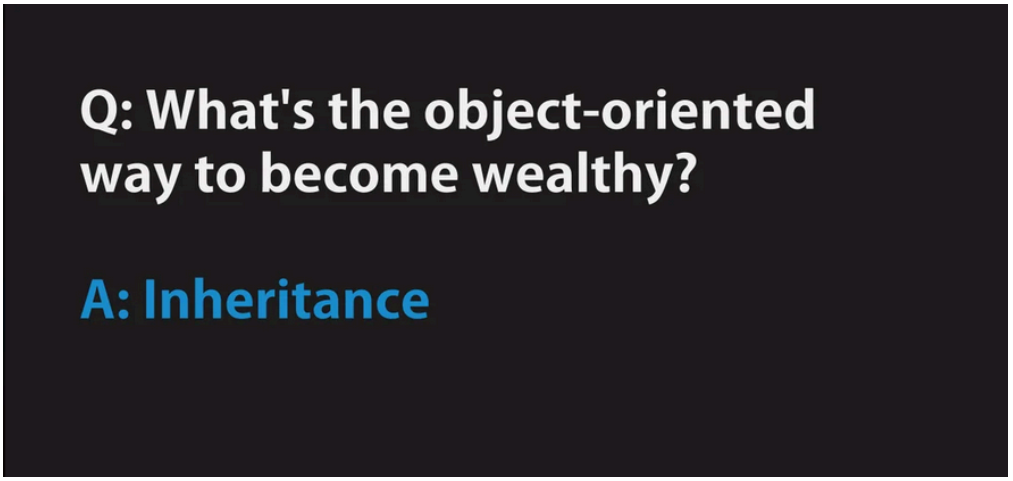
Seminar 3



Sintaxa:

```
class Clasa_Derivata : [modificatori de acces] Clasa_de_Baza1, [modificatori de acces] Clasa_de_Baza2, [modificatori de acces] Clasa_de_Baza3 ........
```

Modificatori de acces

Modificator de moștenire	Membrii `public` ai clasei de bază devin in clasa derivată	Membrii `protected` ai clasei de bază devin	Membrii `private` ai clasei de bază devin
public	public	protected	inaccesibil
protected	protected	protected	inaccesibil
private	private	private	inaccesibil

Dacă nu specificăm tipul de moștenire, avem by default moștenire *private*.
Moștenirea public e cea mai des întâlnită și păstrează vizibilitatea după cum e definită în clasa de bază (private rămâne private și în derivată, protected rămâne protected, iar private e inaccesibil)
Moștenirea protected transformă datele public în protected, datele protected rămân protected, iar private nu sunt accesibile.
Moștenirea private (cea default) există mai mult pentru consistență, căci toate

lucrurile din clasa de bază devin private în clasa moștenită, deci nu avem acces la ele.
Ca o oarecare intuiție ca să fie ușor de reținut, fiecare tip de moștenire face ca în clasa derivată, lucrurile din clasa de bază să fie cel mult de tipul moștenirii.

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

// Clasa de baza
class Baza {
 public: int a;
 protected: int b;
 private: int c;           // private (nu se mosteneste)
};

// Mostenire publica
class DerivataPublic : public Baza {
public:
 void afisare() {
   a = 1;      // OK - public ramane public
   b = 2;      // OK - protected ramane protected
   // c = 3;   // EROARE - c este privat => NU SE MOSTENESTE
 }
};

// Mostenire protected
class DerivataProtected : protected Baza {
public:
 void afisare() {
   a = 4;      // OK - public devine protected
   b = 5;      // OK - protected ramane protected
   // c = 6;   // EROARE - c nu este accesibil
 }
};

// Mostenire privata
class DerivataPrivata : Baza {
public:
 void afisare() {
   a = 7;      // OK - public devine privat
   b = 8;      // OK - protected devine privat
```

```
   // c = 9;   // EROARE - c nu este accesibil
 }
};

int main() {
 DerivataPublic obj1;
 obj1.a = 10;     // OK - a este public in DerivataPublic
 // obj1.b = 20;  // EROARE - b este protected

 DerivataProtected obj2;
 // obj2.a = 30;  // EROARE - a este protected in
DerivataProtected

 DerivataPrivata obj3;
 // obj3.a = 40;  // EROARE - a este privat in DerivataPrivata

 return 0;
}
```

Tipuri de moștenire:

- Moștenire simplă: o clasă e derivată dintr-o singură clasă de bază;
- Moștenire multiplă: o clasă e derivată din mai multe clase de bază;
- Moștenire în lanț: o clasă derivată devine bază pentru o altă clasă derivată;
- Moștenire ierarhică: mai multe clase sunt derivate din aceeași clasă de bază;
- Moștenire în diamant: o clasă e derivată din două clase, derivate la rândul lor din aceași clasă de bază.

Polimorfism:

Polimorfismul în C++ vine în 2 forme:

- Polimorfism la complilare/polimorfism static (compile time)
- Polimorfism la executie/polimorfism dinamic (run time)

La compile-time se fac verificări de sintaxă și de semantică.
La run-time se alocă memoria necesară variabilelor (erori depistate: împărțire la 0, dereferențiere de pointer null, am rămas fără memorie)

Polimorfismul la compilare se realizează prin supraîncărcarea de funcții și de operatori (și template-uri, dar mai încolo):

1. Supraîncărcarea funcțiilor:
   Atunci când se apelează o funcție, compilatorul caută, în ordine:
   a. O potrivire exactă a paremetrilor:
   b. Dacă nu se găsește potrivire exactă, se trece la promovarea prin internal type conversions (pot fi promovate automat char la int, float la double, enum la int)

```
#include <iostream>
using namespace std;
void f(char* x) {cout<<x<<" from char*\n";}
void f(int x) { cout<<x<<" from int\n";}
int main()
{
   f('a'); // char-ul e promovat la int
   return 0;
}
```

   c. Se încercă apoi standard conversion (orice tip numeric cu orice tip numeric, enum cu orice tip numeric, 0 cu pointeri și tipuri numerice, un pointer cu void*)

```
#include <iostream>
using namespace std;
struct employee {int x, y, z;};
void f(float x) {cout<<x<<" from float\n";}
void f(employee x) {cout<<x.x<<" from employee\n";}
int main()
{
   f('a'); // 'a' se converteste la int, apoi la
float
   return 0;
}
```

   d. Se încearcă potrivirea prin conversie definită de utilizator

```
#include <iostream>
using namespace std;
class A
{
public:
   operator int()// operator de conversie definit de
```

```
noi
   {
       return 9;
   }
};
void f(float x) {cout<<x<<" from float\n";}
void f(int x) {cout<<x<<" from int\n";}
int main()
{
   A a;
   f(a); // a se converteste la int prin operatorul
int()
   return 0;
}
```

   e. Dacă nu se găsește niciuna din variante sau avem mai multe potriviri pe acelasi nivel => eroare

2. Supraîncărcarea operatorilor
   Câteva reguli de bază:
   - Dacă aveți de supraîncărcat operatorii =, [], () sau ->, supraîncărcarea se face ca funcție membră a clasei.
   - Un operator unar se supraîncarcă cu o funcție membră.
   - Un operator binar:
     - Dacă nu modifică termenul din stânga (ex: +) se supraîncarcă cu funcție normală sau friend
     - Dacă modifcă termenul din stânga, dar nu se poate modifica definiția acelui termen (ex: operatorul <<) supraîncărcare ca funcție normală sau friend.
     - Dacă se modifică termenul din stânga (ex: +=) și se poate modifica definiția termenului din stânga, supraîncărcare ca funcție membră a clasei.

   Se pot supraîncărca următorii operatori:

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[ ]	()
->	->*	new	new [ ]	delete	delete [ ]

+ Operatorul de typecast (care ne permite să convertim tipul nostru la un alt tip de date)

```
#include <iostream>
#include <string>
using namespace std;
class Portofel
{
public:
   Portofel(int nr_monede=0) : _nr_monede(nr_monede) {};
   //operator de typecast la int
   operator int() const { return _nr_monede; }
   //operator de typecast la string
   operator string() const { return to_string(_nr_monede) + "
monede";}
   int get_nr_monede() const { return _nr_monede; }
   int set_nr_monede(int nr_monede) { _nr_monede=nr_monede; }
private:
   int _nr_monede;
};
int main()
{
   Portofel ob1(9);
   cout<<int(ob1)<<"\n"; // afiseaza 9
   cout<<string(ob1); // afiseaza 9 monede
```

```
    return 0;
}
```

Ce nu putem supraîncărca:

- Operatorul de selecție .
- Operatorul de dereferențiere .*
- Operatorul rezolutie de scop ::
- Operatorul sizeof

Polimorfismul la execuție este capacitatea unui obiect de a apela metode diferite în funcție de tipul său real, chiar dacă este accesat printr-un pointer sau referință la o clasă de bază.

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

class Animal
{
public:
    void sound() {cout<<"undefined \n";}
};
class Cat : public Animal
{
public:
    void sound() {cout<<"meow \n";}
};
class Dog : public Animal
{
public:
    void sound() {cout<<"wof \n";}
};

int main()
{
    Cat cat1;
    cat1.sound(); // se afiseaza meow
    Dog dog1;
    dog1.sound(); // se afiseaza wof
    Cat* catp=new Cat;
    catp->sound(); // se afiseaza meow
    Dog* dogp=new Dog;
```

```
   dogp->sound(); // se afiseaza wof
   // Putem sa facem un animal* pointer catre un obiect de tip cat
sau dog.
   // Putem, pentru ca fiecare cat sau dog are o parte de animal.
   Animal* animal1 = new Cat;
   Animal* animal2 = new Dog;
   animal1->sound(); // se afiaseaza undefined
   animal2->sound(); // se afiaseaza undefined
   return 0;
}
```

De multe ori vom avea astfel de ierarhii cu o clasa de baza mai generala cum este animal. Vom vrea sa tinem vectori de pointeri la astfel de obiecte si sa apelam o anumita functie din fiecare element in parte. Cum facem ca acea functie apelata sa fie cea din cat sau dog si nu animal?
Aici intervine polimorfismul la executie si functiile virtuale. Putem rezolva problema de mai sus prin adaugarea keyword-ului virtual la functia sound din clasa de baza animal:

```
#include <iostream>

using namespace std;

class Animal
{
public:
   virtual void sound() {cout<<"undefined \n";}
};
class Cat : public Animal
{
public:
   void sound() {cout<<"meow \n";}
};
class Dog : public Animal
{
public:
   void sound() {cout<<"wof \n";}
};

int main()
{
```

```
   Cat cat1;
   cat1.sound(); // se afiseaza meow
   Dog dog1;
   dog1.sound(); // se afiseaza wof
   Cat* catp=new Cat;
   catp->sound(); // se afiseaza meow
   Dog* dogp=new Dog;
   dogp->sound(); // se afiseaza wof
   // Putem sa facem un animal* pointer catre un obiect de tip cat
sau dog.
   // Putem, pentru ca fiecare cat sau dog are o parte de animal.
   Animal* animal1 = new Cat;
   Animal* animal2 = new Dog;
   animal1->sound(); // acum se afiaseaza meow
   animal2->sound(); // acum se afiseaza wof
   return 0;
}
```

Virtual și vtable:

Când lucrăm cu funcții virtuale în C++, este esențial să înțelegem mecanismul de bază care le face să funcționeze: **vtable-ul** (tabelul virtual). **Vtable-ul** este o structură de date internă folosită de compilatorul C++ pentru a implementa polimorfismul dinamic (la execuție). Practic, este un tabel de pointeri către funcții.

Fiecare clasă care are funcții virtuale va avea propriul ei vtable, iar fiecare obiect al acelei clase conține un pointer ascuns, numit **vptr** (*virtual pointer*), care pointează către vtable-ul clasei respective. Vtable-ul conține pointeri către funcțiile virtuale definite în acea clasă și în clasele de bază.

Late binding = un mecanism prin care compilatorul adauga cod care identifica tipul de obiect la run time si apoi potriveste functiile la apelul de functie.

Exerciții:

1. Spuneți dacă programul de mai jos este corect. În caz afirmativ, spuneți ce afișează, în caz negativ spuneți de ce nu este corect și realizați o modificare astfel încât acesta să compileze fără a-i schimba funcționalitatea.

```
#include <iostream>
using namespace std;
class B1 { public: int x; };
class B2 { public: int y; };
class B3 { public: int z; };
class B4 { public: int t; };
class D: public B1, private B2, protected B3, B4 { public: int u;
};
int main() {
D d;
cout << d.u;
cout << d.x;
cout << d.y;
cout << d.z;
cout << d.t;
return 0;
}
```

Dar daca in clasele B1, B2, B3, B4 puneam private sau protected?

2. Compileaza sau nu?

```
#include <iostream>
using namespace std;
class B
{
protected:
   int a;
public:
   B() { a = 7; }
};
class D : public B
{ public:
   int b;
   D() { b = a + 7; }
};
int main()
```

```
{
   D d;
   cout << d.b;
   return 0;
}
```

3. Aceeasi cerinta:

```
#include <iostream>
using namespace std;
class Base
{
protected:
int x;
public:
Base(int i) { x = i; }
};
class Derived : public Base
{
public:
Derived(int i) : x(i) {}
void print() { cout << x; }
};
int main()
{
Derived d(10);
d.print();
}
```

4.

```
 #include <iostream>
using namespace std;
class B
{
protected:
   int x;
public:
   B() { x = 78; cout<<"B\n";}
};
class D1 : virtual public B
{
public:
```

```
   D1() { x = 15; cout<<"D1\n";}
};
class D2 : virtual public B
{
public:
   D2() { x = 37; cout<<"D2\n";}
};
class C : public D2, public D1
{
public:
   C() {cout<<"C\n";}
   int get_x() { return x; }
};
int main()
{
   C ob;
   cout << ob.get_x();
   return 0;
}
```

5.

```
#include <iostream>
using namespace std;
class A
{
public:
   int x;
   A(int i = 0) { x = i; }
   virtual A minus() { return (1 - x); }
};
class B : public A
{
public:
   B(int i = 0) { x = i; }
   void afisare() { cout << x; }
};
int main()
{
   A *p1 = new B(18);
   *p1 = p1->minus();
   p1->afisare();
   return 0;
```

```
}
```

6.

```
#include <iostream>
using namespace std;
class Parent {
   int i;
public:
   Parent(int ii) : i(ii) {
       cout << "Parent(int ii)\n";
   }
   Parent(const Parent& b) : i(b.i) {
       cout << "Parent(const Parent&)\n";
   }
   Parent() : i(0) { cout << "Parent()\n"; }
   friend ostream&
   operator<<(ostream& os, const Parent& b) {
       return os << "Parent: " << b.i << endl;
   }
};
class Member {
   int i;
public:
   Member(int ii) : i(ii) {
       cout << "Member(int ii)\n";
   }
   Member(const Member& m) : i(m.i) {
       cout << "Member(const Member&)\n";
   }
   friend ostream&
   operator<<(ostream& os, const Member& m) {
       return os << "Member: " << m.i << endl;
   }
};
class Child : public Parent {
   int i;
   Member m;
public:
   Child(int ii) : Parent(ii), i(ii), m(ii) {
       cout << "Child(int ii)\n";
   }
   // Child(const Child& c) : i(c.i), m(c.m) {}
```

```
   friend ostream&
   operator<<(ostream& os, const Child& c){
       return os << (Parent&)c << c.m
       << "Child: " << c.i << endl;
   }
};
int main() {
   Child c(2);
   cout << "calling copy-constructor: " << endl;
   Child c2 = c; // Calls copy-constructor
   cout << "values in c2:\n" << c2;
} ///:~
```

—-

```
   //Child(const Child& c)
: Parent(c), i(c.i), m(c.m) {
cout << "Child(Child&)\n";
}
```

7.

```
#include <iostream>
using namespace std;
class Base {
public:
   virtual static void greet() {
       std::cout << "Hello from Base" << std::endl;
   }
};

class Derived : public Base {
public:
   static void greet() {
       std::cout << "Hello from Derived" << std::endl;
   }
};
int main()
{
   Derived::greet();
```

```
    Base::greet();
    Base* ptr = new Derived();
    ptr->greet();
}
```