OOP

Gavrilut Dragos

Course 5

Summary

- Inheritance
- Virtual methods
- How virtual methods are modeled by C++ compiler
- Covariance
- Abstract classes (Interfaces)
- Memory alignment in case of inheritance

- Inheritance

Inheritance

- Inheritance is a process that transfer class proprieties (methods and members) from one class (often called the base class to another that inherits the base class – called derived class). The derive class may extend the base class by adding additional methods and/or members.
- Such an example will be the class Automobile, where we can define the following properties:
  - Number of doors
  - Number of wheels
  - Size
- From this class we can derive a particularization of the Automobile class (for example electrical machines) that besides the properties of the base class (doors, wheels, size, etc) has its own properties (battery lifetime).

Inheritance

- Inheritance in case of C++ classes can be simple or multiple:
  - Simple Inheritance

Simple

```
class <class_name>: <access modifier> <base class> { ... }
```

  - Multiple Inheritance

Multiple

```
class <class_name>: <access modifier> <base class 1> ,
                    <access modifier> <base class 2> ,
                    <access modifier> <base class 3> ,
                    ...
                    <access modifier> <base class n> ,
{ ... }
```

- The access modifier is optional and can be one of the following: (public / private or protected).
- If it is not specified, the default access modifier is private.

Inheritance

- Class “Derived” inherits members and methods from class “Base”. That is why we can call methods SetX and SetY from an instance of “Derived” class.

App.cpp

```
class Base
{
public:
	int x;
	void SetX(int value);
};
class Derived : public Base
{
	int y;
public:
	void SetY(int value);
};

void main()
{
	Derived d;
	d.SetX(100);
	d.x = 10;
	d.SetY(200);
}
```

Inheritance

- The following code will not compile. Class “Derived” inherits class “Base”, but member “x” from class Base is private (this means that it can not be accessed in class “Derived”).

App.cpp

```
class Base
{
private:
      int x;
};
class Derived : public Base
{
      int y;
public:
      void SetY(int value);
      void SetX(int value);
};
void Derived::SetX(int value)
{
      x = value;
}
void main()
{
      Derived d;
      d.SetX(100);
      d.SetY(200);
}
```

```
error C2248: 'Base::x': cannot access private member declared
in class 'Base'
note: see declaration of 'Base::x'
note: see declaration of 'Base'
```

Inheritance

- The solution for this case is to use the "***protected***" access modifier. A protected member is a member that can be access by classes that inherits current class, but it can not be accessed from outside the class.

App.cpp

```
class Base
{
protected:
	int x;
};
class Derived : public Base
{
	int y;
public:
	void SetY(int value);
	void SetX(int value);
};
void Derived::SetX(int value)
{
	x = value;
}
void main()
{
	Derived d;
	d.SetX(100);
	d.SetY(200);
}
```

Inheritance

- The code below will not compile. “*x*” is declared as ***protected*** – this means that it can be accessed in method ***SetX*** from a derived class, but it can not be accessed outside it’s scope (class).

App.cpp

```
class Base
{
protected:
	int x;
};
class Derived : public Base
{
	int y;
public:
	void SetY(int value);
	void SetX(int value);
};
void Derived::SetX(int value)
{
	x = value;
}
void main()
{
	Derived d;
	d.SetX(100);
	d.x = 100;
}
```

```
error C2248: 'Base::x': cannot access protected member declared
in class 'Base'
note: see declaration of 'Base::x'
note: see declaration of 'Base'
```

Inheritance

- The following table shows if a member with a specific access modifier can be access and in what conditions:

Access modifier	In the same class	In a derived class	Outside it's scope	Friend function in the base class	Friend function in the derived class
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	No	Yes	Yes
private	Yes	No	No	Yes	No

Inheritance

- The code below will not compile. “x” is a private member of “Base” therefor a friend function defined in “Derived” class can not access it.

App.cpp

```
class Base
{
private:
	int x;
};
class Derived : public Base
{
	int y;
public:
	void SetY(int value);
	void friend SetX(Derived &d);
};
void SetX(Derived &d)
{
	d.x = 100;
}
void main()
{
	Derived d;
	SetX(d);
}
```

Inheritance

- The solution is to change the access modifier of data member “x” from class “Base” from private to protected.

App.cpp

```
class Base
{
protected:
	int x;
};
class Derived : public Base
{
	int y;
public:
	void SetY(int value);
	void friend SetX(Derived &d);
};
void SetX(Derived &d)
{
	d.x = 100;
}
void main()
{
	Derived d;
	SetX(d);
}
```

Inheritance

- Be careful where you define the friend function. In the example below SetX friend function is declared in the “Derived” class. This means that it can access methods and data members from instances of “Derived” class and not other classes (e.g. Base class). The code will not compile.

App.cpp

```
class Base
{
private:
	int x;

};
class Derived : public Base
{
	int y;
public:
	void SetY(int value);
	void friend SetX(Base &d);
};
void SetX(Base &d)
{
	d.x = 100;
}
void main()
{
	Derived d;
}
```

Inheritance

- This code will work properly because the friend function is defined in class “Base”.

App.cpp

```
class Base
{
private:
	int x;
public:
	void friend SetX(Base &d);
};
class Derived : public Base
{
	int y;
public:
	void SetY(int value);

};
void SetX(Base &d)
{
	d.x = 100;
}
void main()
{
	Derived d;
}
```

Inheritance

- Access modifiers can also be applied to the inheritance relation.
- As a result, the members from the base class change their original access modifier in the derived class.

App.cpp

```
class Base
{
public:
	int x;
};
class Derived : public Base
{
	int y;
public:
	void SetY(int value) { ... }
};
void main()
{
	Derived d;
	d.x = 100;
}
```

- In this case, because “x” is public in the “Base” class, and the inheritance relation is also public, “x” will be public as well in the “Derived” class and will be accessible from outside the class scope.

Inheritance

- Access modifiers can also be applied to the inheritance relation.
- As a result, the members from the base class change their original access modifier in the derived class.

App.cpp

```
class Base
{
public:
	int x;
};
class Derived : private Base
{
	int y;
public:
	void SetY(int value) { ... }
};
void main()
{
	Derived d;
	d.x = 100;
}
```

- This code will not compile. “x” is indeed public in class “Base”, but since the inherit relation between class “Base” and class “Derived” is private, “x” will change its access modifier from public to private in class Derived and will not be accessible from outside its scope.
- However, if we are to create an instance of type “Base” we will be able to access “x” for that instance outside its scope.

Inheritance

- The rules that show how an access modifier is change if we change the access modifier of the inheritance relation are as follows:

Access modifier used for the inheritance relation → / Access modifier used for a data member or method	public	private	protected
public	public	private	protected
private	private	private	private
protected	protected	private	protected

private > protected > public

Inheritance

- Let's consider the following case:

App.cpp

```
class A
{
public:
    A() { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};
class B: public A
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};
int main()
{
    B b;
    return 0;
}
```

Output

```
ctor: A is called !
ctor: B is called !
dtor: B is called !
dtor: A is called !
```

Inheritance

- Let's consider the following case:

App.cpp

```
class A
{
public:
    A() { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};
class B: public A
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};
int main()
{
    B b;
    return 0;
}
```

Output

```
push    ebp
mov     ebp,esp
sub     esp,44h

mov     dword ptr [this],ecx
mov     ecx,dword ptr [this]
call    A::A (0DB14B5h)

push    offset string "ctor: B is called !\n"
call    _printf (0DB14A6h)
add     esp,4
mov     eax,dword ptr [this]

mov     esp,ebp
pop     ebp
ret
```

- The cod in YELLOW reflects the execution of the base constructor.

Inheritance

- Let's consider the following case:

App.cpp

```
class A
{
public:
    A() { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};
class B
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};
class C: public B, public A
{
public:
    C() { printf("ctor: C is called !\n"); }
    ~C() { printf("dtor: C is called !\n"); }
};
int main() {
    C c;
    return 0;
}
```

Output

```
ctor: B is called !
ctor: A is called !
ctor: C is called !
dtor: C is called !
dtor: A is called !
dtor: B is called !
```

- In case of multiple inheritance, the order of base classes is used when the constructor is called (in this case – first class B, then class A and finally class C)

Inheritance

- Let's consider the following case:

App.cpp

```cpp
class A
{
public:
    A() { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};
class B
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};
class C: public B, public A
{
public:
};
int main() {
    C c;
    return 0;
}
```

Output

```
ctor: B is called !
ctor: A is called !
dtor: A is called !
dtor: B is called !
```

- If no constructor is defined in class C, but there are at least one constructor defined in one of the class from which C is derived from, the compiler will create a default constructor that calls the constructor of class B followed by the constructor of class A.

Inheritance

- Let's consider the following case:

App.cpp

```
class A
{
public:
    A(int x) { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};
class B
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};
class C: public B, public A
{
public:
};
int main() {
    C c;
    return 0;
}
```

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

- This code will fail, as there is no explicit call to ***A::A(int)*** constructor.

Inheritance

- Let's consider the following case:

App.cpp

```
class A
{
public:
    A(int x) { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};
class B
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};
class C: public B, public A
{
public:
    C() : A(100) { printf("ctor: C is called !\n"); }
    ~C() { printf("dtor: C is called !\n"); }
};
int main() {
    C c;
    return 0;
}
```

Output

```
ctor: B is called !
ctor: A is called !
ctor: C is called !
dtor: C is called !
dtor: A is called !
dtor: B is called !
```

- The solution is to explicitly call the constructor of A in the member initializer list for C::C()

Inheritance

- Let's consider the following case:

App.cpp

```
class A
{
public:
    A(int x) { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};
class B
{
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};
class C: public B, public A
{
public:
    C() : A(100), B() { printf("ctor: C is called !\n"); }
    ~C() { printf("dtor: C is called !\n"); }
};
int main() {
    C c;
    return 0;
}
```

Output

```
ctor: B is called !
ctor: A is called !
ctor: C is called !
dtor: C is called !
dtor: A is called !
dtor: B is called !
```

Using this method ***WILL NOT CHANGE*** the order of the constructors (in this case, even if we call A(100) followed by B(), the compiler will still call ***B::B()*** first and then ***A::A(int)***

Inheritance

- Let's consider the following case:

App.cpp

```cpp
class A {
public:
    A(int x) { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};
class B {
public:
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};
class C {
public:
    C(bool n) { printf("ctor: C is called !\n"); }
    ~C() { printf("dtor: C is called !\n"); }
};
class D: public B, public A
{
    C c;
public:
    D(): c(true), A(100),B() { printf("ctor: D is called !\n"); }
    ~D() { printf("dtor: D is called !\n"); }
};
int main() {
    D d;
    return 0;
}
```

Output

```
ctor: B is called !
ctor: A is called !
ctor: C is called !
ctor: D is called !
dtor: D is called !
dtor: C is called !
dtor: A is called !
dtor: B is called !
```

Inheritance

- Let's consider the following case:

App.cpp

```cpp
struct A {
    A(int x) { printf("ctor: A is called !\n"); }
    ~A() { printf("dtor: A is called !\n"); }
};
struct B {
    B() { printf("ctor: B is called !\n"); }
    ~B() { printf("dtor: B is called !\n"); }
};
struct C {
    C(bool n) { printf("ctor: C is called !\n"); }
    ~C() { printf("dtor: C is called !\n"); }
};
struct D {
    D(bool n) { printf("ctor: D is called !\n"); }
    ~D() { printf("dtor: D is called !\n"); }
};
class E: public B, public A {
    C c;
    D d;
    int v1, v2;
public:
    E(): d(true), A(100), c(true), B(), v2(100), v1(20) { printf("ctor: E is called !\n"); }
    ~E() { printf("dtor: E is called !\n"); }
};
void main() {
    E e;
}
```

Output

```
ctor: B is called !
ctor: A is called !
ctor: C is called !
ctor: D is called !
-- v1 is initialized
-- v2 is initialized
ctor: E is called !
dtor: E is called !
dtor: D is called !
dtor: C is called !
dtor: A is called !
dtor: B is called !
```

Inheritance

- When inheriting from multiple classes, the general rule for calling constructors and destructors is as follows:

1. First all of the constructors from the base classes are called in the order of their inheriting definition (left-to-right)
2. All of the constructors from data members are called (again in their definition order – top-to-bottom)
3. Then the constructor initialization value for data members (basic types, references, constants) are used in the order they are defined in that class.
4. Finally, the code of the constructor of the class is called.

- Destructors are called in a reverse way (starting from point 4 to point 1).

Tested with:

- cl.exe: 19.16.27030.1
- Params: /permissive- /GS- /analyze- /W3 /Zc:wchar_t /ZI /Gm- /Od /sdl /Fd"Debug\vc141.pdb" /Zc:inline /fp:precise /D "WIN32" /D "_DEBUG" /D "_CONSOLE" /D "_UNICODE" /D "UNICODE" /errorReport:prompt /WX- /Zc:forScope /RTCu /arch:IA32 /Gd /Oy- /MDd /FC /Fa"Debug\" /nologo /Fo"Debug\" /Fp"Debug\TestCpp.pch" /diagnostics:classic

- Virtual methods

Virtual methods

App.cpp

```
class A
{
public:
	int a1, a2, a3;
	void Set() { printf("A"); }
};
class B: public A
{
public:
	int b1, b2;
	void Set() { printf("B"); }
};
void main()
{
	B b;
	b.Set();
}
```

- This code prints “***B***” on the screen. From the inheritance point of view, both ***A*** and ***B*** class have the same method called ***Set***
- In this case it is said that class ***B*** hides method ***Set*** from class ***A***

Virtual methods

App.cpp

```
class A
{
public:
	int a1, a2, a3;
	void Set() { printf("A"); }
};
class B: public A
{
public:
	int b1, b2;
	void Set() { printf("B"); }
};
void main()
{
	B b;
	A* a = &b;
	a->Set();
}
```

- In this case, the code will print "*A*" on the screen, because we are using a pointer of type ***A****
- However, in reality, "a" pointer points to an object of type ***B*** → so the expected result <u>should be</u> that the product will print "***B***" and not "***A***"
- So … what can we do to change this behavior ?

Virtual methods

App.cpp

```
class A
{
public:
	int a1, a2, a3;
	virtual void Set() { printf("A"); }
};
class B: public A
{
public:
	int b1, b2;
	void Set() { printf("B"); }
};
void main()
{
	B b;
	A* a = &b;
	a->Set();
}
```

- The solution is to use "**virtual**" keyword in from of a method - definition
- If we do this, the program will print "**B**"
- In this case, it is said that class B ***overrides*** method Set from class A
- **Using virtual keyword makes a method to be part of the instance !**

Virtual methods

Virtual methods can be used for:

- Polymorphism
- Memory deallocation (virtual destructor)
- Anti-debugging techniques

Virtual methods

Polymorphism = the ability to access instances of different classes through the same interface. In particular to C++, this translates into the ability to automatically convert (cast) a pointer to a certain class to its base class.

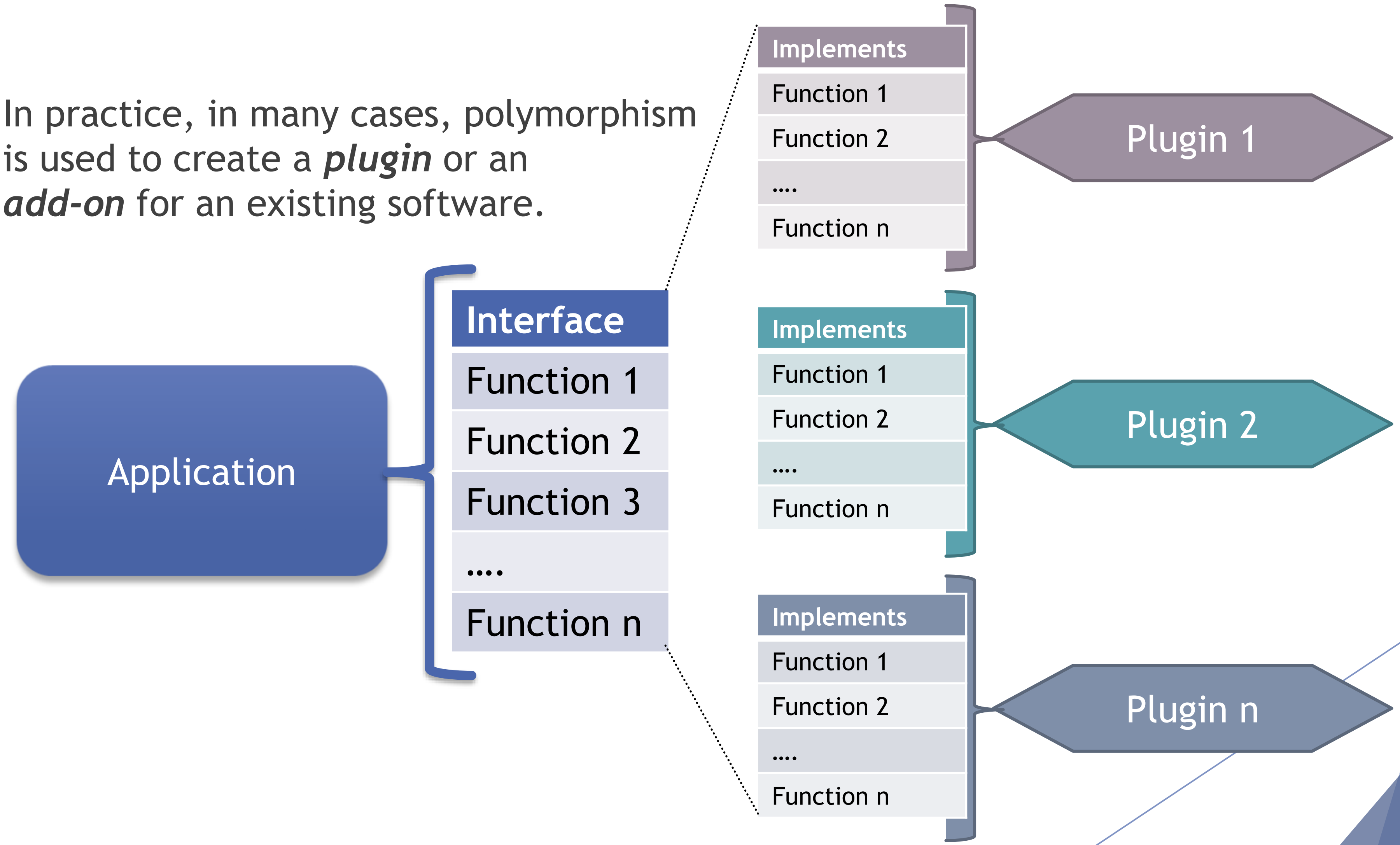
App-1.cpp

```
class Figure {
    public: virtual void Draw() { printf("Figure");  }
};
class Circle: public Figure {
    public: void Draw() { printf("Circle"); }
};
class Square: public Figure {
    public: void Draw() { printf("Square"); }
};
void main()
{
      Figure *f[2];
      f[0] = new Circle();
      f[1] = new Square();
      for (int index = 0;index<2;index++)
            f[index]->Draw();
}
```

- After the execution this code will print on the screen "**Circle**" and "**Square**".
- If we haven't uses ***virtual*** specifier, the program would have printed "**Figure**" twice !

Virtual methods

In practice, in many cases, polymorphism is used to create a ***plugin*** or an ***add-on*** for an existing software.



Virtual methods

In particular for C++ language, ***virtual*** specifier can be used as a specifier for destructors.
Let's analyze the following case:

App-1.cpp

```
class Figure {
    public: virtual void Draw() { printf("Figure");  }
    public: ~Figure() { printf("Delete Figure\n"); }
};
class Circle: public Figure {
    public: void Draw() { printf("Circle"); }
    public: ~Circle() { printf("Delete Circle"); }
};
class Square: public Figure {
    public: void Draw() { printf("Square"); }
    public: ~Square() { printf("Delete Square"); }
};
void main() {
      Figure *f[2];
      f[0] = new Circle();
      f[1] = new Square();
      for (int index = 0;index<2;index++)
            delete (f[index]);
}
```

- After this code gets executed, the following texts will be printed on the screen:
  "**Delete Figure**"
  "**Delete Figure**".
- What would happen if both ***Circle*** and ***Square*** classes allocate some memory ?

Virtual methods

In particular for C++ language, ***virtual*** specifier can be used as a specifier for destructors.
Let's analyze the following case:

App-1.cpp

```
class Figure {
    public: virtual void Draw() { printf("Figure");  }
    public: virtual ~Figure() { printf("Delete Figure\n"); }
};
class Circle: public Figure {
    public: void Draw() { printf("Circle"); }
    public: ~Circle() { printf("Delete Circle"); }
};
class Square: public Figure {
    public: void Draw() { printf("Square"); }
    public: ~Square() { printf("Delete Square"); }
};
void main() {
      Figure *f[2];
      f[0] = new Circle();
      f[1] = new Square();
      for (int index = 0;index<2;index++)
            delete (f[index]);
}
```

- The solution is to declare de destructor as ***virtual***. As a result, the destructor for actual class will be called, fallowed by the destructor of the base class
- The following text will be printed:
  Delete Circle
  Delete Figure
  Delete Square
  Delete Figure

Virtual methods

Let's analyze the following case:

App-1.cpp

```
class A
{
public:
    virtual bool Odd(int x) { return x % 2 == 0; }
};
class B : public A
{
public:
    virtual bool Odd(char x) { return x % 3 == 0; }
};
int main() {
    A* a = new B();
    printf("%d\n", a->Odd(3));
    return 0;
}
```

- ***Odd*** is a virtual function – however, class B **does not *override*** it (as it uses ***char*** as the first parameter instead of ***int***). As a result, class B will have 2 ***Odd*** methods and a->Odd will call the one with an ***int*** parameter. Upon execution, value ***false*** (0) is written to the screen.

Virtual methods

Let's analyze the following case:

App-1.cpp

```
class A
{
public:
    virtual bool Odd(int x) { return x % 2 == 0; }
};
class B : public A
{
public:
    virtual bool Odd(char x) { return x % 3 == 0; }
};
int main() {
    A* a = new B();
    printf("%d\n", a->Odd(3));
    return 0;
}
```

To override a virtual function, one must use the SAME method signature !

- *Odd* is a virtual function – however, class B does not ***override*** it (as it uses ***char*** as the first parameter instead of ***int***). As a result, class B will have 2 ***Odd*** methods and a->Odd will call the one with an ***int*** parameter. Upon execution, value ***false*** (0) is written to the screen.

Virtual methods

Let's analyze the following case:

App-1.cpp

```
class A
{
public:
    virtual bool Odd(int x) { return x % 2 == 0; }
};
class B : public A
{
public:
    virtual bool Odd(char x)  override  { return x % 3 == 0; }
};
int main() {
    A* a = new B();
    printf("%d\n", a->Odd(3));
    return 0;
}
```

error C3668: 'A::Odd': method with override specifier 'override' did not override any base class methods

- Assuming that , in reality, the intent was to override ***Odd*** method, then one way of making sure that this kind of mistakes will not happen is to use the ***override*** keyword (added with C++11 standard). As a result, this code will not compile as it is expected that method Odd to have the same signature !!!

Virtual methods

Let’s analyze the following case:

App-1.cpp

```
class A
{
public:
    virtual bool Odd(int x) { return x % 2 == 0; }
};
class B : public A
{
public:
    virtual bool Odd(int x) override { return x % 3 == 0; }
};
int main() {
    A* a = new B();
    printf("%d\n", a->Odd(3));
    return 0;
}
```

- Now the code compiles and prints “1” (true) on the screen.

Virtual methods

Let's consider the following code:

App-1.cpp

```
struct A {
    virtual bool Odd(int x) = 0;
};
struct B : public A {
    virtual bool Odd(int x) { return x % 2 == 0; }
};
struct C : public B {
    virtual bool Odd(int x) { return x % 3 == 0; }
};
int main() {
    A* a = new C();
    printf("%d\n", a->Odd(3));
    return 0;
}
```

- This program runs and prints value 1 (True) → even if 3 is not an odd number.
- The reason why this could happen is that method ***Odd*** was overridden in class C (keep in mind that we have used ***struct*** in this example to show that the behavior is identical to the one from ***class***).
- What can we do if we want to make sure that ***Odd*** method from class B **can not** be overridden ?

Virtual methods

Let’s consider the following code:

App-1.cpp

```
struct A {
    virtual bool Odd(int x) = 0;
};
struct B : public A {
    virtual bool Odd(int x) final { return x % 2 == 0; }
};
struct C : public B {
    virtual bool Odd(int x) { return x % 3 == 0; }
};
int main() {
    A* a = new C();
    printf("%d\n", a->Odd(3));
    return 0;
}
```

error C3248: 'B::Odd': function declared as 'final' cannot be overridden by 'C::Odd'

- The solution is to use the specifier ***final*** after the declaration of a virtual function. This tells the compiler that other classes that inherit current class can not **override** that method.

Virtual methods

Let's consider the following code:

App-1.cpp

```
struct A {
    virtual bool Odd(int x) = 0;
};
struct B : public A {
    virtual bool Odd(int x)  override final  { return x % 2 == 0; }
};
struct C : public B {
};
int main() {
    A* a = new C();
    printf("%d\n", a->Odd(3));
    return 0;
}
```

- It is possible to use both ***override*** and ***final*** specifiers when declaring a method.
- In this case their meaning is:
  - ***override*** ➔ The purpose of this method is to override the existing method from the base class (in this case, it overrides ***A::Odd***)
  - ***final*** ➔ Other classes that might inherit class B can not override this method.

Virtual methods

Let’s consider the following code:

App-1.cpp

```
struct A {
    virtual bool Odd(int x) = 0;
};
struct B  final  : public A
{
    virtual bool Odd(int x)  override { return x % 2 == 0; }
};
struct C : public B {
};
int main() {
    A* a = new C();
    printf("%d\n", a->Odd(3));
    return 0;
}
```

error C3246: 'C': cannot inherit from 'B' as it has been declared as 'final'

- *final* specifier can also be used directly in the class/struct definition. In this case , it’s meaning is that inheritance from class B is NOT possible.
- This code will not compile !

- How virtual methods are modeled by C++ compiler

How virtual methods are modeled by C++ compiler

- Let’s analyze the following two programs. Their only difference is the usage of ***virtual*** in case of APP-2.

App-1.cpp	App-2.cpp

<pre>class A
{
public:
 int a1, a2, a3;
 void Set() { printf("A"); }
};
void main()
{
 printf("%d",sizeof(A));
}</pre>
	<pre>class A
	{
	public:
	 int a1, a2, a3;
	 virtual void Set() { printf("A"); }
	};
	void main()
	{
	 printf("%d",sizeof(A));
	}</pre>

- When executed, APP-1 will print “**12**” and App-2 will print “**16**” (for x86 architecture). If we run the same App-2 on x64 it will print “**24**”
- **Why ?**

How virtual methods are modeled by C++ compiler

- Using the ***virtual*** keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called ***vfptr*** and if added is the first pointer in the class.

App-1.cpp

```
class A
{
public:
	int a1, a2, a3;
	void Set() { printf("A"); }
};
void main()
{
	A a;
}
```

Memory

How virtual methods are modeled by C++ compiler

- Using the ***virtual*** keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called ***vfptr*** and if added is the first pointer in the class.

App-1.cpp

```
class A
{
public:
	int a1, a2, a3;
	void Set() { printf("A"); }
};
void main()
{
	A a;
}
```

Memory
…
A::Set()

How virtual methods are modeled by C++ compiler

- Using the ***virtual*** keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called ***vfptr*** and if added is the first pointer in the class.

App-1.cpp

```
class A
{
public:
	int a1, a2, a3;
	void Set() { printf("A"); }
};
void main()
{
	A a;
}
```

Memory
…
A::Set()
…
main()

How virtual methods are modeled by C++ compiler

- Using the ***virtual*** keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called ***vfptr*** and if added is the first pointer in the class.

App-1.cpp

```
class A
{
public:
	int a1, a2, a3;
	void Set() { printf("A"); }
};
void main()
{
	A a;
}
```

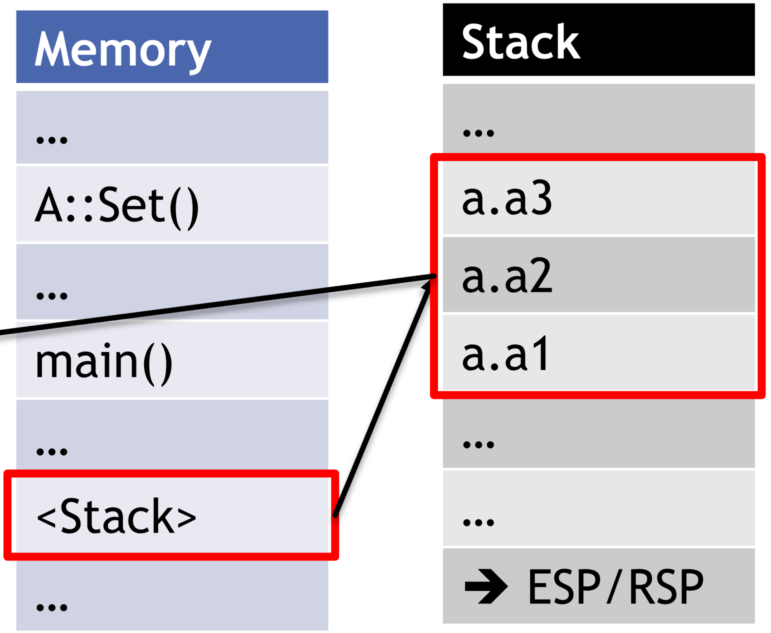
Memory
…
A::Set()
…
main()
…
<Stack>
…

How virtual methods are modeled by C++ compiler

- Using the ***virtual*** keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called ***vfptr*** and if added is the first pointer in the class.

App-1.cpp

```
class A
{
public:
        int a1, a2, a3;
        void Set() { printf("A"); }
};
void main()
{
      A a;
}
```

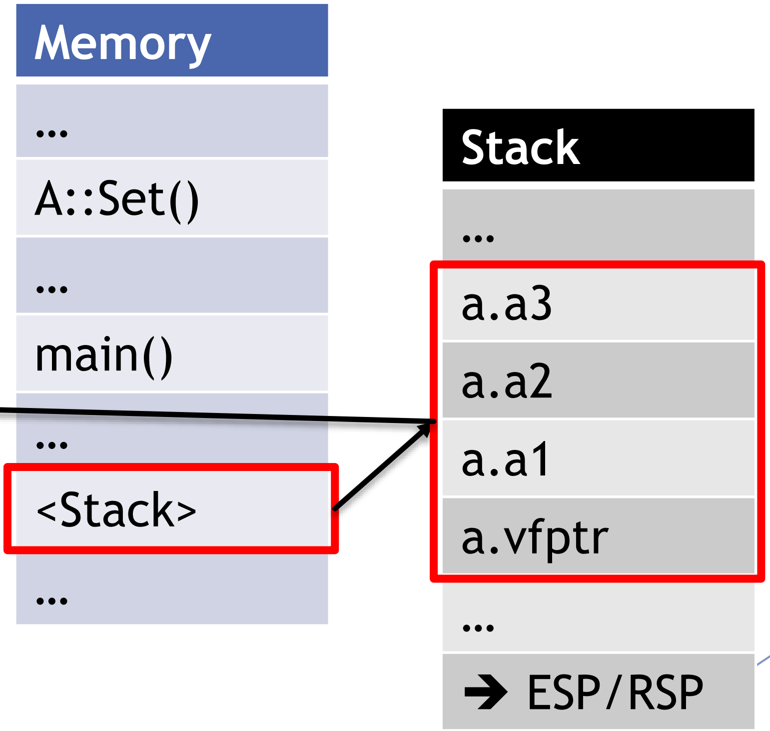


How virtual methods are modeled by C++ compiler

- Using the ***virtual*** keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called ***vfptr*** and if added is the first pointer in the class.

App-2.cpp

```
class A
{
public:
        int a1, a2, a3;
        virtual void Set() { printf("A"); }
};
void main()
{
        A a;
}
```

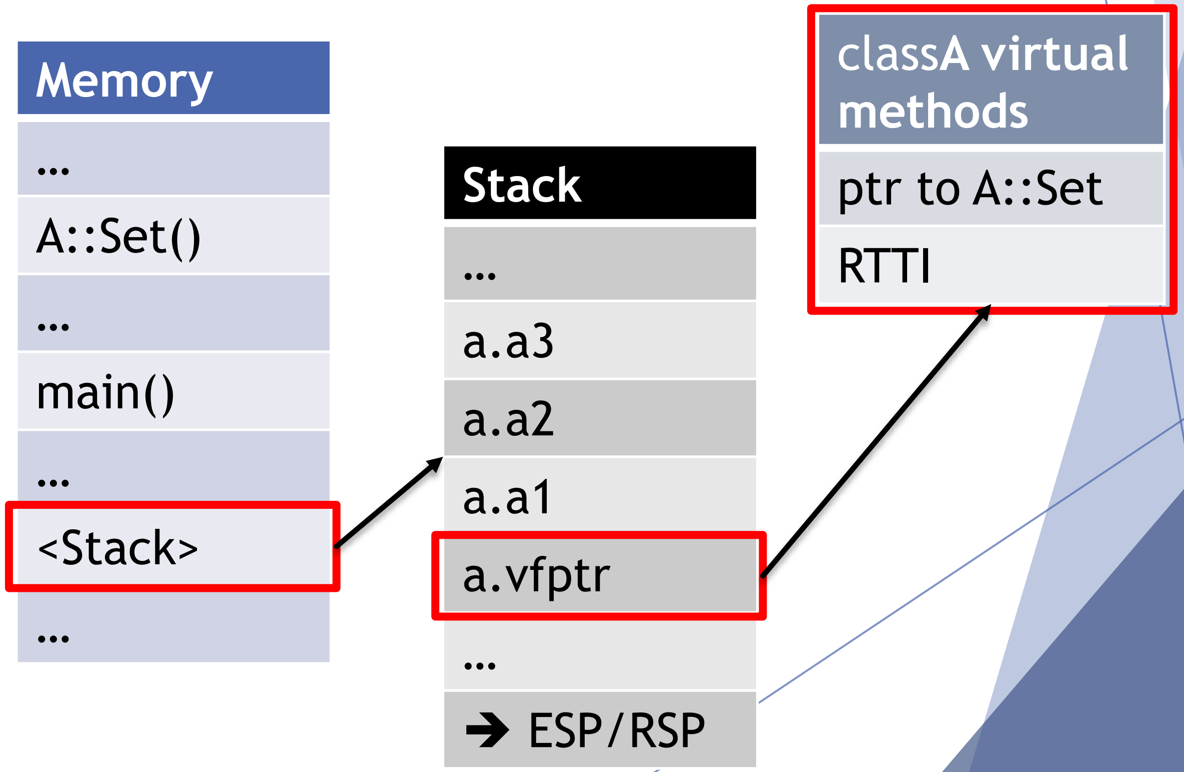


How virtual methods are modeled by C++ compiler

- Using the ***virtual*** keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called ***vfptr*** and if added is the first pointer in the class.

App-2.cpp

```
class A
{
public:
	int a1, a2, a3;
	virtual void Set() { printf("A"); }
};
void main()
{
	A a;
}
```

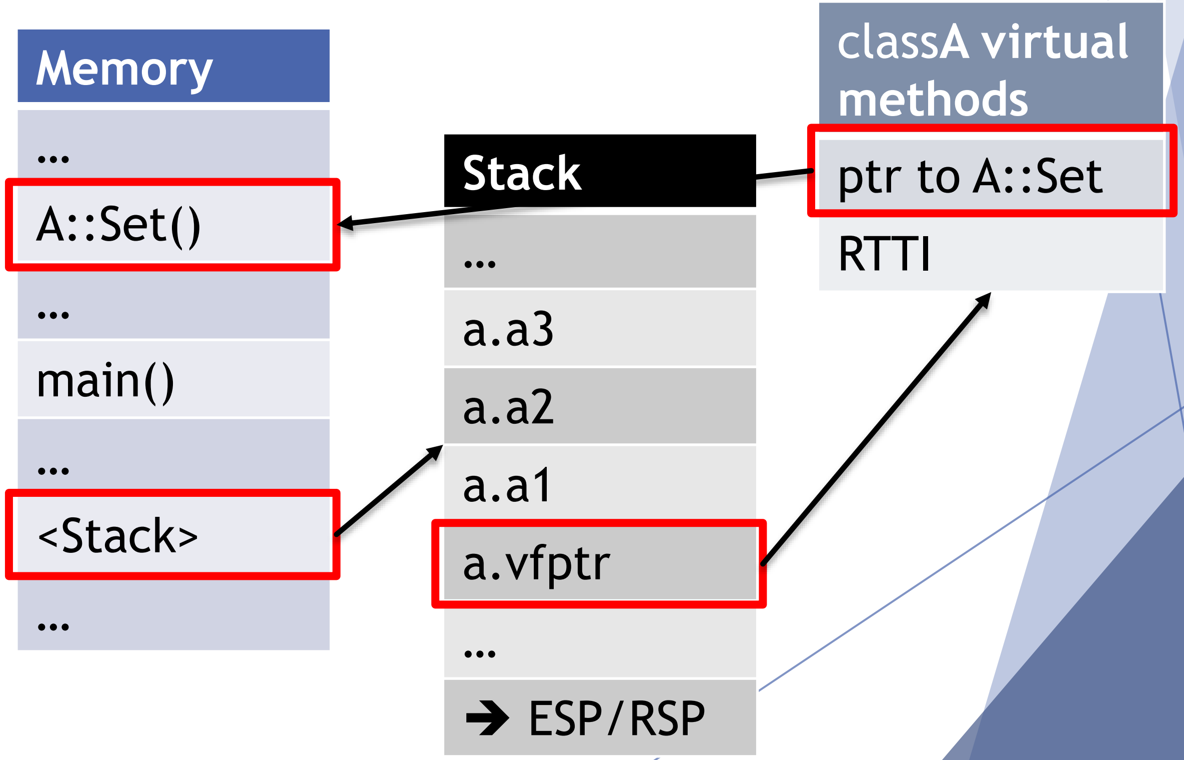


How virtual methods are modeled by C++ compiler

- Using the ***virtual*** keyword will force the compiler to modify the structure of any class by adding another data member (a pointer to a list of pointers to a function). This pointer is called ***vfptr*** and if added is the first pointer in the class.

App-2.cpp

```cpp
class A
{
public:
        int a1, a2, a3;
        virtual void Set() { printf("A"); }
};
void main()
{
        A a;
}
```



How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to be certain that ***vfptr*** pointer is set correctly. As such, any constructor is modified to include the code that sets up the ***vfptr*** pointer. If no constructor is present, the default one will be created automatically.

App.cpp

```
class A
{
public:
	int x, y;
	int Calcul() { return x+y; }
};
void main()
{
	A a;
	a.x = 1;
	a.y = 2;
}
```

Disasm

```
    A a;
    a.x = 1;
mov          dword ptr [ebp-12],1
    a.y = 2;
mov          dword ptr [ebp-8],2
```

- In this case, there no default constructor defined and no need for the compiler to provide one automatically (e.g. virtual methods, const or reference data members, etc).

How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to make certain that ***vfptr*** pointer is set correctly. As such, any constructor is modified to include the code that sets up the ***vfptr*** pointer. If no constructor is present, the default one will be created automatically.

App.cpp

```
class A
{
public:
	int x, y;
	int Calcul() { return x+y; }
	A() { x = y = 0; }
};
void main()
{
	A a;
	a.x = 1;
	a.y = 2;
}
```

Disasm

```
    A a;
lea            ecx,[ebp-16]
call           A::A
    a.x = 1;
mov            dword ptr [ebp-16],1
    a.y = 2;
mov            dword ptr [ebp-12],2
```

- In this case, there is a constructor that will be called when “a” is created.

How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to make certain that ***vfptr*** pointer is set correctly. As such, any constructor is modified to include the code that sets up the ***vfptr*** pointer. If no constructor is present, the default one will be created automatically

App.cpp

```
class A
{
public:
      int x, y;
      int Calcul() { return x+y; }
      A() { x = y = 0; }
};
void main()
{
      A a;
      a.x = 1;
      a.y = 2;
}
```

Disasm (A::A)

```
push        ebp
mov         ebp,esp
mov         dword ptr [ebp-8],ecx // EBP-8=this
mov         eax,dword ptr [ebp-8]
mov         dword ptr [eax+4],0  // this->y = 0
mov         ecx,dword ptr [ebp-8]
mov         dword ptr [ecx],0 // this->x = 0
mov         eax,dword ptr [ebp-8]
pop         ebp
ret
```

- In this case, there is a default constructor and the code from the default constructor will be called when object “a” is created.

How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to make certain that ***vfptr*** pointer is set correctly. As such, any constructor is modified to include the code that sets up the ***vfptr*** pointer. If no constructor is present, the default one will be created automatically

App.cpp

```
class A
{
public:
	int x, y;
	virtual int Calcul() {return x+y;}
};
void main()
{
	A a;
	a.x = 1;
	a.y = 2;
}
```

Disasm

```
    A a;
lea           ecx,[ebp-20]
call          A::A
    a.x = 1;
mov           dword ptr [ebp-16],1
    a.y = 2;
mov           dword ptr [ebp-12],2
```

- In this case, even if no constructor is defined, the compiler will automatically create one to initialize the ***vfptr*** pointer (this is required because ***Calcul*** is a virtual method).

How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to make certain that ***vfptr*** pointer is set correctly. As such, any constructor is modified to include the code that sets up the ***vfptr*** pointer. If no constructor is present, the default one will be created automatically

App.cpp

```
class A
{
public:
        int x, y;
        virtual int Calcul() {return x+y;}
};
void main()
{
        A a;
        a.x = :
        a.y = :
}
```

Disasm

```
    A a;
lea         ecx,[ebp-20]
call        A::A
    a.x = 1;
            ebp-16],1
            ebp-12],2
```

Disasm

```
push        ebp
mov         ebp,esp
mov         dword ptr [ebp-8],ecx
mov         eax,dword ptr [ebp-8]
mov         dword ptr [eax], A-virtual-fnc-list
mov         eax,dword ptr [ebp-8]
mov         esp,ebp
pop         ebp
ret
```

Memory address where a list of pointers to virtual functions is (in this case only one method: *Calcul*)

How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to make certain that ***vfptr*** pointer is set correctly. As such, any constructor is modified to include the code that sets up the ***vfptr*** pointer. If no constructor is present, the default one will be created automatically

App.cpp

```
class A
{
public:
        int x, y;
        virtual int Calcul() {return x+y;}
        A() { x = y = 0; }
};
void main()
{
        A a;
        a.x = 1;
        a.y = 2;
}
```

Disasm

```
    A a;
lea             ecx,[ebp-20]
call            A::A
    a.x = 1;
mov             dword ptr [ebp-16],1
    a.y = 2;
mov             dword ptr [ebp-12],2
```

- If a constructor exists, it will be modified (in a similar manner to the change that is done for const/references data members).

How virtual methods are modeled by C++ compiler

- Whenever a virtual method is added, the compiler needs to make certain that ***vfptr*** pointer is set correctly. As such, any constructor is modified to include the code that sets up the ***vfptr*** pointer. If no constructor is present, the default one will be created automatically.

App.cpp

```
class A
{
public:
	int x, y;
	virtual int Calcul() {return x+y;}
	A() { x = y = 0; }
};
void main()
{
	A a;
	a.x = 1;
	a.y = 2;
}
```

Disasm A::A

```
push        ebp
mov         ebp,esp
mov         dword ptr [ebp-8],ecx
mov         eax,dword ptr [ebp-8]
mov         dword ptr [eax],addr virt fnc
mov         eax,dword ptr [ebp-8]
mov         dword ptr [eax+8],0
mov         ecx,dword ptr [ebp-8]
mov         dword ptr [ecx+4],0
mov         eax,dword ptr [ebp-8]
mov         esp,ebp
pop         ebp
ret
```

- The code colored in **blue** is the code added by the compiler to initialize the **vfptr** pointer.

How virtual methods are modeled by C++ compiler

- The code added by the compiler to initialize the ***vfptr*** pointer will be added for every defined constructor.

App.cpp

```
class A
{
public:
	int x, y;
	virtual int Calcul() {return x+y;}
	A() { x = y = 0; }
	A(const A& a) { x = a.x; y = a.y; }
};
void main()
{
	A a;
	A a2 = a;
}
```

In this case the code for ***vfptr*** initialization will be added for both the default constructor and the copy constructor.

How virtual methods are modeled by C++ compiler

- However, in case of the ***assignment operator*** the compiler will not add any special code to initialize the ***vfptr*** pointer.

App.cpp

```
class A
{
public:
	int x, y;
	virtual int Calcul() {return x+y;}
	A() { x = y = 0; }
	A&  operator = (A &a) { x = a.x; y = a.y; return *this;}
};
void main()
{
	A a;
	A a2;
	a2 = a;
}
```

How virtual methods are modeled by C++ compiler

- A virtual method is called using its reference from the ***vfptr*** table only if the object is a pointer.

App.cpp	Disasm

<pre>class A
{
public:
 int x, y;
 virtual int Calcul() {return x+y;}
 A() { x = y = 0; }
};
void main()
{
 A a;
 a.x = 1;
 a.y = 2;
 a.Calcul();
}</pre>
	<pre> A a;
	lea ecx,[a]
	call A::A
	 a.x = 1;
	mov dword ptr [ebp-10h],1
	 a.y = 2;
	mov dword ptr [ebp-0Ch],2
	 a.Calcul();
	lea ecx,[a]
	call A::Calcul</pre>

- In this case, even if ***Calcul*** method is *virtual* as it called directly with an object , the compiler will not generate code that will find out its address from the ***vfptr*** table (it will use the method ***Calcul*** exact address).

How virtual methods are modeled by C++ compiler

- A virtual method is called using its reference from the ***vfptr*** table only if the object is a pointer.

App.cpp

```
class A
{
public:
        int x, y;
        virtual int Calcul() {return x+y;}
        A() { x = y = 0; }
};
void main()
{
        A a;
        a.x = 1;
        a.y = 2;
        A* a2 = &a;
        a2->Calcul();
}
```

Disasm

```
            A a;
lea             ecx,[a]
call            A::A
            a.x = 1;
mov             dword ptr [ebp-10h],1
            a.y = 2;
mov             dword ptr [ebp-0Ch],2
            A* a2 = &a;
lea             eax,[a]
mov             dword ptr [a2],eax
            a2->Calcul();
mov             eax,dword ptr [a2]     ; EAX = address of a2
mov             edx,dword ptr [eax]    ; EDX = address of VFPTR
mov             ecx,dword ptr [a2]
mov             eax,dword ptr [edx]    ; EAX = address of first function from VFPTR
call            eax
```

- In this case ***vfptr*** is used to find out ***Calcul*** method address.

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
	int x;
	virtual int Calcul() {return 0;}
	A() { x = 0; }
};
void main()
{
	A a;
	a.x = 1;
	a.y = 2;
	A* a2 = &a;
	a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {

};
```

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
	int x;
	virtual int Calcul() {return 0;}
	A() { x = 0; }
};
void main()
{
	A a;
	a.x = 1;
	a.y = 2;
	A* a2 = &a;
	a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {
	int (*Calcul) ();
};
```

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
	int x;
	virtual int Calcul() {return 0;}
	A() { x = 0; }
};
void main()
{
	A a;
	a.x = 1;
	a.y = 2;
	A* a2 = &a;
	a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {
	int (*Calcul) ();
};
class A {
public:

	int x;
```

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
	int x;
	virtual int Calcul() {return 0;}
	A() { x = 0; }
};
void main()
{
	A a;
	a.x = 1;
	a.y = 2;
	A* a2 = &a;
	a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {
	int (*Calcul) ();
};
class A {
public:
	A_VirtualFunctions *vfPtr;
	int x;
```

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
	int x;
	virtual int Calcul() {return 0;}
	A() { x = 0; }
};
void main()
{
	A a;
	a.x = 1;
	a.y = 2;
	A* a2 = &a;
	a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {
	int (*Calcul) ();
};
class A {
public:
	A_VirtualFunctions *vfPtr;
	int x;
	int A_Calcul() { return 0; }
```

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
	int x;
	virtual int Calcul() {return 0;}
	A() { x = 0; }
};
void main()
{
	A a;
	a.x = 1;
	a.y = 2;
	A* a2 = &a;
	a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {
	int (*Calcul) ();
};
class A {
public:
	A_VirtualFunctions *vfPtr;
	int x;
	int A_Calcul() { return 0; }

A_VirtualFunctions Global_A_vfPtr;
Global_A_vfPtr.Calcul = &A::A_Calcul;
```

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
	int x;
	virtual int Calcul() {return 0;}
	A() { x = 0; }
};
void main()
{
	A a;
	a.x = 1;
	a.y = 2;
	A* a2 = &a;
	a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {
	int (*Calcul) ();
};
class A {
public:
	A_VirtualFunctions *vfPtr;
	int x;
	int A_Calcul() { return 0; }
	A() {

		x = 0;
	}
};
A_VirtualFunctions Global_A_vfPtr;
Global_A_vfPtr.Calcul = &A::A_Calcul;
```

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
        int x;
        virtual int Calcul() {return 0;}
        A() { x = 0; }
};
void main()
{
        A a;
        a.x = 1;
        a.y = 2;
        A* a2 = &a;
        a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {
        int (*Calcul) ();
};
class A {
public:
        A_VirtualFunctions *vfPtr;
        int x;
        int A_Calcul() { return 0; }
        A() {
                vfPtr = &Global_A_vfPtr;
                x = 0;
        }
};
A_VirtualFunctions Global_A_vfPtr;
Global_A_vfPtr.Calcul = &A::A_Calcul;
```

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
	int x;
	virtual int Calcul() {return 0;}
	A() { x = 0; }
};
void main()
{
	A a;
	a.x = 1;
	a.y = 2;
	A* a2 = &a;
	a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {
	int (*Calcul) ();
};
class A {
public:
	A_VirtualFunctions *vfPtr;
	int x;
	int A_Calcul() { return 0; }
	A() {
		vfPtr = &Global_A_vfPtr;
		x = 0;
	}
};
A_VirtualFunctions Global_A_vfPtr;
Global_A_vfPtr.Calcul = &A::A_Calcul;

void main()
{
	A a;
	a.x = 1;
	a.y = 2;
	A* a2 = &a;

}
```

How virtual methods are modeled by C++ compiler

App.cpp

```
class A
{
public:
	int x;
	virtual int Calcul() {return 0;}
	A() { x = 0; }
};
void main()
{
	A a;
	a.x = 1;
	a.y = 2;
	A* a2 = &a;
	a2->Calcul();
}
```

Pseudo C/C++ Code

```
struct A_VirtualFunctions {
	int (*Calcul) ();
};
class A {
public:
	A_VirtualFunctions *vfPtr;
	int x;
	int A_Calcul() { return 0; }
	A() {
		vfPtr = &Global_A_vfPtr;
		x = 0;
	}
};
A_VirtualFunctions Global_A_vfPtr;
Global_A_vfPtr.Calcul = &A::A_Calcul;

void main()
{
	A a;
	a.x = 1;
	a.y = 2;
	A* a2 = &a;
	a2->vfPtr->Calcul();
}
```

How virtual methods are modeled by C++ compiler

- Keep in mind the ***vfptr*** is just a pointer. As such, it can be changed during execution

App.cpp

```
class A
{
public:
	int x;
	virtual void Print() { printf("A"); }
};
class B
{
public:
	int x;
	virtual void Print() { printf(“B"); }
};
void main()
{
	A a;
	B b;
	A* a2 = &a;
	a.Print();
	a2->Print();
}
```

- This code will print “**AA**” on the screen. First time when method **Print** is called directly *(“a.Print()”), second time when method* **Print** is called using the ***vfptr*** pointer *(“a2->Print()”)*

How virtual methods are modeled by C++ compiler

- Keep in mind the ***vfptr*** is just a pointer. As such, it can be changed during execution

App.cpp

```
class A
{
public:
	int x;
	virtual void Print() { printf("A"); }
};
class B
{
public:
	int x;
	virtual void Print() { printf("B"); }
};
void main()
{
	A a;
	B b;
	memcpy(&a, &b, sizeof(void*));
	A* a2 = &a;
	a.Print();
	a2->Print();
}
```

- This code will however print "**AB**". Using **memcpy** function allow us to overwrite the actual **vfptr-ul** of object "***a***" with the one from object "***b***". As method ***Print*** has the same signature in both classes (***A*** and ***B***) the result will be "AB"

How virtual methods are modeled by C++ compiler

- Keep in mind the ***vfptr*** is just a pointer. As such, it can be changed during execution

App.cpp

```
class A
{
public:
	int x;
	virtual void Print() { printf("A"); }
};
class B
{
public:
	int x;
	virtual void Print() { printf("B"); }
};
void main()
{
	A a;
	B b;
	memcpy(&a, &b, sizeof(void*));
	A* a2 = &a;
	A a3 = (*a2);
	A *a4 = &a3;
	a4->Print();
}
```

- Every constructor called will set the ***vfptr*** to its correct value. In this case , "***A a3=(*a2)***" will call the copy constructor for class ***A*** and will set the ***vfptr*** for local variable ***a3*** correctly.
- As a result, this code will print "**A**" on the screen , even if "***a2***" has the ***vfptr*** of "***b***"

How virtual methods are modeled by C++ compiler

- A virtual function can be overwritten in the derived class.

App.cpp

```
class A
{
public:
      int x, y;
      virtual int Suma() { return x + y; }
      virtual int Diferenta() { return x - y; }
      virtual int Produs() { return x*y; }
};
class B : public A
{
public:
      int Suma() { return 1; }
};

void main()
{
      B b;
      b.x = 1;
      b.y = 2;
      A* a;
      a = &b;
      int x = a->Suma();
}
```

- In this case, "**x**" will be 1 as "***a***" is in fact an object of type "***b***" that has overwrite method "***Suma***"
- For the rest of the methods (***Diferenta*** and ***Produs***) the behavior will be identical to the one from the base class (A).

How virtual methods are modeled by C++ compiler

Instance of type A
Address of VFTable A
x
y

Instance of type B
Address of VFTable B
A::x
A::y

VFTable for class A
Address of A::Suma
Address of A::Diferenta
Address of A::Produs
RTTI

VFTable for class B
Address of B::Suma
Address of A::Diferenta
Address of A::Produs
RTTI

How virtual methods are modeled by C++ compiler

- A derived class can also add other (new) virtual methods .

App.cpp

```
class A
{
public:
	int x, y;
	virtual int Suma() { return x + y; }
	virtual int Diferenta() { return x - y; }
	virtual int Produs() { return x*y; }
};
class B : public A
{
public:
	int Suma() { return 1; }
	virtual int Modul() { return 0; }
};

void main()
{
}
```

- In this case, class ***B*** also have a new virtual method called "***Module***") that is not present on class ***A***.
- This means that any class that will be derived from ***B*** will have this method as well.

How virtual methods are modeled by C++ compiler

Instance of type A
Address of VFTable A
x
y

Instance of type B
Address of VFTable B
A::x
A::y

VFTable for class A
Address of A::Suma
Address of A::Diferenta
Address of A::Produs
RTTI

VFTable for class B
Address of B::Suma
Address of A::Diferenta
Address of A::Produs
Address of B::Modul
RTTI

How virtual methods are modeled by C++ compiler

- When a class is derived from two(or more) classes that have virtual functions, the compiler creates multiple ***vfptr*** pointers (one for each base class).

App.cpp

```
class A {
public:
	int a1;
	virtual int Suma() { return 1; }
	virtual int Diferenta() { return 2; }
};
class B {
public:
	int b1,b2;
	virtual int Inmultire() { return 3; }
	virtual int Impartire() { return 4; }
};
class C : public A, public B {
public:
	int x, y;
};
void main() {
	C c;
	C *cptr = &c;
	cptr->Impartire();
	cptr->Diferenta();
}
```

Disasm

```
    cptr->Impartire();
mov        ecx,dword ptr [cptr]
add        ecx,8 //this for type B
mov        eax,dword ptr [cptr]
mov        edx,dword ptr [eax+8]
mov        eax,dword ptr [edx+4]
call       eax
    cptr->Diferenta();
mov        eax,dword ptr [cptr]
mov        edx,dword ptr [eax]
mov        ecx,dword ptr [cptr]
mov        eax,dword ptr [edx+4]
call       eax
```

How virtual methods are modeled by C++ compiler

- When a class is derived from two(or more) classes that have virtual functions, the compiler creates multiple ***vfptr*** pointers (one for each base class).

App.cpp

```
class A {
public:
        int a1;
        virtual int Suma() { return 1; }
        virtual int Diferenta() { return 2; }
};
class B {
public:
        int b1,b2;
        virtual int Inmultire() { return 3; }
        virtual int Impartire() { return 4; }
};
class C : public A, public B {
public:
        int x, y;
};
void main() {
        C c;
        C *cptr = &c;
        cptr->Impartire();
        cptr->Diferenta();
}
```

Offset	Field
+ 0	A::vfptr
+ 4	A::a1
+ 8	B::vfptr
+ 12	B::b1
+ 16	B::b2
+ 20	C::x
+ 24	C::y

VFTable for class A
Address of A::Suma
Address of A::Diferenta
RTTI

VFTable for class B
Address of B::Inmultire
Address of B::Impartire
RTTI

How virtual methods are modeled by C++ compiler

- The same memory alignment is used for classes derived out of class ***C*** (e.g. in this example, class ***D***)

App.cpp

```
class A {
public:
	int a1;
	virtual int Suma() { return 1; }
	virtual int Diferenta() { return 2; }
};
class B {
public:
	int b1,b2;
	virtual int Inmultire() { return 3; }
	virtual int Impartire() { return 4; }
};
class C : public A, public B {
public:
	int x, y;
};
class D : public C {
public:
	int d1;
};
```

Offset	Field
+ 0	A::vfptr
+ 4	A::a1
+ 8	B::vfptr
+ 12	B::b1
+ 16	B::b2
+ 20	C::x
+ 24	C::y
+ 28	D::d1

VFTable for class A
Address of A::Suma
Address of A::Diferenta
RTTI

VFTable for class B
Address of B::Inmultire
Address of B::Impartire
RTTI

- Covariance

Covariance

- **Let's analyze the following code:**

App.cpp

```
class A
{
public:
	int a1, a2;
	virtual A* clone() { return new A(); }
};
class B : public A
{
public:
	int b1, b2;
	virtual A* clone() { return new B(); }
};
void main()
{
	B *b = new B();
	B *ptrB;
	ptrB = b->clone();
}
```

```
error C2440: '=': cannot convert from 'A *' to 'B *'
note: Cast from base to derived requires dynamic_cast
or static_cas
```

- This code will not compile. However, in reality "*b->clone()*" returns an object of *type **B*** so it should work.

Covariance

- **Let’s analyze the following code:**

App.cpp

```
class A
{
public:
      int a1, a2;
      virtual A* clone() { return new A(); }
};
class B : public A
{
public:
      int b1, b2;
      virtual A* clone() { return new B(); }
};
void main()
{
      B *b = new B();
      B *ptrB;
      ptrB = b->clone();
}
```

- We have two solutions for this problem:

Covariance

- Let's analyze the following code:

App.cpp

```
class A
{
public:
	int a1, a2;
	virtual A* clone() { return new A(); }
};
class B : public A
{
public:
	int b1, b2;
	virtual A* clone() { return new B(); }
};
void main()
{
	B *b = new B();
	B *ptrB;
	ptrB = (B*) b->clone();
}
```

- We have two solutions for this problem:

1. Use an explicit cast and convert the pointer from ***A**** to ***B****

Covariance

- Let's analyze the following code:

App.cpp

```
class A
{
public:
	int a1, a2;
	virtual A* clone() { return new A(); }
};
class B : public A
{
public:
	int b1, b2;
	virtual B* clone() { return new B(); }
};
void main()
{
	B *b = new B();
	B *ptrB;
	ptrB = b->clone();
}
```

- We have two solutions for this problem:

1. Use an explicit cast and convert the pointer from ***A**** to ***B****
2. Use covariance. This means that we can modify the return type of the method ***clone*** in class ***B*** to return a ***B* pointer*** instead of an ***A* pointer***.

Covariance

- Let’s analyze the following code:

App.cpp

```
class A
{
public:
	int a1, a2;
	virtual A* clone() { return new A(); }
};
class B : public A
{
public:
	int b1, b2;
	virtual B* clone() { return new B(); }
};
void main()
{
	B *b = new B();
	B *ptrB;
	ptrB = b->clone();
	A *a = (A*)b;
	ptrB = (B*)a->clone();
}
```

- We have two solutions for this problem:

1. Use an explicit cast and convert the pointer from ***A**** to ***B****
2. Use **covariance**. This means that we can modify the return type of the method ***clone*** in class ***B*** to return a ***B* pointer*** instead of an ***A* pointer***.

Covariance is related to the pointer type. In this case, even if the compiler calls “***B::clone***”, the expected value is ***A**** (specific to a ***A**** pointer that is “***a***” → “***A::clone***”)

Covariance

- **Let's analyze the following code:**

App.cpp

```
class A
{
public:
	int a1, a2;
	virtual A* clone() { return new A(); }
};
class B : public A
{
public:
	int b1, b2;
	virtual B* clone() { return new B(); }
};
void main()
{
	B *b = new B();
	B *ptrB;
	ptrB = b->clone();
	A *a = (A*)b;
	ptrB = a->clone();
}
```

- We have two solutions for this problem:

1. Use an explicit cast and convert the pointer from ***A**** to ***B****
2. Use **covariance**. This means that we can modify the return type of the method ***clone*** in class ***B*** to return a ***B* pointer*** instead of an ***A* pointer***.

That is why this code will **NOT** compile, as the result for ***a->clone*** is ***A**** and not ***B****. During execution, "***B::clone***" will be call, nevertheless.

Covariance

- Let's analyze the following code:

App.cpp

```
class A
{
public:
	int a1, a2;
	virtual A* clone() { return new A(); }
};
class B : public A
{
public:
	int b1, b2;
	virtual int* clone() { return new int(); }
};
void main()
{
	…
}
```

error C2555: 'B::clone': overriding virtual function return type differs and is not covariant from 'A::clone'

- This code will not compile. The return type for virtual functions can be changed, but only to a type that is derived from the return type of the virtual method described in the base class. In this case, **int*** is not derived from **A***

Abstract classes (Interfaces)

Abstract classes (Interfaces)

- In C++ we can define a virtual method without a body (it is called a **pure virtual method** and it is defined by adding **"=*0*"** at the end of its definition).
- If a class contains a pure virtual method, that class is an abstract class (a class that can not be instantiated). In other languages this concept is <u>similar</u> to the concept of an interface.
- Having a pure virtual method forces the one that implements a derived class to implement that method as well if he/she would like to create an instance from the newly created class.

App.cpp

```
class A
{
public:
	int a1, a2, a3;
	virtual void Set() = 0;
};
void main()
{
	A a;
}
```

```
error C2259: 'A': cannot instantiate abstract class
note: due to following members:
note: 'void A::Set(void)': is abstract
note: see declaration of 'A::Set'
```

- The code will not compile as "A" is an abstract class.

Abstract classes (Interfaces)

- In C++ we can define a virtual method without a body (it is called a **pure virtual method** and it is defined by adding "=0" at the end of its definition).

App.cpp

```
class A
{
public:
	int a1, a2, a3;
	virtual void Set() = 0;
};
class B: A
{
public:
	int a1, a2, a3;
	void Set(){… };
}
void main()
{
	B b;
}
```

- This code will compile because class ***B*** has an implementation for method ***Set***
- In order to be able to create an instance of a class, all of its pure virtual methods (defined in that class or obtained via inheritance) **MUST** be implemented !

Abstract classes (Interfaces)

- In C++ we can define a virtual method without a body (it is called a **pure virtual method** and it is defined by adding “=0” at the end of its definition).

App.cpp

```
class A
{
public:
	int a1, a2, a3;
	virtual void Set() = 0;
};
class B: A
{
public:
	int a1, a2, a3;
	void Set(){… };
}
void main()
{
	B b;
	A* a;
}
```

- This code will however compile. It is possible (and recommended whenever working with polymorphism) to create a pointer towards an abstract class (in this case an ***A**** pointer).

Abstract classes (Interfaces)

- Other languages (such as Java or C#) have a similar concept called ***interface*** (primarily used in these languages to avoid multiple inheritance).
- ***interfaces*** are however different from an abstract class. An interface ***CAN NOT*** have data members, or methods that are not pure virtual. An abstract class is a class that has <u>at least one</u> pure virtual method. An abstract class can have methods, constructors, destructor or data members.
- In C++ it is often easier to use ***struct*** instead of class to describe in interface due to the fact that the default access modifier is ***public***
- Cl.exe (Microsoft) has a keyword (***__interface***) that works like an interface (allows you to create on). However, this is not part of the standard.

- Memory alignment in case of inheritance

Memory alignment in case of inheritance

```
class A
{
public:
   int a1,a2,a3;
};
```

```
sizeof(A) = 12
```

```
class B: public A
{
public:
   int b1,b2
};
```

```
sizeof(B) = 20
```

<table>
<tr><th>Offset</th><th>Field</th><th>C1</th><th>C2</th></tr>
<tr><td>+ 0</td><td>A::a1</td><td rowspan="3">A</td><td rowspan="5">B</td></tr>
<tr><td>+ 4</td><td>A::a2</td></tr>
<tr><td>+ 8</td><td>A::a3</td></tr>
<tr><td>+ 12</td><td>B::b1</td><td></td></tr>
<tr><td>+ 16</td><td>B::b2</td><td></td></tr>
</table>

Memory alignment in case of inheritance

```
class A
{
public:
   int a1,a2,a3;
};
```

```
sizeof(A) = 12
```

```
class B: public A
{
public:
   int b1,b2
};
```

```
sizeof(B) = 20
```

<table>
<tr><th>Offset</th><th>Field</th><th>C1</th><th>C2</th></tr>
<tr><td>+ 0</td><td>A::a1</td><td rowspan="3">A</td><td rowspan="5">B</td></tr>
<tr><td>+ 4</td><td>A::a2</td></tr>
<tr><td>+ 8</td><td>A::a3</td></tr>
<tr><td>+ 12</td><td>B::b1</td><td></td></tr>
<tr><td>+ 16</td><td>B::b2</td><td></td></tr>
</table>

Memory alignment in case of inheritance

```
class A
{
public:
   int a1,a2,a3;
};
```

```
sizeof(A) = 12
```

```
class B: public A
{
public:
   int b1,b2
};
```

```
sizeof(B) = 20
```

Offset	Field	C1	C2
+ 0	A::a1	A	B
+ 4	A::a2		
+ 8	A::a3		
+ 12	B::b1		
+ 16	B::b2		

Memory alignment in case of inheritance

```
class A
{
public:
   int a1,a2,a3;
};
```

```
sizeof(A) = 12
```

```
class B:
{
public:
   int b1,b2;
};
```

```
sizeof(B) = 8
```

```
class C:public A,B
{
public:
   int c1,c2;
};
```

```
sizeof(C) = 28
```

Offset	Field	C1	C2	C3
+ 0	A::a1			
+ 4	A::a2	A		
+ 8	A::a3		B	
+ 12	B::b1			C
+ 16	B::b2			
+20	C::c1			
+24	C::c2			

Memory alignment in case of inheritance

```
class A
{
public:
   int a1,a2,a3;
};
```

sizeof(A) = **12**

```
class B:
{
public:
   int b1,b2;
};
```

sizeof(B) = **8**

```
class C:public B,A
{
public:
   int c1,c2;
};
```

sizeof(C) = **28**

When building a derived class in memory, if the inheritance is does not contain the virtual specifier, will be done using the left-to-right rule for any base classes.

Offset	Field	C1	C2	C3
+ 0	B::b1	B		
+ 4	B::b2			
+ 8	A::a1		A	
+ 12	A::a2			C
+ 16	A::a3			
+20	C::c1			
+24	C::c2			

Memory alignment in case of inheritance

warning C4584: 'C' : base-class 'A' is already a base-class of 'B'.

- Multiple inheritance can create ambiguous situations. For example, in this case the fields from class ***A*** are copied twice in class ***C***.

App.cpp

```
class A
{
public:
	int a1, a2, a3;
};
class B: public A
{
public:
	int b1, b2;
};
class C : public A, public B
{
public:
	int c1, c2;
};
void main()
{
}
```

Offset	Field	C1	C2	C3
+0	A::a1			
+4	A::a2	A		
+8	A::a3			
+12	B::A::a1			
+16	B::A::a2	B::A		C
+20	B::A::a3		B	
+24	B::b1			
+28	B::b2			
+32	C::c1			
+36	C::c2			

Memory alignment in case of inheritance

- Multiple inheritance can create ambiguous situations. For example, in this case the fields from class ***A*** are copied twice in class ***C***.

App.cpp

```
class A {
public:
	int a1, a2, a3;
};
class B: public A {
public:
	int b1, b2;
};
class C : public A, public B {
public:
	int c1, c2;
};
void main()
{
	C c;
	c.a1 = 10;
}
```

This is an ambiguous case. "***c.a1 = 10***" can refer to the member "***a1***" from the direct inheritance of class **A**, or the member "***a1***" from the direct inheritance of class **B** that in terms inherits class **A**.

This code will NOT compile !!!

```
warning C4584: 'C': base-class 'A' is already a base-class of 'B'
note: see declaration of 'A'
note: see declaration of 'B'
---------------------------------------------------------------
error C2385: ambiguous access of 'a1'
note: could be the 'a1' in base 'A'
note: or could be the 'a1' in base 'A
```

Memory alignment in case of inheritance

- Multiple inheritance can create ambiguous situations. For example, in this case the fields from class ***A*** are copied twice in class ***C***.

App.cpp

```
class A {
public:
	int a1, a2, a3;
};
class B: public A {
public:
	int b1, b2;
};
class C : public A, public B {
public:
	int c1, c2;
};
void main()
{
	C c;
	c.A::a1 = 10;
	c.B::A::a1 = 20;
}
```

- The solution is to describe any field/data member using its full scope. For example:
  - "***c.A::a1***" means data member "a1" from the direct inheritance of "A" in class "C"
  - "***c.B::A::a1***" means data member "a1" from the inheritance of "A" in class "B" that is directly inherit by class "C"

- What can we do if we want to have only one copy of the fields from class "A" in our object ?
- This problem is also known as the "***Diamond Problem***"

Memory alignment in case of inheritance

- Multiple inheritance can create ambiguous situations. For example, in this case the fields from class *A* are copied twice in class *C*.

App.cpp

```
class A {
public:
	int a1, a2, a3;
};
class B: public virtual A {
public:
	int b1, b2;
};
class C : public virtual A, public B {
public:
	int c1, c2;
};
void main()
{
	C c;
	c.a1 = 10;
	c.a2 = 20;
}
```

- One solution to this problem is to use the ***virtual*** specifier when deriving from a class. In this case, class "A" is inherited virtually (meaning that its fields must be added once).
- For this code to work, both "C" and "B" class need to inherit class "A" using ***virtual*** keyword.

Memory alignment in case of inheritance

- Just like in the case of virtual methods, if no constructor is present, one will be created by the compiler. However, this constructor is a little bit different than the others (as it has one parameter of type bool).

App.cpp

```
class A {
public:
      int a1, a2, a3;
};
class B: public virtual A {
public:
      int b1, b2;
};
class C : public virtual A, public B {
public:
      int c1, c2;
};
void main()
{
      C c;
      c.a1 = 10;
      c.b1 = 20;
}
```

Disasm

```
     C c;
push         1        <- TRUE
lea          ecx,[c]
call         C::C
     c.a1 = 10;
mov          eax,dword ptr [c]
mov          ecx,dword ptr [eax+4]
mov          dword ptr [c+ecx],10
     c.b1 = 20;
mov          dword ptr [c+20],20
```

Memory alignment in case of inheritance

- The first parameter, tells the constructor if a special table with indexes needs to be created or not !

App.cpp

```
class A {
public:
	int a1, a2, a3;
};
class B: public virtual A {
public:
	int b1, b2;
};
class C : public virtual A, public B {
public:
	int c1, c2;
};
void main()
{
	C c;
	c.a1 = 10;
	c.b1 = 20;
}
```

Disasm C::C

```
	push	ebp
	mov	ebp,esp
	mov	dword ptr [this],ecx
	cmp	dword ptr [ebp+8],0
	je	DONT_SET_VAR_PTR
	mov	eax,dword ptr [this]
	mov	dword ptr [eax],addr_index
DONT_SET_VAR_PTR:
	push	0
	mov	ecx,dword ptr [this]
	call	B::B
	mov	eax,dword ptr [this]
	mov	esp,ebp
	pop	ebp
	ret	4
```

Memory alignment in case of inheritance

- Once the constructor is called, an object that has virtual inheritance will look as follows:

Offset	Field	C1	C2
+ 0	Ptr Class C Variable Offsets Table		
+ 4	B::b1	B	C
+ 8	B::b2		
+ 12	C::c1		
+ 16	C::c2		
+ 20	A::a1	A	
+ 24	A::a2		
+ 28	A::a3		

Offset	Offset relative to C
+ 0	0
+ 4	Virtual A 20

Memory alignment in case of inheritance

- Accessing a data member / field that benefits from the ***virtual*** inheritance, is done in 3 steps (not in one) in the following way:

App.cpp

```cpp
class A { … }
class B: public virtual A { … }
class C : public virtual A, public B { … }
void main()
{
      C c;
      c.a1 = 10;
}
```

Disasm

```
     c.a1 = 10;
mov            eax,dword ptr [c]
mov            ecx,dword ptr [eax+4]
mov            dword ptr [c+ecx],10
```

Memory alignment in case of inheritance

- In the first step, EAX register gets the pointer to the table where offsets of data member/fields from A class are stored

App.cpp

```
class A { … }
class B: public virtual A { … }
class C : public virtual A, public B { … }
```

Offset	Field	C1	C2
+ 0	Ptr Class C Variable Offsets Table		
+ 4	B::b1	B	C
+ 8	B::b2		
+ 12	C::c1		
+ 16	C::c2		
+ 20	A::a1	A	
+ 24	A::a2		
+ 28	A::a3		

Disasm

```
    c.a1 = 10;
mov          eax,dword ptr [c]
mov          ecx,dword ptr [eax+4]
mov          dword ptr [c+ecx],10
```

Memory alignment in case of inheritance

- Second step - ECX gets the value from the second index in that table (+4), more exactly value 20 (that reflects the offset of "*A*" from the beginning of "*C*")

App.cpp

```
class A { … }
class B: public virtual A { … }
class C : public virtual A, public B { … }
```

Offset	Field	C1	C2
+ 0	Ptr Class C Variable Offssets Table		
+ 4	B::b1	B	C
+ 8	B::b2		
+ 12	C::c1		
+ 16	C::c2		
+ 20	A::a1	A	
+ 24	A::a2		
+ 28	A::a3		

Disasm

```
     c.a1 = 10;
mov           eax,dword ptr [c]
mov           ecx,dword ptr [eax+4]
mov           dword ptr [c+ecx],10
```

Offset	Offset relative to C	
+ 0		0
+ 4	Virtual A	20

Memory alignment in case of inheritance

- Last step, we use "ECX" register as an offset to access A::a1 from the beginning of local variable "c".

App.cpp

```
class A { … }
class B: public virtual A { … }
class C : public virtual A, public B { … }
```

Offset	Field	C1	C2
+ 0	Ptr Class C Variable Offssts Table		
+ 4	B::b1	B	C
+ 8	B::b2		
+ 12	C::c1		
+ 16	C::c2		
+ 20	A::a1	A	
+ 24	A::a2		
+ 28	A::a3		

Disasm

```
    c.a1 = 10;
mov          eax,dword ptr [c]
mov          ecx,dword ptr [eax+4]
mov          dword ptr [c+ecx],10
```

Offset	Offset relative to C	
+ 0		0
+ 4	Virtual A	20

Memory alignment in case of inheritance

- Fields/Data members that are obtained via virtual inheritance are usually added at the end of the class alignment.

App.cpp

```
class A
{ … }
class B: public virtual A
{ … }
class C : public virtual A,
          public B
{ … }
```

Offset	Offset relative to C	
+ 0		0
+ 4	Virtual A	20

Offset	Field
+ 0	ptr class C virtual members offsets
+ 4	C::B::**b1**
+ 8	C::B::**b2**
+ 12	C::**c1**
+ 16	C::**c2**
+ 20	A::**a1** (virtual A from C)
+ 24	A::**a2** (virtual A from C)
+ 28	A::**a3** (virtual A from C)

Memory alignment in case of inheritance

- If we use virtual inheritance when deriving "C" from "B" (in addition to the usage of virtual inheritance for class "A") we will obtain the following alignment:

App.cpp

```
class A
{ … }
class B: public virtual A
{ … }
class C : public virtual A,
          public virtual B
{ … }
```

Offset	Field
+ 0	ptr class C virtual members offsets
+ 4	C::**c1**
+ 8	C::**c2**
+ 12	A::**a1** (virtual A from C)
+ 16	A::**a2** (virtual A from C)
+ 20	A::**a3** (virtual A from C)
+ 24	ptr class B virtual members offsets
+ 28	B::b1 (virtual B from C)
+ 32	B::b2 (virtual B from C)

Offset	Offset relative to C	
+ 0		0
+ 4	Virtual A	12
+ 8	Virtual B	24

Memory alignment in case of inheritance

- In case of the index table for class "B", the offset "-12" refers to the position of "A" class (also obtain via virtual inheritance) relative to B with respect to C class (24 (offset of B) - 12 = 12 (offset of A))

App.cpp

```
class A
{ … }
class B: public virtual A
{ … }
class C : public virtual A,
          public virtual B
{ … }
```

Offset	Field
+ 0	ptr class C virtual members offsets
+ 4	C::**c1**
+ 8	C::**c2**
+ 12	A::**a1** (virtual A from C)
+ 16	A::**a2** (virtual A from C)
+ 20	A::**a3** (virtual A from C)
+ 24	ptr class B virtual members offsets
+ 28	B::b1 (virtual B from C)
+ 32	B::b2 (virtual B from C)

Offset	Offset relative la B	
+ 0		0
+ 4	Virtual A	-12

Memory alignment in case of inheritance

- If we make only the inheritance of B from C to be virtual, the memory alignment is as follows:

App.cpp

```
class A
{ … }
class B: public A
{ … }
class C : public A,
          public virtual B
{ … }
```

Offset	Offset relative la C	
+ 0		-12
+ 4	Virtual B	12

Offset	Field
+ 0	A::**a1**
+ 4	A::**a1**
+ 8	A::**a3**
+ 12	ptr class C virtual members offsets
+ 16	C::**c1**
+ 20	C::**c2**
+ 24	B::A::**a1**
+ 28	B::A::**a2**
+ 32	B::A::**a3**
+ 36	B::**b1**
+ 40	B::**b2**

Memory alignment in case of inheritance

- If we make only the inheritance of B from C to be virtual, the memory alignment is as follows:

App.cpp

```
class A
{ … }
class B: public A
{ … }
class C : public A,
          public virtual B
{ … }
```

Offset	Field
+ 0	A::**a1**
+ 4	A::**a1**
+ 8	A::**a3**
+ 12	ptr class C virtual members offsets
+ 16	C::**c1**
+ 32	B::A::**a3**
+ 36	B::**b1**
+ 40	B::**b2**

Offset	Offset relative la C	
+ 0		-12
+ 4	Virtual B	12

First index (+0 offset, value -12) represents the offset of object C relative to the table of indexes). It is usually 0 (as this table is the first entry), however in this case it is a negative value.

Memory alignment in case of inheritance

- If we make only the inheritance of B from C to be virtual, the memory alignment is as follows:

App.cpp

```
class A
{ … }
class B: public A
{ … }
class C : public A,
          public virtual B
{ … }
```

Offset	Field
+ 0	A::**a1**
+ 4	A::**a1**
+ 8	A::**a3**
+ 12	ptr class C virtual members offsets
+ 16	C::**c1**
+ 20	C::**c2**
+ 36	B::**b1**
+ 40	B::**b2**

Offset	Offset relative la C	
+ 0		-12
+ 4	Virtual B	12

The second offset (+4, value +12) reflects the position of B relative to the offset of the index table (12+12=24).

Q & A