Facultatea de Matematică și Informatică
Universitatea din București

Programare orientată pe obiecte

- suport de curs -

Anca Dobrovăț
Andrei Păun

An universitar 2024 – 2025
Semestrul II
Seriile 13, 14 si 15

Curs 7 & 8

Agenda cursului

1. Moșteniri și funcții virtuale în C++

- Funcții virtuale în C++
- Destructori și virtualizare
- Constructori si virtualizare
- Copy & swap, RAII
- Interfețe non-virtuale

2. Downcasting

3. Moștenire multiplă în C++

4. Tratarea excepțiilor

Multumiri Asist drd Marius Micluta – Campeanu pentru co-realizarea (intr-o mare masura) a acestui material!

Redefinirea funcțiilor membre

Clasa derivată are acces la toți membrii cu acces **protected** sau **public** ai clasei de bază.

Este permisă supradefinirea funcţiilor membre clasei de bază cu funcții membre ale clasei derivate.

-2 modalități de a redefini o funcție membră:

- **cu același antet ca în clasa de bază** (“redefining” - în cazul funcţiilor oarecare / “overriding” - în cazul funcțiilor virtuale);
- **cu schimbarea listei de argumente sau a tipului returnat**.

Redefinirea funcțiilor membre

Obs:

Schimbarea interfeței clasei de bază prin modificarea tipului returnat sau a signaturii unei funcții, înseamnă, de fapt, utilizarea clasei în alt mod.

Scopul principal al moștenirii: polimorfismul.

Schimbarea signaturii sau a tipului returnat = schimbarea interfeței = contravine exact polimorfismului (un aspect esențial este păstrarea interfeței clasei de bază).

Mod de prevenire la funcțiile virtuale: cuvântul cheie **override** (C++11)

Funcții virtuale

Codul poate "creşte" fără schimbări semnificative: programe **ușor** de extins

Exemplu de separare dintre interfață și implementare

- Clasele care folosesc interfața definită în clasa de bază **nu se modifică** atunci când schimbăm implementarea sau când adăugăm o nouă derivată

Pointer de tip bază care arată către obiect de tip derivat şi cheamă o funcţie virtuală din bază execută funcția redefinită în cea mai specifică derivată - **late binding**

Upcasting - Tipul derivat poate lua locul tipului de bază (L-ul din SOLID – va urma)

Funcții virtuale pure: forțează derivatele să definească o implementare

Tipul de retur al unei funcții virtuale nu poate fi schimbat în derivate.
Excepție: tipuri covariante.

Tipuri covariante și în alte limbaje: C#, Dart, Java, Python, Scala, TypeScript

1. Moștenire, funcții virtuale

```
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {  public:
  virtual void play(note) const = 0;
};
void Instrument::play(note) const {  std::cout << "Instrument::play" << std::endl;  }

class Wind : public Instrument {
public:  void play(note) const override {  std::cout << "Wind::play" << std::endl; }};

class String : public Instrument {
public:  void play(note) const override {  std::cout << "String::play" << std::endl; }};

void tune(Instrument& i) {  i.play(middleC);  }

int main() {
  Wind flute;  String cello;
  tune(flute);  tune(cello);
}
```

Constructori și virtualizare

Obs. **NU putem avea constructori virtuali** (în sensul "virtual Instrument() {}")

NU apelăm funcții virtuale în constructori sau destructori (detalii)

- Se va apela definiția din clasa curentă, nu dintr-o clasă mai derivată
- Deși de obicei nu este o problemă în sine, poate cauza confuzie în proiecte mari
- În limbaje dinamice (e.g. Python) nu este o problemă, dar acolo nu avem verificări de tipuri prea stricte

Soluții:

- Constructori “virtuali” in sensul de (**funcții de clonare**)
- Clase și funcții de tip factory

Dacă avem nevoie de funcții virtuale în destructori, ar trebui să ne întrebăm de ce am avea nevoie de așa ceva

- "soluție": apel explicit: Base::f() sau Derived::f()

Destructori și virtualizare

Se cheamă în ordine inversă decât constructorii.

În general două situații:

- Destructor public și virtual
  - Dacă avem și alte funcții virtuale, deci folosim pointeri către bază
- Destructor protected și non-virtual
  - Dacă dorim să folosim doar obiecte derivate

Dacă vrem să eliminăm porțiuni alocate dinamic și pentru clasa derivată dar facem upcasting trebuie să folosim destructori virtuali.

Destructori și virtualizare

```
class Base1 {public:  ~Base1() { cout << "~Base1()\n"; } };

class Derived1 : public Base1 {public:  ~Derived1() { cout << "~Derived1()\n"; } };

class Base2 {public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};

class Derived2 : public Base2 {public:  ~Derived2() { cout << "~Derived2()\n"; } };

int main() {
  Base1* bp = new Derived1;
  delete bp; // Afis: ~Base1()
  Base2* b2p = new Derived2;
  delete b2p; // Afis: ~Derived2()  ~Base2()
}
```

Destructori virtuali puri

Utilizare: poate fi utilizat dacă avem funcții virtuale pe care nu vrem să le suprascriem în toate derivatele.
Restricție: trebuie să aibă o definiție (chiar dacă este abstractă).
La moștenire nu mai trebuie redefiniți (se construiește un destructor din oficiu)
De ce? Pentru a preveni instanțierea clasei.
Obs. Nu are nici un efect dacă nu se face upcasting, dar atunci am folosi destructor protected și non-virtual.

```
class AbstractBase {
public:
  virtual ~AbstractBase() = 0;
};
AbstractBase::~AbstractBase() {}

class Derived : public AbstractBase {};
// No overriding of destructor necessary?
int main() { Derived d; }
```

Destructori protected și non-virtuali

Utilizare: dorim să ținem într-o clasă de bază comună atribute și funcții, dar construim doar obiecte derivate și nu avem nevoie de pointeri sau referințe de bază.

De ce nu public?

Pentru a preveni instanțierea clasei din exterior (nu avem object slicing).

De ce protected și nu private?

Pentru a putea fi apelat de clasele derivate.

```
class Base {
protected:
  ~Base() = default;
};
class Derived1 : public Base {};
class Derived2 : public Base {};

int main() { Derived1 d1; Derived2 d2; /* Base b; */ /*error */ }
```

Constructori și virtualizare

Situație: într-o clasă reținem ca atribut un pointer de tip bază al altei clase pentru a putea apela funcții virtuale prin acel pointer.

```
class Instrument {
public:
   virtual ~Instrument() = 0;
   virtual void play(int) const {} };
Instrument::~Instrument() = default;
class Wind : public Instrument { /* implementare play */ };
class String : public Instrument { /* implementare play */ };
class Orchestra {
   std::vector<Instrument*> instruments;
public:
   void add(Instrument* inst) { instruments.push_back(inst); }
   void rehearse() { for(const auto& inst : instruments)  inst->play(0); }
};
int main() { Orchestra o1; o1.add(new Wind); o1.add(new String); o1.rehearse(); }
```

Constructori și virtualizare

Ce probleme pot apărea?

Avem memory leaks (am folosit DrMemory).

```
~~Dr.M~~ Error #3: LEAK 8 direct bytes 0x0000018f7e600840-0x0000018f7e600848 + 0 indirect bytes
~~Dr.M~~ # 0 replace_operator_new              [D:\a\drmemory\drmemory\common\alloc_replace.c:2903]
~~Dr.M~~ # 1 main
~~Dr.M~~
~~Dr.M~~ Error #4: LEAK 8 direct bytes 0x0000018f7e6008a0-0x0000018f7e6008a8 + 0 indirect bytes
~~Dr.M~~ # 0 replace_operator_new              [D:\a\drmemory\drmemory\common\alloc_replace.c:2903]
~~Dr.M~~ # 1 main
~~Dr.M~~
~~Dr.M~~ ERRORS FOUND:
~~Dr.M~~       1 unique,     2 total unaddressable access(es)
~~Dr.M~~       0 unique,     0 total uninitialized access(es)
~~Dr.M~~       0 unique,     0 total invalid heap argument(s)
~~Dr.M~~       0 unique,     0 total GDI usage error(s)
~~Dr.M~~       0 unique,     0 total handle leak(s)
~~Dr.M~~       0 unique,     0 total warning(s)
~~Dr.M~~       2 unique,     2 total,     16 byte(s) of leak(s)
~~Dr.M~~       1 unique,     1 total,     26 byte(s) of possible leak(s)
~~Dr.M~~ ERRORS IGNORED:
~~Dr.M~~       3 potential error(s) (suspected false positives)
```

Constructori și virtualizare

Cine ar trebui să elibereze memoria? Adăugăm destructorul și nu mai avem leaks.

```
class Instrument {
public:
   virtual ~Instrument() = 0;
   virtual void play(int) const {} };
Instrument::~Instrument() = default;
class Wind : public Instrument { /* implementare play */ };
class String : public Instrument { /* implementare play */ };
class Orchestra {
   std::vector<Instrument*> instruments;
public:
   void add(Instrument* inst) { instruments.push_back(inst); }
   void rehearse() { for(const auto& inst : instruments)  inst->play(0); }
   ~Orchestra() { for(auto* inst : instruments)  delete inst; }
};
int main() { Orchestra o1; o1.add(new Wind); o1.add(new String); o1.rehearse(); }
```

Constructori și virtualizare

Nu mai avem memory leaks, dar... este corect?

```
class Instrument {
public:
   virtual ~Instrument() = 0;
   virtual void play(int) const {} };
Instrument::~Instrument() = default;
class Wind : public Instrument { /* implementare play */ };
class String : public Instrument { /* implementare play */ };
class Orchestra {
   std::vector<Instrument*> instruments;
public:
   void add(Instrument* inst) { instruments.push_back(inst); }
   void rehearse() { for(const auto& inst : instruments)  inst->play(0); }
   ~Orchestra() { for(auto* inst : instruments)  delete inst; }
};
int main() { Orchestra o1; o1.add(new Wind); o1.add(new String); o1.rehearse();
   Orchestra o2 = o1;  // sau Orchestra o3(o1);
}
```

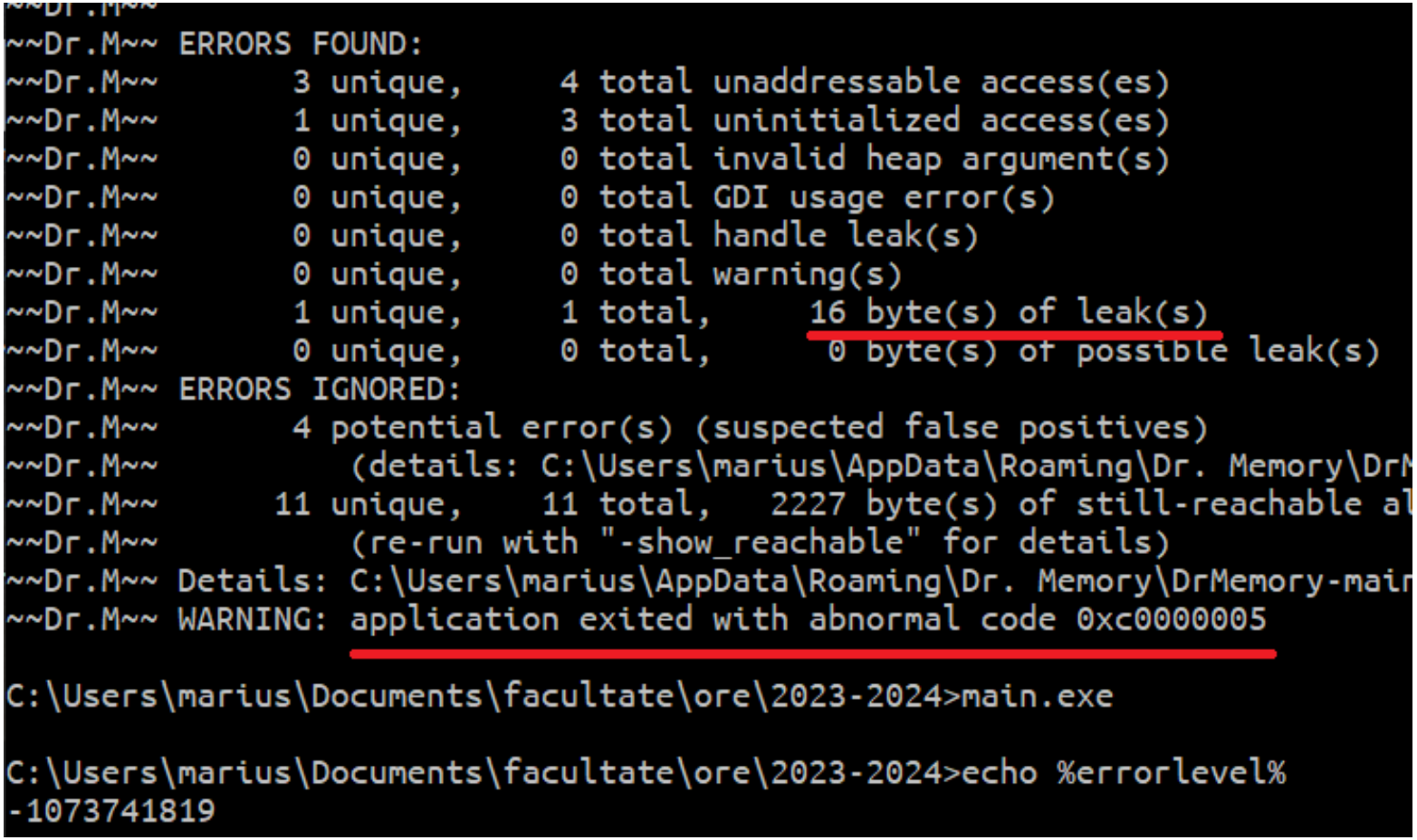
Constructori și virtualizare

Nu, cedeaz ă la execuție.
Dacă avem atribute de tip pointeri, constructorul de copiere copiază adrese de memorie.

```
~~Dr.M~~ Error #2: UNADDRESSABLE ACCESS of freed memory: reading 0x00000176cb2a0840-0x00000176cb2a0848 8 byte(s)
~~Dr.M~~ # 0 vec::~vec
~~Dr.M~~ # 1 main
~~Dr.M~~ Note: @0:00:00.434 in thread 7412
~~Dr.M~~ Note: next higher malloc: 0x00000176cb2a08d0-0x00000176cb2a08e0
~~Dr.M~~ Note: prev lower malloc:  0x00000176cb2a06a0-0x00000176cb2a07a0
~~Dr.M~~ Note: 0x00000176cb2a0840-0x00000176cb2a0848 overlaps memory 0x00000176cb2a0840-0x00000176cb2a0848 that was free
d here:
~~Dr.M~~ Note: # 0 replace_operator_delete_nothrow              [D:\a\drmemory\drmemory\common\alloc_replace.c:2978]
~~Dr.M~~ Note: # 1 derivata1::~derivata1
~~Dr.M~~ Note: # 2 vec::~vec
~~Dr.M~~ Note: # 3 main
~~Dr.M~~ Note: instruction: mov    (%rax) -> %rdx
~~Dr.M~~
~~Dr.M~~ Error #3: UNADDRESSABLE ACCESS beyond heap bounds: reading 0x00000176cb2a0888-0x00000176cb2a0890 8 byte(s)
~~Dr.M~~ # 0 vec::~vec
~~Dr.M~~ # 1 main
```

Constructori și virtualizare

Nu, cedează la execuție.
Dacă avem atribute de tip pointeri, constructorul de copiere copiază adrese de memorie.



Constructori și virtualizare

Soluție: suprascriem constructorul de copiere, dar... **Ce copiem?**

```
class Orchestra {
   std::vector<Instrument*> instruments;
public:
   void add(Instrument* inst) { instruments.push_back(inst); }
   void rehearse() { for(const auto& inst : instruments)  inst->play(0); }
   ~Orchestra() { for(auto* inst : instruments)  delete inst; }
   Orchestra() = default;
   Orchestra(const Orchestra& other) {
      for(const auto& inst : other.instruments)
         instruments.push_back(new ???);   // new Wind() sau new String()???
   }
};
```

Constructori și virtualizare

"Soluția" 1: atribut pentru subtip.

```
class Instrument {
public:
   enum tip {wind, string};
   virtual ~Instrument() = 0;    virtual void play(int) const {} };
Instrument::~Instrument() = default;
class Wind : public Instrument { /* inițializare tip cu wind, implementare play */ };
class String : public Instrument { /* inițializare tip cu string, implementare play */ };
class Orchestra {
   std::vector<Instrument*> instruments;
public:    Orchestra() = default;
   Orchestra(const Orchestra& other) {
      for(const auto& inst : other.instruments)
         if(inst.getTip() == Instrument::wind)
            instruments.push_back(new Wind(static_cast<Wind*>(inst)));  // apelăm cc
         else if // etc
}};
```

Constructori și virtualizare

Problema cu "soluția" 1: trebuie să actualizăm de fiecare dată clasa de bază atunci când adăugăm o nouă derivată (recompilăm toate subclasele).
"Soluția" 2: dynamic_cast.

```
class Instrument {
public:
   virtual ~Instrument() = 0;    virtual void play(int) const {} };
Instrument::~Instrument() = default;
class Wind : public Instrument { /* implementare play */ };
class String : public Instrument { /* implementare play */ };
class Orchestra {
   std::vector<Instrument*> instruments;
public:    Orchestra() = default;
   Orchestra(const Orchestra& other) {
      for(const auto& inst : other.instruments)
         if(auto* wind = dynamic_cast<Wind*>(inst))
            instruments.push_back(new Wind(*wind));
         else if // etc
}};
```

Constructori și virtualizare

Problema cu "soluția" 2: peste tot unde avem nevoie de o copie a unui instrument va trebui să avem ramuri if/else și să facem un dynamic_cast (sau typeid + static_cast).

Pentru fiecare nouă derivată va trebui să adăugăm **peste tot** câte o nouă ramură if/else.

Soluția corectă 1: nu permitem copieri, folosim doar mutări. Dezavantaj dpdv didactic: necesare multe apeluri std::move

```
class Orchestra {
    std::vector<Instrument*> instruments;
public:
    Orchestra() = default;
    Orchestra(const Orchestra& other) = delete;
    Orchestra& operator=(const Orchestra& other) = delete;
    Orchestra(Orchestra&& other) = default;
    Orchestra& operator=(Orchestra&& other) = default;
    ~Orchestra() = default;
};
```

Constructori și virtualizare

Soluția corectă 2 (funcție clone): fiecare subclasă ar trebui să știe să se copieze pe sine.

Având în vedere că în clasa Orchestra avem doar pointeri de tip bază, pentru copiere vom folosi o **funcție virtuală.**

```
class Instrument {
public:
   virtual ~Instrument() = default;  // sau = 0
   virtual void play(int) const {} };
   virtual Instrument* clone() const = 0; };

class Wind : public Instrument { /* implementare play */
public:  Instrument* clone() const override { return new Wind(*this); } };  // apelăm cc
};
class String : public Instrument { /* implementare play */
public:  Instrument* clone() const override { return new String(*this); } };
};
```

Constructori și virtualizare

Soluția corectă 2 (funcție clone)
Clasa Orchestra se transformă în felul următor:

```
class Orchestra {
   std::vector<Instrument*> instruments;
public:
   void add(Instrument* inst) { instruments.push_back(inst->clone()); }
   void rehearse() { for(const auto& inst : instruments)  inst->play(0); }
   Orchestra() = default;
   ~Orchestra() { for(auto* inst : instruments)  delete inst; }
   Orchestra(const Orchestra& other) {
      for(const auto& inst : other.instruments)
         instruments.push_back(inst->clone());    }
};
```

Apelăm funcția clone și în funcția "add" deoarece vrem să fim siguri că obiectul de tip Orchestra **deține toate resursele sale**, deci nu depinde de ce se va întâmpla cu parametrii.

Constructori și virtualizare

Soluția corectă 2 (funcție clone)
Clasa Orchestra se transformă în felul următor:

```
class Orchestra {
   std::vector<Instrument*> instruments;
public:
   void add(const Instrument& inst) { instruments.push_back(inst.clone()); }
   Orchestra() = default;
   ~Orchestra() { for(auto* inst : instruments)  delete inst; }
   Orchestra(const Orchestra& other) {
      for(const auto& inst : other.instruments)
         instruments.push_back(inst->clone());    }
};
int main() {
   Orchestra o1;    // o1.add(new Wind); // aici am avea memory leak
   String *s1 = new String; o1.add(*s1); delete s1;  // o1 are în continuare o copie lui s1
   String s2; o1.add(s2);  /* nu se pune problema să facem delete la o variabilă locală */
}
```

Constructori și virtualizare

Este corect acum?
Copierile funcționează, dar nu și atribuirile:

```
int main() {
    Orchestra o1;
    String s; o1.add(s);
    Orchestra o2 = o1;    //  apel constructor de copiere
    o2 = o1;              //  apel operator=
}
```

Constructori și virtualizare

Este corect acum?

Copierile funcționează, dar nu și atribuirile:

```
int main() {
    Orchestra o1;
    String s; o1.add(s);
    Orchestra o2 = o1;
    o2 = o1;
}
```

```
~~Dr.M~~ Error #2: UNADDRESSABLE ACCESS of freed memory: reading 0x000002d2690e0840-0x000002d2690e0848 8 byte(s)
~~Dr.M~~ # 0 vec::~vec
~~Dr.M~~ # 1 main
~~Dr.M~~ Note: @0:00:00.422 in thread 1060
~~Dr.M~~ Note: next higher malloc: 0x000002d2690e08d0-0x000002d2690e08e0
~~Dr.M~~ Note: prev lower malloc:  0x000002d2690e06a0-0x000002d2690e07a0
~~Dr.M~~ Note: 0x000002d2690e0840-0x000002d2690e0848 overlaps memory 0x000002d2690e0840-0x000002d2690e0848 that was free
d here:
~~Dr.M~~ Note: # 0 replace_operator_delete_nothrow              [D:\a\drmemory\drmemory\common\alloc_replace.c:2978]
~~Dr.M~~ Note: # 1 derivata1::~derivata1
~~Dr.M~~ Note: # 2 vec::~vec
~~Dr.M~~ Note: # 3 main
~~Dr.M~~ Note: instruction: mov    (%rax) -> %rdx
```

Constructori și virtualizare

Este corect acum?

Copierile funcționează, dar nu și atribuirile:

```
int main() {
    Orchestra o1;
    String s; o1.add(s);
    Orchestra o2 = o1;
    o2 = o1;
}
```

```
ERRORS FOUND:
      3 unique,     4 total unaddressable access(es)
      1 unique,     3 total uninitialized access(es)
      0 unique,     0 total invalid heap argument(s)
      0 unique,     0 total GDI usage error(s)
      0 unique,     0 total handle leak(s)
      0 unique,     0 total warning(s)
      3 unique,     3 total,     32 byte(s) of leak(s)
      0 unique,     0 total,      0 byte(s) of possible le
ERRORS IGNORED:
      4 potential error(s) (suspected false positives)
         (details: C:\Users\marius\AppData\Roaming\Dr. Mem

     11 unique,    11 total,   2227 byte(s) of still-reach
         (re-run with "-show_reachable" for details)
Details: C:\Users\marius\AppData\Roaming\Dr. Memory\DrMemo
WARNING: application exited with abnormal code 0xc0000005
```

Constructori și virtualizare

Suprascriem și operator=

```
class Orchestra {
   std::vector<Instrument*> instruments;
public:
   void add(const Instrument& inst) { instruments.push_back(inst.clone()); }
   Orchestra() = default;
   ~Orchestra() { for(auto* inst : instruments)  delete inst; }
   Orchestra(const Orchestra& other) {
      for(const auto& inst : other.instruments)
         instruments.push_back(inst->clone());    }
   Orchestra& operator=(const Orchestra& other) {    if(this == &other)   return *this;
      for(auto* inst : instruments)  delete inst;
      instruments.clear();
      for(const auto& inst : other.instruments)
         instruments.push_back(inst->clone());     return *this;      }
};
```

Regula celor trei

Dacă într-o clasă trebuie să suprascriem cc/op=/destr, cel mai probabil trebuie suprascrise toate cele trei funcții speciale.

```
class Orchestra {
   std::vector<Instrument*> instruments;
public:
   void add(const Instrument& inst) { instruments.push_back(inst.clone()); }
   Orchestra() = default;
   ~Orchestra() { for(auto* inst : instruments)  delete inst; }
   Orchestra(const Orchestra& other) {
      for(const auto& inst : other.instruments)
         instruments.push_back(inst->clone());    }
   Orchestra& operator=(const Orchestra& other) {    if(this == &other)   return *this;
      for(auto* inst : instruments)  delete inst;
      instruments.clear();
      for(const auto& inst : other.instruments)
         instruments.push_back(inst->clone());
      return *this;  }
};
```

Constructori și virtualizare

```
class Orchestra {
   std::vector<Instrument*> instruments;
public:
   void add(const Instrument& inst) { instruments.push_back(inst.clone()); }
   Orchestra() = default;
   ~Orchestra() { for(auto* inst : instruments)  delete inst; }
   Orchestra(const Orchestra& other) {
      for(const auto& inst : other.instruments)
         instruments.push_back(inst->clone());    }
   Orchestra& operator=(const Orchestra& other) {    if(this == &other)   return *this;
      for(auto* inst : instruments)  delete inst;
      instruments.clear();
      for(const auto& inst : other.instruments)
         instruments.push_back(inst->clone());   // ce se întâmplă dacă nu reușește copierea?
      return *this;  }
};
```

Obiectul va fi într-o stare invalidă! Am pierdut datele vechi și nu am copiat datele noi.

Copy and swap

Soluția 2.5: Trebuie să efectuăm copierea noilor atribute **înainte** de a șterge datele vechi.

Pentru copiere **refolosim** implementarea din constructorul de copiere, iar pentru eliberarea vechilor resurse **refolosim** implementarea destructorului:

```
class Orchestra {
   std::vector<Instrument*> instruments;
public:
   Orchestra() = default;
   ~Orchestra() { for(auto* inst : instruments)  delete inst; }
   Orchestra(const Orchestra& other) {
      for(const auto& inst : other.instruments)    instruments.push_back(inst->clone());    }
   Orchestra& operator=(const Orchestra& other) {    if(this == &other)   return *this;
      auto copie = other;   // aici se apelează constructorul de copiere
      std::swap(this->instrumente, copie.instrumente);
      return *this;
   }  // aici se apelează destructorul pentru copie
};
```

Copy and swap

Soluția 2.6: Pentru copiere putem apela constructorul de copiere transmițând parametrul de la operator= prin valoare, simplificând astfel codul:

```
class Orchestra {
   std::vector<Instrument*> instruments;
public:
   Orchestra() = default;
   ~Orchestra() { for(auto* inst : instruments)  delete inst; }
   Orchestra(const Orchestra& other) {
      for(const auto& inst : other.instruments)    instruments.push_back(inst->clone());    }
   Orchestra& operator=(Orchestra other) // aici se apelează constructorul de copiere
   {
      if(this == &other)   return *this;
      std::swap(this->instrumente, other.instrumente);
      return *this;
   }  // aici se apelează destructorul pentru copie
};
```

Copy and swap

Caz general: partea de swap poate fi refolosită în alte situații, motiv pentru care definim o funcție separată de swap:

```
class Orchestra {
   std::vector<Instrument*> instruments;
public:
   Orchestra() = default;
   ~Orchestra() { for(auto* inst : instruments)  delete inst; }
   Orchestra(const Orchestra& other) {
      for(const auto& inst : other.instruments)    instruments.push_back(inst->clone());    }
   Orchestra& operator=(Orchestra other) // aici se apelează constructorul de copiere
   {
      if(this == &other)   return *this;
      swap(this, other);
      return *this;
   }  // aici se apelează destructorul pentru copie
};
```

Copy and swap

Caz general: partea de swap poate fi refolosită în alte situații, motiv pentru care definim o funcție separată de swap:

```
class Orchestra {
   // codul anterior
   friend void swap(Orchestra& o1, Orchestra& o2) {
      using std::swap;
      swap(o1.instrumente, o2.instrumente);
      //swap(o1.dirijor, o2.dirijor);
   }
};
```

De ce funcție friend?

- Dacă și în altă clasă avem o funcție de swap, aceasta va fi găsită de ADL
- ADL (argument dependent lookup) găsește (doar) funcții friend

De ce "using std::swap"?

- Pentru a scrie la fel toate apelurile de swap, nu unele cu std:: și altele doar swap

RAII (resource acquisition is initialization)

Ideea de gestionare a resurselor în C++ este aceea că resursele ar trebui alocate doar în constructori și eliberate doar în destructori.

De ce?

Constructorii și destructorii sunt apelați automat de limbaj. Dacă nu facem alocări/dezalocări în alte locuri, este imposibil să avem memory leaks într-un program corect.

Așadar, spre deosebire de multe alte limbaje OOP, nu apare necesitatea de garbage collection: folosind RAII nu lăsăm în urmă "gunoaie".

Exemplu de RAII din limbaj: smart pointers.

RAII (resource acquisition is initialization)

Pentru a folosi smart pointers în funcțiile de clone, avem de făcut doar aceste modificări:
- Înlocuim Instrument* cu std::shared_ptr<Instrument>
- Înlocuim new Wind(args) cu std::make_shared<Wind>(args) (idem pentru String)
- Eliminăm apelurile de delete

Avantaje:
- Nu avem nevoie de destructori expliciți în care să facem delete
- Este corect să facem apeluri de forma

```
func(std::make_shared<T1>(), std::make_shared<T2>())
```

Dezavantaj:
- Nu putem folosi tipuri de date covariante

Tipuri de date covariante

```
#include <iostream>
class Baza {
public:
    virtual ~Baza() = default;
    virtual Baza* clone() const = 0;
};

class Derivata1 : public Baza {
public:
    Baza* clone() const override {
        return new Derivata1(*this);
    }
    void f() { std::cout << "f der1\n"; }
};
class Derivata2 : public Baza {
public:
    Derivata2* clone() const override {
        return new Derivata2(*this);
    }
    void g() { std::cout << "g der2\n"; }
};
```

```
int main() {
    Baza* b1 = new Derivata1;
    // Derivata1* d1 = b1->clone();  // eroare
    // b1->f();  // eroare
    delete b1;
    Baza* b2 = new Derivata2;
    Derivata2 d2;
    // Derivata2* d2_1 = b2->clone();  // eroare
    Derivata2* d2_2 = d2.clone();  // ok
    d2_2->g();  // ok
    delete b2;
    delete d2_2;
}
```

Interfață non-virtuală (NVI)

Utilitate: toate clasele derivate au o implementare comună și au nevoie să suprascrie doar anumite porțiuni.

Exemplu fără NVI: observăm că doar o mică parte din implementările din derivate diferă.

```
class Instrument {
public:
   virtual ~Instrument() = default;  virtual void play(int) const {} };
```

```
class Wind : public Instrument {
public:
   void play(int note) const override {
      readNote(note);
      prepare();
      std::cout << "play wind\n";
      rest(10ms);
   }
};
```

```
class String : public Instrument {
public:
   void play(int note) const override {
      readNote(note);
      prepare();
      std::cout << "play string\n";
      rest(10ms);
   }
};
```

Interfață non-virtuală (NVI)

Exemplu cu NVI: mutăm partea comună în clasa de bază, iar derivatele suprascriu strict partea care diferă.

```
class Instrument {
public:
   virtual ~Instrument() = default;
   void perform(int note) const {
      readNote(note);   prepare();   play(note);   rest(10ms);
}
private:
   virtual void play(int) const {} };
```

```
class Wind : public Instrument {
public:
   void play(int note) const override {
      std::cout << "play wind\n";
   }
};
```

```
class String : public Instrument {
public:
   void play(int note) const override {
      std::cout << "play string\n";
   }
};
```

Interfață non-virtuală (NVI)

Prin această abordare, este mult mai ușor să modificăm în mod uniform structura implementării la nivelul întregii ierarhii (și avem de recompilat un singur fișier).

În plus, prin NVI forțăm ca apelul să se realizeze doar prin funcția publică non-virtuală, ceea ce înseamnă că o derivată nu va putea "ocoli" implementarea comună impusă de clasa de bază.

Exemple de situații:
- setup/cleanup comun, testare, benchmarks
- rezolvă unele probleme de la moștenirea multiplă
- vezi mai târziu și TemplateMethod pattern, D din SOLID

2. Downcasting

Folosit in ierarhii polimorfice (cu funcţii virtuale) pentru a obține înapoi derivata către care arată un pointer de tip bază.

Problema: upcasting e sigur pentru că respectivele funcţii/atribute trebuie să fie definite în bază, downcasting e problematic.

Explicit cast prin: **dynamic_cast**

Dacă știm cu siguranță tipul obiectului, putem folosi "static_cast".

static_cast întoarce pointer către obiectul care satisface cerinţele sau 0.

dynamic_cast folosește tabelele VTABLE pentru determinarea tipului.

Conversiile tip C++ documentează de ce avem nevoie de un anumit cast

- Numele conversiilor sunt lungi pentru a atrage atenția (și pentru a le evita)

Conversiile tip C nu exprimă ce intenție avem și anulează verificările compilatorului.

2. Downcasting

Problema: downcasting e problematic și din alt motiv: dacă avem de făcut "if" pe tipuri de date, ori nu este abstractizarea bună, ori trebuie să folosim de fapt funcții virtuale.

Un program cu multe comparații explicite de subclase este ușor de scris pe termen scurt (exemplu: un semestru), dar dificil de întreținut și extins pe termen mediu/lung.

dynamic_cast sau typeid + static_cast?

dynamic_cast este mai lent, dar codul va funcționa în continuare dacă transmitem un pointer și mai derivat (este "mai polimorfic"): "este un fel de derivata1"

typeid este mai rapid, dar compară un singur tip de date: "este exact derivata1"

2. Downcasting

dynamic_cast

```
class Pet { public: virtual ~Pet(){}};
class Dog : public Pet {};
class Cat : public Pet {};

int main() {
    Pet* b = new Cat; // Upcast
    if (auto* d1 = dynamic_cast<Dog*>(b)) {
        std::cout << "d1 = " << d1 << std::endl;
    }
    else if (auto* d2 = dynamic_cast<Cat*>(b)) {
        std::cout << "d2 = " << d2 << std::endl;  // b și d2 rețin aceeași adresă
        std::cout << "b  = " << b << std::endl;
    }
    try{
        auto& d1 = dynamic_cast<Dog&>(*b);  // dacă nu reușește, se aruncă excepție
        auto& d2 = dynamic_cast<Cat&>(*b);
    } catch(std::bad_cast&) { std::cout << "eroare cast\n"; }
}
```

2. Downcasting

typeid (sau atribut intern) + static_cast

```
class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
  Circle c;
  Shape* s = &c; // Upcast: normal and OK
  // More explicit but unnecessary:
  s = static_cast<Shape*>(&c);
  // (Since upcasting is such a safe and common
  // operation, the cast becomes cluttering)
  Circle* cp = 0;
  Square* sp = 0;
  // Static Navigation of class hierarchies
requires extra type information:
  if(typeid(s) == typeid(cp)) // C++ RTTI
    cp = static_cast<Circle*>(s);
  if(typeid(s) == typeid(sp))
    sp = static_cast<Square*>(s);
  if(cp != 0)
    cout << "It's a circle!" << endl;
  if(sp != 0)
    cout << "It's a square!" << endl;
  // Static navigation is ONLY an efficiency
hack;
  // dynamic_cast is always safer. However:
  // Other* op = static_cast<Other*>(s);
  // Conveniently gives an error message,
while
  Other* op2 = (Other*)s;
  // does not
}
```

3. Moștenire multiplă și virtuală

Moștenire multiplă (MM)

- puține limbaje au MM;

- moștenirea multiplă e complicată: ambiguitate LA MOȘTENIREA IN ROMB / IN DIAMANT;
  - Soluție: interfață non-virtuală
- nu e nevoie de MM (se simulează cu moștenire simplă);

- se moștenește în același timp din mai multe clase;

Sintaxa:

class Clasa_Derivată : [modificatori de acces] Clasa_de_Bază1, [modificatori de acces] Clasa_de_Bază2, [modificatori de acces] Clasa_de_Bază3 .....

3. Moștenire multiplă și virtuală

Moștenire multiplă (MM)

- dar dacă avem nevoie doar de o copie a lui i?
- nu vrem să consumăm spațiu în memorie;
- ***folosim moștenire virtuală***:

```
class base {   public:      int i; };
class derived1 : virtual public base {  public:      int j; };
class derived2 : virtual public base {   public:      int k; };
class derived3 : public derived1, public derived2 {public:     int sum; };
```

- Dacă avem moștenire de două sau mai multe ori dintr-o clasă de bază (fiecare moștenire trebuie să fie virtuală) atunci compilatorul alocă spațiu pentru o singură copie;
- În clasele derived1 și 2 moștenirea e la fel ca mai înainte (niciun efect pentru virtual în acel caz)
- Dar... fiecare moștenire virtuală crește sizeof-ul unui obiect al acelei clase

3. Moștenire multiplă și virtuală

Moștenire multiplă (MM)

- dacă în clasa de bază avem doar constructor cu parametri, derivatele trebuie să apeleze explicit acest constructor
  - de ce? Clasa de bază se inițializează o singură dată, la început

```
class base {      int i;   public:  base(int z) : i(z) {} };
class derived1 : virtual public base {      int j;    public:  derived1() : base(1) {} };
class derived2 : virtual public base {      int k;    public: derived2() : base(2) {} };
class derived3 : public derived1, public derived2 {
   int sum;
public:
   derived3() : base(3), derived1(), derived2() {} };
```

3. Moștenire multiplă și virtuală

Moștenire multiplă (MM)

- În exemplul următor afișăm atributele din derivate printr-un pointer de bază folosind o implementare naivă a funcției virtuale de afișare

```
#include <iostream>
class Baza {
   int i{1};
protected:   virtual void afis(std::ostream& os) const {     os << "i: " << i << "\n";    }
public:
   virtual ~Baza() = default;
   friend std::ostream& operator<<(std::ostream& os, const Baza& b) {    b.afis(os);  return os; }  };

class Der1 : public virtual Baza {
   int j{2};
protected:    void afis(std::ostream& os) const override {  Baza::afis(os);  os << "j: " << j << "\n"; } };

class Der2 : public virtual Baza {
   int k{3};
protected:    void afis(std::ostream& os) const override { Baza::afis(os); os << "k: " << k << "\n"; } };
```

Moștenire multiplă (MM)

- Observăm că atributele din clasa de bază se afișează de două ori

```
class Der3 : public virtual Der1, public virtual Der2 {
    int l{4};
    void afis(std::ostream& os) const override {
        Der1::afis(os);
        Der2::afis(os);
        os << "l: " << l << "\n";
    }
};
int main() {
    Baza* b = new Der3;
    std::cout << *b;
    delete b;
}
// se va afișa
i: 1
j: 2
i: 1
k: 3
l: 4
```

3. Moștenire multiplă și virtuală

Moștenire multiplă (MM)

- Soluție: aplicăm ideea de interfață non-virtuală (NVI)
- operator<< este funcția publică non-virtuală
- Este suficient să modificăm clasa de bază ca mai jos, iar în Der1 și Der2 să eliminăm apelul Baza::afis():

```
class Baza {
    int i{1};
protected:
    virtual void afis(std::ostream& os) const {}
public:
    virtual ~Baza() = default;
    friend std::ostream& operator<<(std::ostream& os, const Baza& b) {
        os << "i: " << b.i << "\n";
        b.afis(os);
        return os;
    }
};
```

```
// se va afișa
i: 1
j: 2
k: 3
l: 4
```

4. Tratarea excepțiilor în C++

În urma execuției unui program pot apărea diverse erori.

Câteva mecanisme de tratare a erorilor:
- Coduri de eroare
- Aserțiuni
- Excepții
- Tipuri de date rezultat (vezi anul 3 semestrul 2)

Excepțiile (în C++) pot fi cauzate
- în mod implicit de către limbaj (alocare dinamică) și de funcții din biblioteca standard (argumente invalide, erori de conversie)
- în mod explicit de către noi

Scop: simplificarea tratării erorilor

Sintaxă: într-un bloc try/catch prindem excepții aruncate cu throw, iar în fiecare clauză catch tratăm un anumit tip de eroare

4. Tratarea excepțiilor în C++

```
try {
  // try block
}
catch (type1 arg) {
  // catch block
}
catch (type2 arg) {
  // catch block
}
catch (type3 arg) {
  // catch block
}...
catch (typeN arg) {
  // catch block
}
```

tipul argumentului arg din catch arată care bloc catch este executat

dacă nu este generată excepție, nu se execută nici un bloc catch

instrucțiunile catch sunt verificate în ordinea în care sunt scrise, primul de tipul erorii este folosit

4. Tratarea excepțiilor în C++

Observații:

- dacă se face throw şi nu există un bloc try din care a fost aruncată excepția sau o funcție apelată dintr-un bloc try: eroare

- dacă nu există un catch care să fie asociat cu throw-ul respectiv (tipuri de date egale sau compatibile) atunci programul se termină prin terminate()

- terminate() poate să fie redefinită să facă altceva

- Nu se recomandă folosirea excepțiilor dacă locul unde are loc eroarea este foarte apropiat de catch-ul asociat
  - Mai simplu și mai clar folosind coduri de eroare

4. Tratarea excepțiilor în C++

Toate [excepțiile standard](#) moștenesc din std::exception

Multe dintre ele sunt în headerele <exception> sau <stdexcept>

Standard exceptions

- logic_error
  - invalid_argument
  - domain_error
  - length_error
  - out_of_range
  - future_error (since C++11)
- runtime_error
  - range_error
  - overflow_error
  - underflow_error
  - regex_error (since C++11)
  - system_error (since C++11)
    - ios_base::failure (since C++11)
    - filesystem::filesystem_error (since C++17)
  - tx_exception (TM TS)
  - nonexistent_local_time (since C++20)
  - ambiguous_local_time (since C++20)
  - format_error (since C++20)

4. Tratarea excepțiilor în C++

- aruncarea de erori din clase de bază şi derivate
- un catch pentru tipul de bază va fi executat pentru un obiect aruncat de tipul derivat
- să se pună catch-ul pe tipul derivat primul şi apoi catchul pe tipul de bază

```
class B { };
class D: public B { };
int main()
{
      D derived;
      try {     throw derived;   }
      catch(B b) {     cout << "Caught a base class.\n";   }
      catch(D d) {     cout << "This won't execute.\n";   }
      // primim warning la compilare
return 0;
}
```

4. Tratarea excepțiilor în C++

Când ar trebui să aruncăm excepții?

- În constructori și în funcții care aruncă obiecte dacă obiectul rezultat nu ar fi valid
  - împiedicăm construirea unui obiect invalid
  - Execuția sare la primul catch care se potrivește
- Atunci când alternativa cu coduri de eroare este mai complicată
- Atunci când codul este mai clar de înțeles cu excepții decât fără
  - Dacă avem separare clară între happy path și bad path
- Dacă avem voie

Ce punem în catch?

- Prindem excepțiile prin referință (const dacă nu modificăm)
- Încercăm să găsim un echilibru între a prinde doar erori cât mai generale (cod simplu) și erori specifice (util la depanare)

4. Tratarea excepțiilor în C++

- Limbajul C++ ne permite să ne definim o ierarhie de excepții de la zero
- De obicei nu este recomandat, deoarece modulul scris de noi va trebui tratat în mod special față de restul codului

- Prima idee: moștenim din std::runtime_error (sau logic_error)
  - Putem moșteni direct din std::exception, dar nu avem constructor cu mesaj de eroare

- Problemă: nu putem face distincția dintre excepții aruncate de codul nostru și excepții aruncate de biblioteca standard (sau de alte module/biblioteci)

4. Tratarea excepțiilor în C++

```
#include <fmi>
class StudentError : public std::runtime_error {
   using std::runtime_error::runtime_error;
};
class TeacherError : public std::runtime_error {
   using std::runtime_error::runtime_error;
};
int main() {
   try {
      Student st1, st2; Teacher t1, t2;
      st1.studyLate();  t1.prepareLate();  // might throw StudentError/TeacherError
      // might throw PPTError : public std::runtime_error
      FMIComputer::loadPPT(t1.getLecture());
      st2.studyLate();  t2.prepareLate();  // might throw StudentError/TeacherError
   } catch(std::runtime_error& err) {
      std::cout << err.what() << "\n";   // am prins eroare din codul nostru sau din <fmi>?
   }
}
```

4. Tratarea excepțiilor în C++

Soluție: fiecare modul ar trebui să aibă o ierarhie proprie de excepții cu baza derivată direct sau indirect din std::exception.

Ierarhia noastră devine următoarea:

```
class SchoolError : public std::runtime_error { using std::runtime_error::runtime_error; };
class StudentError : public SchoolError {  using SchoolError::SchoolError;  };
class TeacherError : public SchoolError {  using SchoolError::SchoolError;  };
```

4. Tratarea excepțiilor în C++

Soluție: fiecare modul ar trebui să aibă o ierarhie proprie de excepții cu baza derivată direct sau indirect din std::exception.

Codul din main:

```
int main() {
   try {
      Student st1, st2; Teacher t1, t2;
      st1.studyLate();  t1.prepareLate();  // might throw StudentError/TeacherError
      FMIComputer::loadPPT(t1.getLecture());  // might throw PPTError : public FMIError
      st2.studyLate();  t2.prepareLate();  // might throw StudentError/TeacherError
   } catch(SchoolError& err) {
      std::cout << err.what() << "\n";
   } catch(FMIError& err) {
      std::cout << err.what() << "\n";
   } catch(std::runtime_error& err) {
      std::cout << err.what() << "\n"; // alte erori
   }
}
```

4. Tratarea excepțiilor în C++

Observații:

void Xhandler1(int test) noexcept
void Xhandler2(int test) noexcept(false)

- se poate specifica dacă o funcție aruncă excepții sau nu

- re-aruncarea unei excepții: throw; // fără excepție din catch
  - Util pentru handlers care tratează erori comune
  - Atenție! throw err; efectuează o copie prin valoare
    - Poate cauza object slicing (la fel catch prin valoare)
    - Dacă totuși avem nevoie: https://isocpp.org/wiki/faq/exceptions#throwing-polymorphically

4. Tratarea excepțiilor în C++

Rearuncarea unei excepții

- re-aruncarea unei excepții: throw; // fără excepție din catch

Avem ierarhia următoare:

```
class AppError : public std::runtime_error {
    using std::runtime_error::runtime_error;
};
class SomeCommonError : public AppError {
    using AppError::AppError;
};
class OtherCommonError : public AppError {
    using AppError::AppError;
};
class SomeSpecialError : public AppError {
    using AppError::AppError;
};
class OtherSpecialError : public AppError {
    using AppError::AppError;
};
```

4. Tratarea excepțiilor în C++

Rearuncarea unei excepții

```
void handleCommonErrors() {
    try {
        throw;
    } catch(SomeCommonError& err) { /* handle error */
    } catch(OtherCommonError& err) { /* handle error */
      catch(AppError& err) { /* handle error */
    }
}

void f() {
    try {
        // cod care aruncă diverse erori
    } catch(SomeSpecialError& err) { /* handle error */
    } catch(OtherSpecialError& err) { /* handle error */
    } catch(...) { handleCommonErrors(); }
}
```

Perspective

Cursul 9:

Șabloane