" 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

"auto" keyword

- decltype can be 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;
}
```

- 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”

"auto" 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 (corect code)

```
void main() {
	decltype(&"Te") x = &”CC”;
}
```

"auto" 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] &”