OOP

Gavrilut Dragos

Course 3

Summary

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

Initialization lists (recap)

Initialization lists

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

App.cpp

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

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

Assembly code generated

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

Initialization lists

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

App.cpp

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

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

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

Initialization lists

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

App.cpp

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

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

App.cpp

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

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

Initialization lists

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

App.cpp

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

```
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
```

- Constructors

Constructors

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

Constructors

- The ***constructor*** is defined as a function ***with the same name as the class*** and ***no return value***
- A class :
  - May have NO constructors
  - May have only one constructor
  - May have multiple constructors
  - May have constructors that are not ***public***

App.cpp

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

App.cpp

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

App.cpp

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

App.cpp

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

Type of constructors

Constructors:

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

App.cpp

```
class MyClass
{
    int x;
public:
    MyClass ();

    MyClass (const MyClass & objToCopyFrom);

    MyClass (const MyClass && objToMoveFrom);

};
```

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

Constructors

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

App.cpp

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

Constructors

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

App.cpp

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

MyClass global("global variable");

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

    MyClass local("local variable");

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

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

Constructors

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

App.cpp

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

MyClass global("global variable");

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

    MyClass local("local variable");

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

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

Constructors

- This code will compile correctly and produc

App.cpp

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

MyClass global("global variable");

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

    MyClass local("local variable");

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

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

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

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

Constructors

- This code will compile correctly and produc

App.cpp

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

MyClass global("global variable");

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

    MyClass local("local variable");

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

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

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

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

Constructors

- This code will compile correctly and produc

App.cpp

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

MyClass global("global variable");

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

    MyClass local("local variable");

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

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

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

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

Constructors

▶ This code will compile correctly and produc

App.cpp

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

MyClass global("global variable");

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

    MyClass local("local variable");

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

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

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

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

sizeof (MyClass)

Has *new* returned nullptr ?

Constructors

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

App.cpp

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

Constructors

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

App.cpp

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

```
push        ebp
mov         ebp,esp
sub         esp,44h
push        ebx
push        esi
push        edi
```

Stub

```
lea         ecx,[d]
call        Date::Date (0BB1325h)
```

Date d;

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

Stub

Constructors

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

App.cpp

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

Stub

```
push        ebp
mov         ebp,esp
sub         esp,44h
push        ebx
push        esi
push        edi
```

x = 10;

```
mov         dword ptr [this],ecx
mov         eax,dword ptr [this]
mov         dword ptr [eax],10
mov         eax,dword ptr [this]
```

Stub

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

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

Constructors

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

App.cpp

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

Constructors

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

App.cpp

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

Constructors

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

App.cpp

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

class Date
{
    MyClass m;
public:
    Date(): m(100) { }
};
void main()
{
    Date d;
}
```

Constructors

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

  A) Explicitly call that constructor in all of its defined constructors

App.cpp

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

class Date
{
    MyClass m;
public:
    Date() { }
};
void main()
{
    Date d;
}
```

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

- This code will NOT compile !!!

Constructors

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

  A) Explicitly call that constructor in all of its defined constructors

App.cpp

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

class Date
{
    MyClass m;
public:
    Date() : m(123) { }
};
void main()
{
    Date d;
}
```

- This code will compile properly

Constructors

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

  A) Explicitly call that constructor in all of its defined constructors

App.cpp

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

class Date
{
    MyClass m;
public:
    Date() : m(123) { }
    Date(int value) { }
};
void main()
{
    Date d;
}
```

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

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

Constructors

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

  A) Explicitly call that constructor in all of its defined constructors

App.cpp

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

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

- Now the code compiles correctly

Constructors

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

  B) Add a default constructor

App.cpp

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

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

- This code compiles correctly.

Constructors

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

  C) Remove all constructors

App.cpp

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

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

- This code compiles correctly.

Constructors

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

  D) Use initialization lists

App.cpp

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

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

OR

App.cpp

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

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

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

Constructors

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

App.cpp

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

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

Constructors

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

App.cpp

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

Tree t ➔ is the first data member from Object

Car c ➔ is the second data member from Object

Output:
Tree: oak
Car: Toyota
Animal: fox

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

Constructors

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

App.cpp

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

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

Constructors

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

App.cpp

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

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

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

Constructors

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

App.cpp

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

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

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

Const & Reference

- Data members

Const & Reference data members

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

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

App.cpp (A)

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

};

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

App.cpp (B)

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

};

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

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

Const & Reference data members

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

App.cpp (A)

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

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

App.cpp (B)

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

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

Const & Reference data members

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

App.cpp (A)

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

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

App.cpp (B)

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

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

Const & Reference data members

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

App.cpp (A)

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

}
```

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

App.cpp (B)

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

}
```

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

Const & Reference data members

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

App.cpp (A)

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

}
void main()
{
	Date d;
	Date d2(100);
}
```

App.cpp (B)

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

}
void main()
{
	Date d;
	Date d2(100);
}
```

Const & Reference data members

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

App.cpp

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

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

App.cpp

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

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

Const & Reference data members

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

App.cpp

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

}
void main()
{
	Date d;
	Date d2(100);
}
```

App.cpp

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

}
void main()
{
	Date d;
	Date d2(100);
}
```

Const & Reference data members

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

App.cpp (A)

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

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

App.cpp (B)

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

```
This code compiles !
```

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

Const & Reference data members

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

App.cpp (B)

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

Const & Reference data members

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

App.cpp (B)

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

Stub

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

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

Stub

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

C++ code translation:

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

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

Const & Reference data members

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

App.cpp (B)

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

Will print *123* to the screen

Const & Reference data members

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

App.cpp (B)

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

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

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

- Delegating constructor

Delegating constructor

- A constructor can call another constructor during its initialization.

App.cpp

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

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

    Object() : Object(0) {  }
};

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

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

Delegating constructor

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

App.cpp

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

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

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

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

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

Delegating constructor

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

App.cpp

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

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

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

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

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

gcc (Linux)
error: mem-initializer for ‘Object::y’ follows constructor
delegation

clang (MAC/OSX)
error: an initializer for a delegating constructor must
appear alone
    Object() : Object(0), y(0) {  }
               ^~~~~~~~~  ~~~~
```

Delegating constructor

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

App.cpp

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

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

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

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

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

Delegating constructor

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

App.cpp

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

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

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

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

Delegating constructor

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

App.cpp

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

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

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

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

Delegating constructor

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

App.cpp

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

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

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

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

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

Delegating constructor

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

App.cpp

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

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

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

Initialization lists for classes

Initialization lists for classes

- Classes and structures can be initialized using initialization lists:

App.cpp

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

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

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

Initialization lists for classes

- Classes and structures can be initialized using initialization lists:

App.cpp

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

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

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

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

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

Initialization lists for classes

- Classes and structures can be initialized using initialization lists:

App.cpp

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

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

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

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

Initialization lists for classes

- Classes and structures can be initialized using initialization lists:

App.cpp

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

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

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

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

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

Initialization lists for classes

- Classes and structures can be initialized using initialization lists:

App.cpp

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

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

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

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

Initialization lists for classes

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

App.cpp

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

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

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

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

Initialization lists for classes

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

App.cpp

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

    Data(int xx, int tt, const void * p) : x(xx), t(tt), m((const char *)p) {};
};
void main()
{
    Data d1{ true, 'A', "test" };
    Data d2 = { true, 'A', "test" };
}
```

- This code compiles correctly

Initialization lists for classes

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

App.cpp

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

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

- This code compiles correctly

Initialization lists

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

App.cpp

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

    Data(int xx, int tt, const void * p) : x(xx), t(tt), m((const char *)p) {};
};
void main()
{
    Data d1{};
    Data d2 = {};
}
```

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

Initialization lists for classes

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

App.cpp

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

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

App.cpp

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

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

Initialization lists for classes

- If no constructor is present:

App.cpp

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

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

Initialization lists for classes

- If no constructor is present:

App.cpp

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

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

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

Initialization lists for classes

- If no constructor is present:

App.cpp

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

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

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

Initialization lists for classes

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

App.cpp

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

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

Initialization lists for classes

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

App.cpp (A)

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

App.cpp (B)

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

App.cpp (C)

```
class Data
{
    int x;
public:
    Data(int value) :
        x(value) {}
};
void main()
{
    Data d = 4.5;
}
```

Initialization lists for classes

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

App.cpp

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

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

App.cpp

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

};

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

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

Initialization lists for classes

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

App.cpp

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

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

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

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

Initialization lists for classes

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

App.cpp

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

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

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

```
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
```

x=5

y=10.5

t = false (0)

- The compiler will add a new default con already in the class). That new construc "y" and "t"

Initialization lists for classes

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

App.cpp

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

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

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

Initialization lists for classes

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

App.cpp

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

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

```
mov      dword ptr [this],ecx
mov      eax,dword ptr [this]
mov      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 integrate the default initializa

Initialization lists for classes

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

App.cpp

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

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

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

Initialization lists for classes

- Initialization lists can also be used for pointers.

App.cpp

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

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

App.cpp

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

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

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

Initialization lists for classes

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

App.cpp

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

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

App.cpp

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

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

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

Initialization lists for classes

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

App.cpp

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

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

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

- Value Types

Value Types

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

Value Types: glvalue (generalized lvalue)

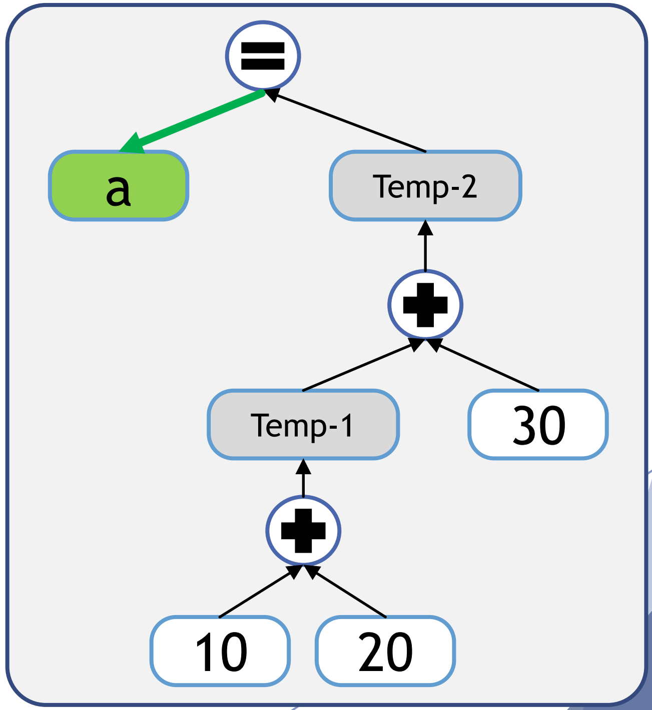
- A **glvalue** is an expression that results in an object
- Examples:
  1. Assignment ➔ *a = <expression>,* where “a” is a variable/data member (“a” in this context will be a **glvalue**). It’s valid for other type of assignments operators such as “+=, -= , *= , etc)
  2. Pre-increment/decrement ➔ *++a,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

Value Types: xvalue (eXpiring value)

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

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

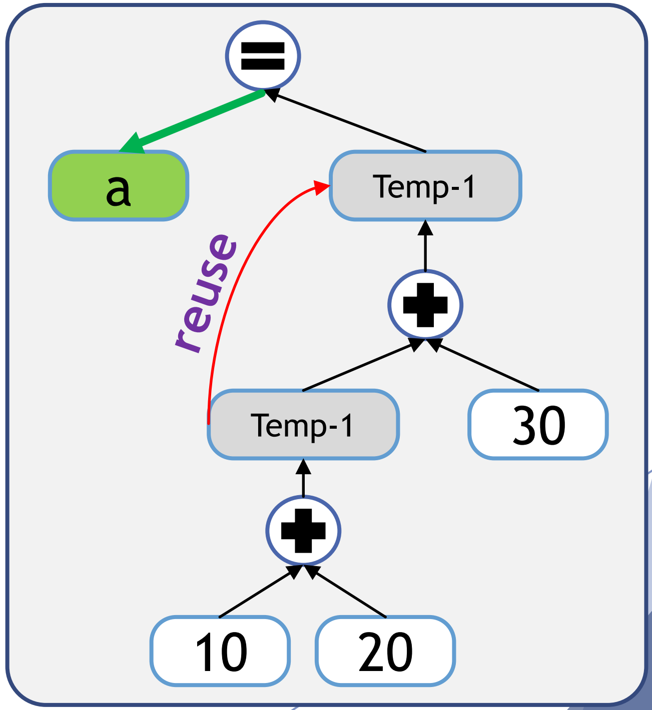
- This code will be evaluated in the following way:
  1. We first add "10" with "20"
  2. The result is a temporary value (Temp-1)
  3. Then we add Temp-1 with "30"
  4. The result is another temporary value (Temp-2)
  5. Finally – we copy the value from Temp-2 into "a"
- Both "Temp-1" and "Temp-2" are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



Value Types: xvalue (eXpiring value)

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

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



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

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

Value Types: prvalue (pure rare value)

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

Value Types: lvalue (left value)

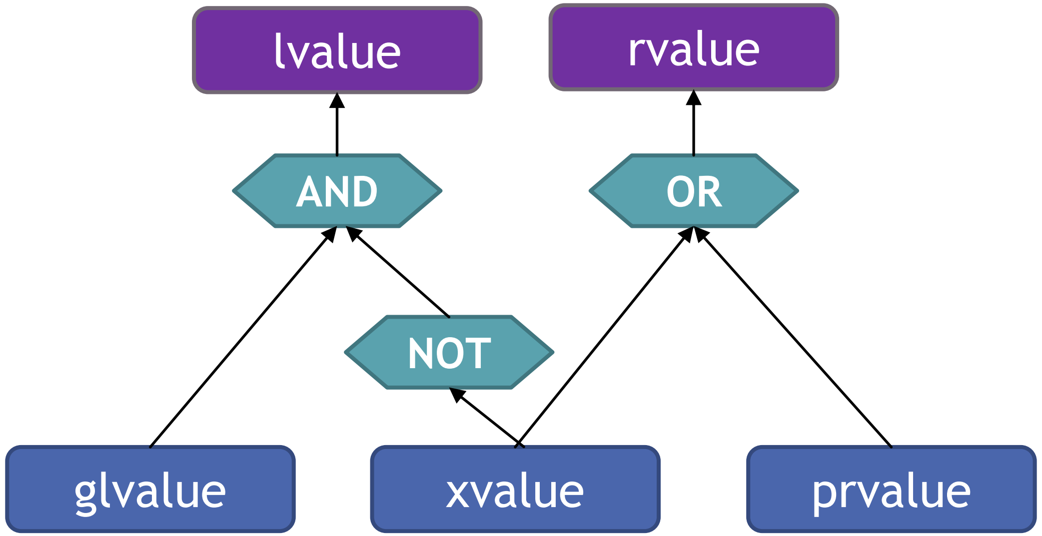
- A **lvalue** is a **glvalue** that is **NOT** an **xvalue**
- lvalue got its name because it refers mostly to the left location within an expression.

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

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

Value Types: rvalue (right value)

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



Copy & Move Constructors

Copy constructor

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

App.cpp

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

Copy Constructor

- It is usually used in the following way:

App.cpp

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

Copy Constructor

Copy constructor

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

App.cpp

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

Copy Constructor

-

App.cpp

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

```
lea     eax,[d]
push    eax
lea     ecx,[d2]
call    Date::Date
```

Copy constructor

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

App.cpp

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

void Process(Date d) { … }

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

In this case , a copy of "d" is made, and that copy is pass to function Process. More on this mechanisms on next course.

Copy constructor

- Similarly, if a function returns an object (not a reference or a pointer) the copy constructor is used.

App.cpp

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

Date Process()
{
	Date d(1);
	return d;
}
```

This is where the copy construction will be called. More on this topic on the next course

Copy constructor

- A copy constructor can be declared in two ways:
  - With a const parameter (this is the most generic usage)
  - With a non-const parameter

App.cpp

```
class Date
{
        int x;
public:
        Date(const Date &d) { x = d.x; }

        Date(Date &d) { x = d.x * 2; }

        Date(int v) { x = v; }
};
```

Both declarations are copy constructors.

- Some compilers might produce a warning in this case: ***"warning C4521: 'Date': multiple copy constructors specified"***
- It's best to use the copy constructor that uses a constant reference as a parameter (this is more generic, and any non-constant references can be converted to a const reference).

Copy constructor

- If both types of copy constructors are present (with const and non-const parameter), the compiler will choose the best fit.
- In this case, since “d” is not a constant, the non-constant form of the copy constructor will be used.

App.cpp

```
class Date
{
	int x;
public:
	Date(const Date &d) { x = d.x; }
	Date(Date &d) { x = d.x * 2; }
	Date(int v) { x = v; }
};

int main()
{
	Date d(1);
	Date d2 = d;
	return 0;
}
```

Copy constructor

- In this case, event if the “d” is not a constant, its non-const reference can be converted to a constant reference and then used the copy constructor with a constant parameter.

App.cpp

```
class Date
{
	int x;
public:
	Date(const Date &d) { x = d.x; }

	Date(int v) { x = v; }
};

int main()
{
	Date d(1);
	Date d2 = d;
	return 0;
}
```

Copy constructor

- In this case the compiler will call the copy constructor that has a “const” parameter.

App.cpp

```
class Date
{
	int x;
public:
	Date(const Date &d) { x = d.x; }
	Date(Date &d) { x = d.x * 2; }
	Date(int v) { x = v; }
};

int main()
{
	const Date d(1);

	Date d2 = d;

	return 0;
}
```

Copy constructor

- In this case the code will not compile. There a copy constructor defined, but it does not accept const parameters !

App.cpp

```
class Date
{
	int x;
public:

	Date(Date &d) { x = d.x * 2; }
	Date(int v) { x = v; }
};

int main()
{
	const Date d(1);

	Date d2 = d;

	return 0;
}
```

error C2440: 'initializing': cannot convert from 'const Date' to 'Date'
note: Cannot copy construct class 'Date' due to ambiguous copy constructors or no available copy constructor

Copy constructor

- This code will compile.
- Since there is no copy constructor defined, the compiler will generate a code that copies the data from “d” to “d2” (similar to what ***memcpy*** function does).

App.cpp

```
class Date
{
	int x;
public:


	Date(int v) { x = v; }
};

int main()
{
	const Date d(1);

	Date d2 = d;

	return 0;
}
```

```
mov   eax,dword ptr [d]
mov   dword ptr [d2],eax
```

Move constructor

- Let's analyze the following code:

App.cpp

```
char * DuplicateString(const char * string) {
    char * result = new char[strlen(string) + 1];
    memcpy(result, string, strlen(string) + 1);
    return result;
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }
int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Move constructor

- Let's analyze the following code :

App.cpp

```
char * DuplicateString(const char * string) {
    char * result = new char[strlen(string) + 1];
    memcpy(result, string, strlen(string) + 1);
    return result;
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }
int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

```
CTOR: Allocate sir to 00AB0610
COPY-CTOR: Copy sir from 00AB0610 to 00AB0648
```

Move constructor

- Let's analyze the following code :

App.cpp

```
char * DuplicateString(const char * strin
    char * result = new char[strlen(strin
    memcpy(result, string, strlen(string) + 1);
    return result;
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }
int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Step	What happened
1.	A temporary object is created (it's temporary because it is not assigned to any variable)

CTOR: Allocate sir to 00AB0610

Move constructor

- Let’s analyze the following code :

App.cpp

```
char * DuplicateString(const char * string
    char * result = new char[strlen(string
    memcpy(result, string, strlen(string)
    return result;
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }
int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Step	What happened
1.	A temporary object is created (it’s temporary because it is not assigned to any variable)
2.	That temporary object is sent to Get(…) function

CTOR: Allocate sir to 00AB0610

Move constructor

- Let’s analyze the following code :

App.cpp

```
char * DuplicateString(const char * string
    char * result = new char[strlen(string
    memcpy(result, string, strlen(string)
    return result;
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }
int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Step	What happened
1.	A temporary object is created (it’s temporary because it is not assigned to any variable)
2.	That temporary object is sent to Get(…) function
3.	Get(…) function returns it (so it calls the copy-ctor)

```
CTOR: Allocate sir to 00AB0610
```

Move constructor

- Let's analyze the following code :

App.cpp

```
char * DuplicateString(const char * string
    char * result = new char[strlen(string
    memcpy(result, string, strlen(string)
    return result;
}

class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %p to %p \n", d.sir, sir);
    }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }
int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Step	What happened
1.	A temporary object is created (it's temporary because it is not assigned to any variable)
2.	That temporary object is sent to Get(…) function
3.	Get(…) function returns it (so it calls the copy-ctor)
4.	Within the copy-ctor the string is copied againg

CTOR: Allocate sir to 00AB0610
COPY-CTOR: Copy sir from 00AB0610 to 00AB0648

Move constructor

So … what is the problem ?

- We allocate memory and we copy the same string twice !!!
- First as part of the constructor
- Second as part of the copy-constructor

This is not unusual; however, the first constructor creates a temporary object (an object that only exists during the evaluation of the following expression:

```
Date d = Get(Date("test"));
```

So … we know that we have allocated memory for an object that we can not control after the expression is evaluated.

Q: Do we need to allocate the memory twice ? (or can’t we just use the original memory that was allocated ?)

Move constructor

- A move constructor is declared using "&&" to refer to temporary value. It is mostly used to reuse an allocated memory.

App.cpp

```
class Date
{
	char * pointer;
public:
	Date(Date && d) { char * temp = d.pointer; d.pointer = nullptr; this->pointer = temp; }
};
```

- If no "move" constructor is provided, but a "copy" constructor exists, the compiler will use the copy-constructor. This is valid only for temporary values (such as an **xvalue**). Move constructor is never used for a glvalue or a rvalue.
- A move constructor can be used with a const parameter

App.cpp

```
class Date
{
	Date(const Date && d) { … }
};
```

However, as usually in the "move-constructor" the parameter received will be modified, the "const" form is not used.

Move constructor

- Let’s analyze the following code :

App.cpp

```
char * DuplicateString(const char * string) {
    …
}
class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %
    }

    Date(Date &&d) { printf("Move from %p to %p \n", &d, this);sir = d.sir;d.sir = nullptr; }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }
int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Move Constructor

Move constructor

- Let's analyze the following code :

App.cpp

```
char * DuplicateString(const char * string) {
    …
}
class Date
{
    char * sir;
public:
    Date(const Date &d) {
        sir = DuplicateString(d.sir); printf("COPY-CTOR: Copy sir from %
    }

    Date(Date &&d) { printf("Move from %p to %p \n", &d, this);sir = d.sir;d.sir = nullptr; }

    Date(const char * tmp)
    {
        sir = DuplicateString(tmp); printf("CTOR: Allocate sir to %p \n", sir);
    }
};

Date Get(Date d) { return d; }
int main()
{
    Date d = Get(Date("test"));
    return 0;
}
```

Move Constructor

CTOR: Allocate sir to 00AB0610
Move from 00AB0610 to 00D8FE04

- Constraints

Constraints

Singleton pattern.

- Problem: what if we want to model a class that can only have one instance ? The solution is to combine a private constructor with a static function:

App.cpp

```
class Object
{
    int value;
    static Object* instance;
    Object() { value = 0; }
public:
    static Object* GetInstance();
};
Object* Object::instance = nullptr;

Object* Object::GetInstance()
{
    if (instance == nullptr)
        instance = new Object();
    return instance;
}

void main()
{
    Object *obj = Object::GetInstance();
}
```

The default constructor is private – thus an object of this type can not be create !

As ***GetInstance*** is a static method of class Object, it can access any private constructors. However, as ***Object::instance*** is a static variable, the ***new*** operator will only be called once (when the first instance is requested → therefor the name Singleton).

Constraints

Singleton pattern.

- Problem: what if we want to model a class that can only have one instance ? The solution is to combine a private constructor with a static function:

App.cpp

```
class Object
{
    int value;
    static Object* instance;
    Object() { value = 0; }
public:
    static Object* GetInstance() { … }
};
Object* Object::instance = nullptr;

void main()
{
    Object *obj1 = Object::GetInstance();
    Object *obj2 = Object::GetInstance();
    Object *obj3 = Object::GetInstance();
}
```

```
Object* Object::GetInstance() {
    if (instance == nullptr)
        instance = new Object();
    return instance;
}
```

- Both ***obj1*** , ***obj2*** and ***obj3*** are in reality the same pointer. When ***obj1*** is first requested, Object::instance is first allocated, then it gets returned for ***obj2*** and ***obj3***.

Constraints

Singleton pattern.

- Problem: what if we want to model a class that can only have one instance ? The solution is to combine a private constructor with a static function:

App.cpp

```
class Object
{
    int value;
    static Object* instance;
    Object() { value = 0; }
public:
    static Object* GetInstance() { … }
};
Object* Object::instance = nullptr;

void main()
{
    Object obj1;
    Object * obj = new Object();
}
```

```
Object* Object::GetInstance() {
    if (instance == nullptr)
        instance = new Object();
    return instance;
}
```

error C2248: 'Object::Object': cannot access private member declared in class 'Object'
note: see declaration of 'Object::Object'
note: see declaration of 'Object'

- This code will not compile !

Constraints

- Private constructors can also be used with friend function. This is useful if we want the entire functionality of a class to have a limited availability (only a couple of classes can use its functionality).

App.cpp

```
class Object
{
    int value;
    Object(): value(0) { }
    friend class ObjectUser;
};
class ObjectUser
{
public:
    int GetValue();
};
int ObjectUser::GetValue()
{
    Object o;
    return o.value;
}
void main()
{
    ObjectUser ou;
    printf("%d\n", ou.GetValue());
}
```

- This code will compile !

Constraints

- Private constructors can also be used with friend function. This is useful if we want the entire functionality of a class to have a limited availability (only a couple of classes can use its functionality).

App.cpp

```
class Object
{
    int value;
    Object(): value(0) { }
    friend class ObjectUser;
};
class ObjectUser
{
public:
    int GetValue();
};
int ObjectUser::GetValue()
{
    Object o;
    return o.value;
}
void main()
{
    Object obj;
}
```

error C2248: 'Object::Object': cannot access private member declared in class 'Object'
note: see declaration of 'Object::Object'
note: see declaration of 'Object'

- This code will NOT compile !

Constraints

- Let's analyze the following code:

App.cpp

```
class Date
{
public:
	static int Suma(int x, int y) { return x + y; }
	static int Dif(int x, int y) { return x + y; }
	static int Mul(int x, int y) { return x * y; }
};


int main()
{
	Date d;
	printf("%d\n", Date::Suma(10, 20));
}
```

- Since class Data only has static functions, it makes no sense to allow creating instances of this class. However, with the current code, this is possible ! What can we do so that a programmer **CAN NOT** create an instance of type Data ?

Constraints

- Solution 1 → make the default constructor private:

App.cpp

```
class Date
{
	Date();
public:
	static int Suma(int x, int y) { return x + y; }
	static int Dif(int x, int y) { return x + y; }
	static int Mul(int x, int y) { return x * y; }
};

int main()
{
	Date d;
	printf("%d\n", Date::Suma(10, 20));
}
```

- This code will not compile. However, a static method can still create an instance of Data.

Constraints

- Solution 2 → use the keyword **delete**

App.cpp

```
class Date
{
public:
	Date() = delete;
	static int Suma(int x, int y) { return x + y; }
	static int Dif(int x, int y) { return x + y; }
	static int Mul(int x, int y) { return x * y; }
};

int main()
{
	Date d;
	printf("%d\n", Date::Suma(10, 20));
}
```

- In this case we are telling the compiler that there is NO default constructor and it (the compiler) should not create one by default.

Constraints

- Let's analyze the following code:

App.cpp

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

int main()
{
	Date d('0');

	return 0;
}
```

- This code works, due to promotion mechanism (‘a’ (a char) is promoted to an int).
- What can we do if we **do not want** to allow creating objects with a char parameter, but we **do want** to allow creating objects with an int parameter ?

Constraints

- The solution is similar as with the previous cases:

App.cpp

```
class Date
{
	int value;
public:
	Date(char x) = delete;
	Date(int x) { value = x; }
};

int main()
{
	Date d('0');

	return 0;
}
```

error C2280: 'Date::Date(char)': attempting to reference a deleted function
note: see declaration of 'Date::Date'
note: 'Date::Date(char)': function was explicitly deleted

- This code will not compile.
- Using the ***delete*** keyword in this manner tells the compiler that there is a constructor that has a ***char*** parameter, but it can not be used !

Constraints

- Let's analyze the following code:

App.cpp

```
class Date
{
	int value;
public:
	Date(int v) { value = v; }
};

int main()
{
	Date d = 100;
	return 0;
}
```

```
push	100
lea	ecx,[d]
call	Date::Date
```

- This code compiles. Due to the initialization lists methods, the constructor that has an *int* parameter is called.

Constraints

- Similarly

App.cpp

```
class Date
{
	int value;
public:
	Date(int v1, int v2, int v3) { value = v1+v2+v3; }
};
int main()
{
	Date d = { 1, 2, 3 };

	return 0;
}
```

```
push	3
push	2
push	1
lea	ecx,[d]
call	Date::Date
```

- This code compiles. Again, due to the initialization lists method, if there is constructor that has 3 int parameter, it will be used.
- What can we do to force the usage of the constructor and not the initialization list (e.g. if we want to have a code that is compatible with older standards → this will only work for C++11 and after).

Constraints

- Similarly

App.cpp

```
class Date
{
	int value;
public:
	explicit Date(int v1, int v2, int v3) { value = v1+v2+v3; }
};
int main()
{

	Date d = { 1, 2, 3 };

	return 0;
}
```

error C3445: copy-list-initialization of 'Date' cannot use an explicit constructor
note: see declaration of 'Date::Date'

- The solution is to use the keywork **explicit.** In this case we tell the compiler that it should use the constructor based initialization and not the initialization list method.
- This code does not compile. However, if we replace “***Date d = {1,2,3};***” with “***Date d(1, 2, 3);***” the code will compile !

Q & A