

Cursul de programare orientata pe obiecte

Seria 13

Saptamana 1, 18 feb 2014

Andrei Paun

Cuprinsul cursului: 18 feb. 2014

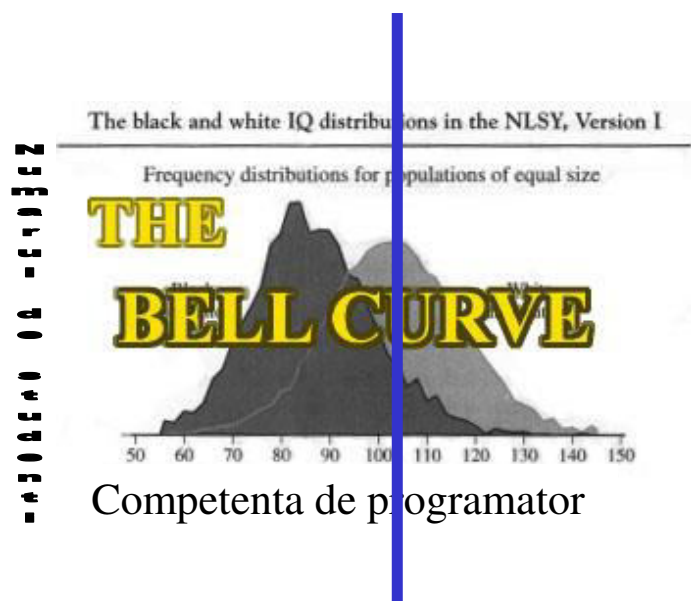
- Sa ne cunoastem
- Generalitati despre curs
- Reguli de comportament
- Generalitati despre OOP
- Sa ne cunoastem mai bine...

Sa ne cunoastem

- Cine este Andrei Paun
- Romania, Canada, SUA
- apaun@fmi.unibuc.ro
- andreipaun@gmail.com

Sa ne cunoastem

- Cine sunt studentii?
- Ideea cursului:



Generalitati despre curs

- Cursul:
- Marti 14-16 seria 13; Amfitreatul 2: Pompeiu
- Laborator: OBLIGATORIU
- Seminar: din doua in doua saptamani
- **Examen: 20 iunie 2014**

Generalitati despre curs

- Curs de programare OO
- Oferă o baza de pornire pentru alte cursuri
- C++
- Programare orientata pe obiect: orientarea pe clase
 - Diferit de
 - programare OO: orientarea pe prototip
 - programare procedurala (imperativa)
 - Programare functionala
 - Programare logica
 - Programare vizuala, etc.
 - Self, JavaScript
 - Pascal, C, Fortran
 - Lisp, Scheme, R
 - Prolog
 - Quartz Composer

Reguli si sugestii pentru curs

- Fara celulare
- Fara galagie
- Cu cat mai multe intrebari
- Moodle
- E-mail
- calculator

Important

- Fara celulare
- Fara deranjat colegii
- Prezenta la curs/seminar: nu e obligatorie
- Nota mare/de trecere pentru curs:
 - nu e obligatorie
- Laboratoarele: obligatorii!

Important II

- Fara copiat
- acces la calculator

Organizatorice

- Examenul
 - dupa definitivarea orarului programarea examenului
 - impreuna seriile 13 si 14
 - Conf. Radu Gramatovici

Programa cursului

1. Principiile programarii orientate pe obiecte
2. Proiectarea ascendenta a claselor. Incapsularea datelor in C++
3. Supraincarcarea functiilor si operatorilor in C++
4. Proiectarea descendenta a claselor. Mostenirea in C++
5. Constructori si destructori in C++
6. Modificatori de protectie in C++. Conversia datelor in C++
7. Mostenirea multipla si virtuala in C++
8. Membrii constanti si statici ai unei clase in C++
9. Parametrizarea datelor. Sabloane in C++. Clase generice
10. Parametrizarea metodelor (polimorfism). Functii virtuale in C++. Clase abstracte
11. Controlul tipului in timpul rularii programului in C++
12. Tratarea exceptiilor in C++
13. Recapitulare, concluzii
14. Tratarea subiectelor de examen

Bibliografie

- Herbert Schildt. C++ manual complet. Ed.Teora, Bucuresti, 1997 (si urmatoarele).
- Bruce Eckel. Thinking in C++ (2nd edition). Volume 1: Introduction to Standard C++. Prentice Hall, 2000. (cartea se poate descarca in format electronic, gratuit si legal de la adresa <http://mindview.net/Books/TICPP/ThinkingInCPP2e.html>)

Bibliografie avansata

- Bruce Eckel, Chuck Allison. Thinking in C++ (2nd edition). Volume 2: Practical Programming. Prentice Hall, 2003. (cartea se poate descarca in format electronic, gratuit si legal de la adresa <http://mindview.net/Books/TICPP/ThinkingInCPP2e.html>)

Regulament de desfasurare si notare

- Disciplina **Proiectare si programare orientata pe obiecte** pentru anul I Informatica este organizata in curs si laborator, fiecare dintre aceste activitati avand alocate 2 ore pe saptamana, precum si seminar cu 1 ora pe saptamana.
- Disciplina este programata in semestrul II, avand o durata de desfasurare de 14 saptamani.
- Materia este de nivel elementar mediu si se bazeaza pe cunostintele dobandite la cursul de **Programare procedurala** din semestrul I.
- Limbajul de programare folosit la curs si la laborator este **C++**.
- Programa disciplinei este impartita in 14 cursuri.
- Evaluarea studentilor se face cumulativ prin:
 - 3 lucrari practice (proiecte)
 - Test practic
 - Test scris
- **Toate cele 3 probe de evaluare sunt obligatorii.**
- Condițiile de promovare enuntate mai sus se pastreaza la oricare din examenele restante ulterioare aferente acestui curs.

Regulament de desfasurare si notare (2)

- Cele 3 lucrari practice se realizeaza si se noteaza in cadrul laboratorului, dupa urmatorul program:
 - **Saptamana 1:** Test de evaluare a nivelului de intrare.
 - **Saptamana 2:** Atribuirea temelor pentru LP1.
 - **Saptamana 3:** Consultatii pentru LP1.
 - **Saptamana 4:** Consultatii pentru LP1. **Termen predare LP1: TBA.**
 - **Saptamana 5:** Evaluarea LP1.
 - **Saptamana 6:** Atribuirea temelor pentru LP2.
 - **Saptamana 7:** Consultatii pentru LP2.
 - **Saptamana 8:** Predarea LP2. **Termen predare LP2: TBA.**
 - **Saptamana 9:** Evaluarea LP2.
 - **Saptamana 10:** Atribuirea temelor pentru LP3.
 - **Saptamana 11:** Consultatii pentru LP3.
 - **Saptamana 12:** Consultatii pentru LP3. **Termen predare LP3: TBA.**
 - **Saptamana 13:** Evaluarea LP3.
 - **Saptamana 14:** Test practic de laborator.
- Prezenta la laborator in saptamanile 2, 5, 6, 9, 10, 13, 14 pentru atribuirea si evaluarea lucrarilor practice si pentru sustinerea testului practic este obligatorie.

Regulament de desfasurare si notare (3)

- Consultatiile de laborator se desfasoara pe baza intrebarilor studentilor. Prezenta la laborator in saptamanile 1, 3, 4, 7, 8, 11, 12 pentru consultatii este recomandata, dar facultativa.
- Continutul lucrarilor practice va urmari materia predata la curs. Lucrarile practice se realizeaza individual. Notarea fiecărei lucrări practice se va face cu note de la 1 la 10.
- Atribuirea temelor pentru lucrarile practice se face prin prezentarea la laborator in saptamana precizata mai sus sau in oricare din urmatoarele 2 saptamani. Indiferent de data la care un student se prezinta pentru a primi tema pentru una dintre lucrarile practice, termenul de predare a acesteia ramane cel precizat in regulament. In consecinta, tema pentru o lucrare practica nu mai poate fi preluata dupa expirarea termenului ei de predare.
- Predarea lucrarilor practice se face atat prin email cat si prin MOODLE, la adresa indicata de tutorele de laborator, inainte de termenele limita de predare, indicate mai sus pentru fiecare LP. Dupa expirarea termenelor respective, lucrarea practica se mai poate trimite prin email pentru o perioada de gratie de 2 zile (48 de ore). Pentru fiecare zi partiala de intarziere se vor scadea 2 puncte din nota atribuita pe lucrare. Dupa expirarea termenului de gratie, lucrarea nu va mai fi acceptata si va fi notata cu 1.
- Pentru activitatea din timpul semestrului, se va atribui o nota calculata ca medie aritmetica a celor mai bune 2 din cele 3 note obtinute pe lucrarile practice. Pentru evidentierea unor lucrari practice, tutorele de laborator poate acorda un bonus de pana la 2 puncte la nota pe proiecte astfel calculata. **Studentii care nu obtin cel putin nota 5 pentru activitatea pe proiecte nu pot intra in examen si vor trebui sa refaca aceasta activitate, inainte de prezentarea la restanta.**

Regulament de desfasurare si notare (4)

- Testul practic se va sustine in saptamana 14, va consta dintr-un program care trebuie realizat individual intr-un timp limitat (90 de minute) si va avea un nivel elementar. Notarea testului practic se va face cu o nota de la 1 la 10, conform unui barem anuntat odata cu cerintele. Testul practic este obligatoriu. **Studentii care nu obtin cel putin nota 5 la testul practic de laborator nu pot intra in examen si vor trebui sa il dea din nou, inainte de prezentarea la restanta.**
- Testul scris se va sustine in sesiunea de examene si consta dintr-un set de 18 intrebari (6 intrebari de teorie si 12 intrebari practice). Notarea testului scris se va face cu o nota de la 1 la 10 (1 punct din oficiu si cate 0,5 puncte pentru fiecare raspuns corect la cele 18 intrebari). **Studentii nu pot lua examenul decat daca obtin cel putin nota 5 la testul scris.**
- Examenul se considera luat daca studentul respectiv a obtinut cel putin nota 5 la fiecare dintre cele 3 evaluari (activitatea practica din timpul semestrului, testul practic de laborator si testul scris). In aceasta situatie, nota finala a fiecarui student se calculeaza ca medie ponderata intre notele obtinute la cele 3 evaluari, ponderile cu care cele 3 note intra in medie fiind:
 - 25% - nota pe lucrarile practice (proiecte)
 - 25% - nota la testul practic
 - 50% - nota la testul scris

Statistici 2009-2010

	Numar studenti	Procent din total studenti	Procent din total prezenti
Total	Aprox. 200		
Prezenti	157	Aprox. 70 %	
Absenti	Aprox. 40	Aprox. 30 %	
Trecuti	83	40 %	53 %
Picati	74	37 %	47 %
Nota 10	2	1 %	1.30 %
Nota 9	20	10 %	12.75 %
Nota 8	26	13 %	16.56 %
Nota 7	24	12 %	15.29 %
Nota 6	6	3 %	3.82 %
Nota 5	0	0 %	0 %
Nota 4	25	12.5 %	15.92 %
Nota 3	54	27 %	34.40 %

Anul universitar 2008-2009

Sesiunea I

	Numar studenti	Procent din total studenti	Procent din total prezenti
Total	217		
Prezent i	155	71 %	
Absenti	62	29 %	
Trecuti	89	41 %	57 %
Picati	66	30 %	43 %
Nota 10	3	1 %	2 %
Nota 9	13	6 %	8 %
Nota 8	22	10 %	14 %
Nota 7	32	15 %	21 %
Nota 6	16	8 %	10 %
Nota 5	3	1 %	2 %
Nota 4	24	11 %	16 %
Nota 3	42	19 %	27 %

Anul universitar 2007-2008

	Sesiunea I			Sesiunea I + II		
	Numar studenti	Procent din total studenti	Procent din total prezenti	Numar studenti	Procent din total studenti	Procent din total prezenti
Total	261			261		
Prezenti	195	75%		205	78%	
Absenti	66	25%		56	22%	
Trecuti	129	50%	66%	152	58%	74%
Picati	66	25%	34%	53	20%	26%
Nota 10	5	2%	3%	5	2%	2%
Nota 9	17	7%	9%	19	7%	9%
Nota 8	42	15%	21%	46	18%	23%
Nota 7	52	20%	26%	61	22%	29%
Nota 6	12	5%	6%	20	8%	10%
Nota 5	1	1%	1%	1	1%	1%
Nota 4	41	15%	21%	18	7%	9%
Nota 3	25	10%	13%	35	13%	17%

modificari

- Laborator: notare mai “clara”
- Seminar: 0.5 bonus pentru max. 20% din studenti
- Laborator: implementare a unui proiect bonus de dimensiuni medii, pe fiecare subgrupa se va face un clasament si primul si al doilea proiect de acest fel primesc 1 punct bonus la examen
- Cel mai bun proiect pentru serie primeste 2 puncte bonus la examen
- 5 cursuri prezenta: 0.5 puncte bonus la examen
daca aveti prezenta la TOATE cele 5 cursuri
- bonusuri dupa ce se promoveaza examenul scris

Principiile programarii orientate pe obiecte

- ce este programarea
- definirea programatorului:
 - rezolva problema
- definirea informaticianului:
 - rezolva problema **bine**

rezolvarea “mai bine” a unei probleme

- “bine” depinde de caz
 - drivere: cat mai repede (asamblare)
 - jocuri de celulare: memorie mica
 - rachete, medicale: erori duc la pierderi de vieti
- programarea OO: cod mai corect
 - Microsoft: nu conteaza erorile minore, conteaza data lansarii

Problema Rachetelor Patriot

- In 15 Februarie 1991 un sistem de rachete Patriot care opera in Dhahran, Arabia Saudita, in timpul Operatiunii Desert Storm nu a interceptat corect o racheta Scud. Aceasta racheta Scud a reusit sa detoneze intr-o zona militara omorand 28 soldati americani. [GAO]



23

http://news.bbc.co.uk/1/shared/spl/hi/middle_east/03/v3_iraq_timeline/html/scuds.stm

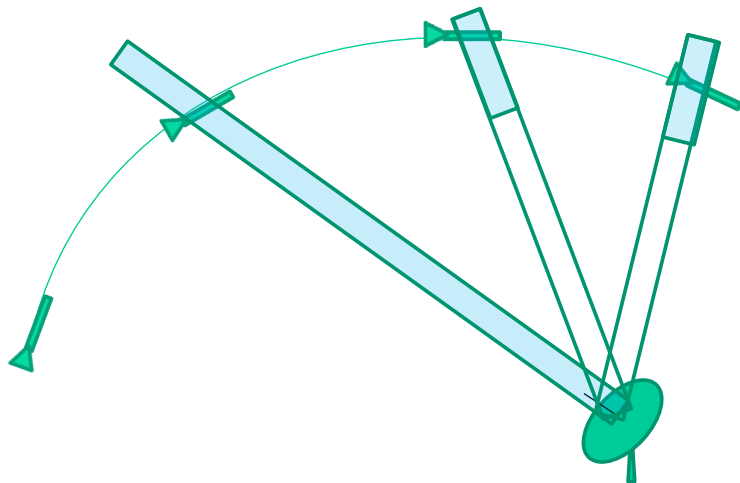
Patriot: probleme cu software-ul

- o problema software in calculatorul care controla armele a dus la calcularea inexacta a traiectoriei rachetelor inamice, problema ce s-a inrautatit cu cat mai mult cu cat sistemul a fost tinut in opratiune
- La momentul incidentului bateria de rachete opera in mod continuu de peste 100 ore. Din aceasta cauza inacuratetea era atat de serioasa incat sistemul se uita in zona gresita pentru racheta SCUD.

24

Tracking a missile: ce ar trebui sa se intample

- Search: scanarea prin radar la distanta mare cand racheta este detectata,
range gate se calculeaza noua zona de scanare pentru racheta
- Validation, Tracking: doar zona calculata anterior este scanata



25

25

Greseala la design-ul software

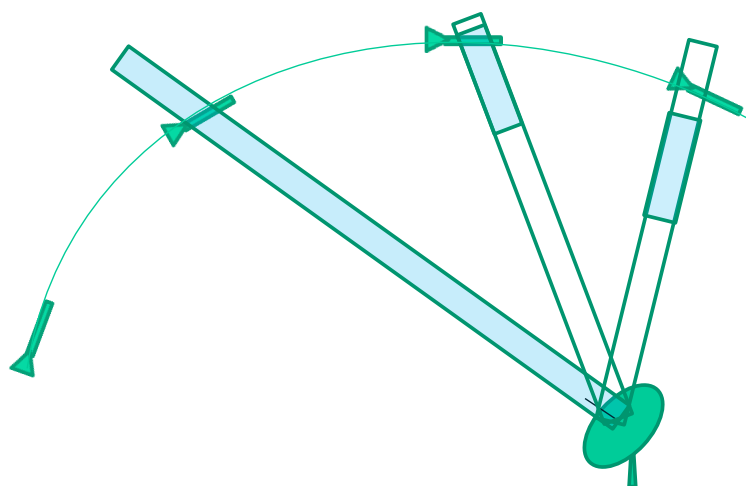
- Range gate este calculata/prezisa bazat pe
 - timpul ultimei detectii prin Radar: intreg, masurat in milisecunde
 - Viteza cunoscuta a rachetei: valoare floating-point
- Problema:
 - Range gate a folosit registrii pe 24 biti si fiecare 0,1 incrementare in secunde adauga o mica eroare
 - De-a lungul timpului aceasta eroare a devenit serioasa incat range gate nu mai calcula pozitia rachetei

26

26

Ce s-a intamplat

- Zona calculata de range gate a fost shiftata si nu a mai gasit racheta



27

27

Sursele problemei:

- Rachetele Patriot au fost concepute pentru a contracara rachetele incete (Mach 2), nu Scud-uri (Mach 5)
 - Calibrarea aparatului/softului nu a fost facuta - din cauza/frica unui singur fapt: adaugarea unei componente externe ar destabiliza sistemul (!)
- Sistemul de rachete Patriot este de obicei folosit in intervale scurte - nu mai mult de 8 ore
 - Concepute ca sisteme mobile: quick on/off, pentru a nu fi detectate

28

28

Pierderea lui Ariane 5

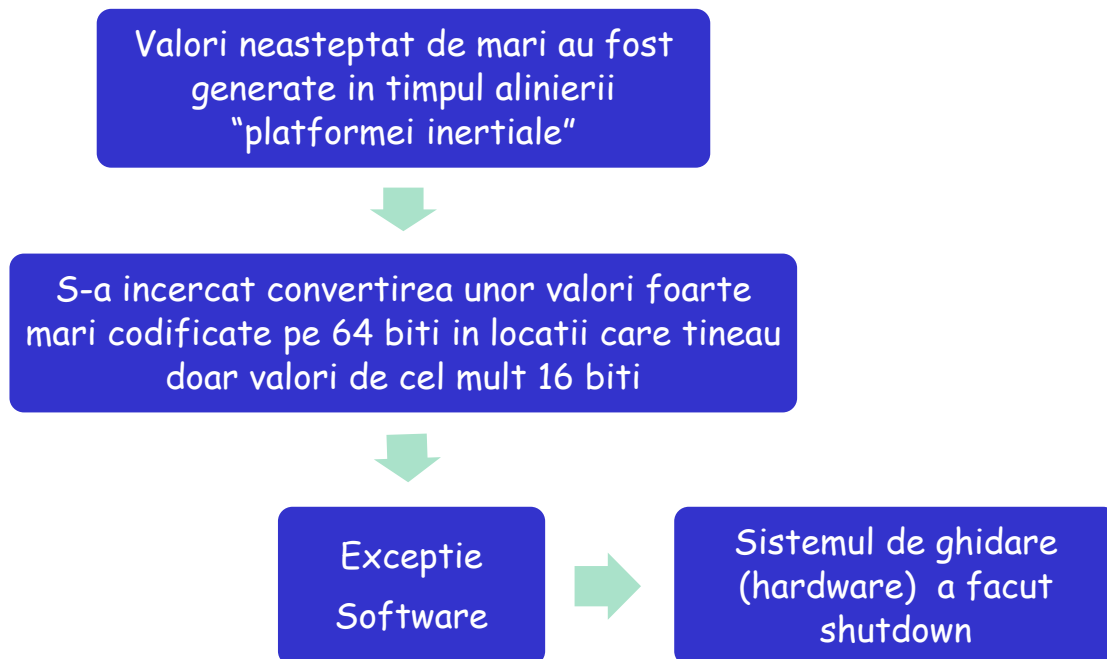
- Pe 4 Iunie 1996 zborul initial "maiden flight" a lansatorului de sateliti Ariane 5 a fost distrus. La numai 40 sec de la initierea zborului, altitudine de aprox 3700m, lansatorul a iesit din traseul planificat, s-a rupt si a explodat.



29

29

Ariane 5: O problema software



30

30

Sursa problemei

- Codul de aliniere a rachetei a fost reutilizat de la rachetele mai mici si mai putin puternice Ariane 4
 - Valorile vitezelor lui Ariane 5 au fost in afara parametrilor conceputi pentru Ariane 4
- In mod ironic alinierea rachetei nu mai era necesara dupa decolare!
- De ce rula totusi acel cod?
 - Inginerii au decis sa il lase sa ruleze pentru inca 40 secunde de la momentul de decolare planificat
 - Pentru a permite un restart rapid pentru cazul in care lansarea rachetei ar fi fost amanata pentru cateva minute

Accidente de la Panama Cancer Institute (Gage & McCormick, 2004)

- Noiembrie 2000: 27 pacienti de cancer primesc doze masive de radiatie
 - Din cauza (partial) unor neajunsuri in Multidata software
 - Doctorii care foloseau softul au fost gasiti vinovati de crima in Panama
 - Nota: In anii 1980 se cunoaste un alt incident "Therac-25" cand probleme software au dus la doze masive de radiatie fiind administrate la pacienti

Softul Multidata

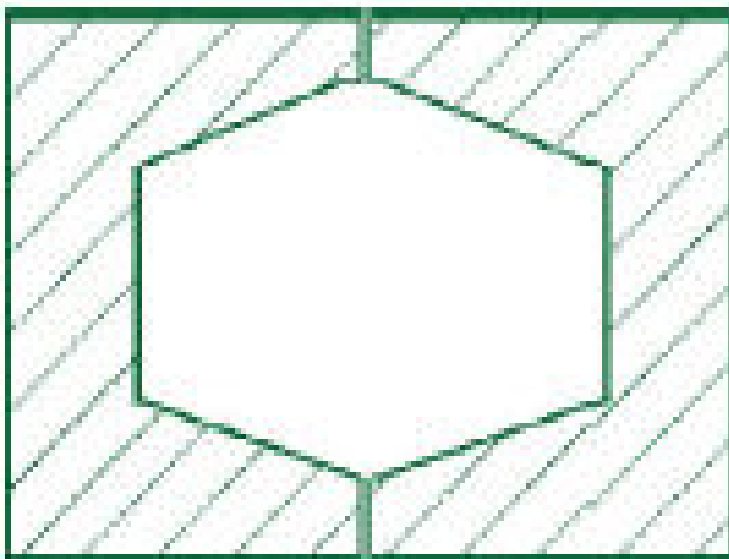
- Folosit pentru planificarea tratamentului de radiatie
 - Operatorul introduce datele pacientului
 - Operatorul indica amplasarea unor "blocuri" (scuturi metalice folosite pentru a proteja zonele sensibile ale pacientului) prin folosirea unui editor grafic
 - Softul da apoi o predictie 3D a zonelor unde radiatia ar trebui administrata
 - urmand aceste date se determina dozajul

33

33

Editorul de plasare a blocurilor

NRC Information Notice 2001-08, Supp. 2



- blocurile sunt desenate ca poligoane separate (in poza sunt descrise doua blocuri)
- Limitare software: cel mult 4 blocuri
- Ce facem daca doctorii vor sa foloseasca mai multe blocuri?

34

34

o "solutie"

NRC Information Notice 2001-08, Supp. 2

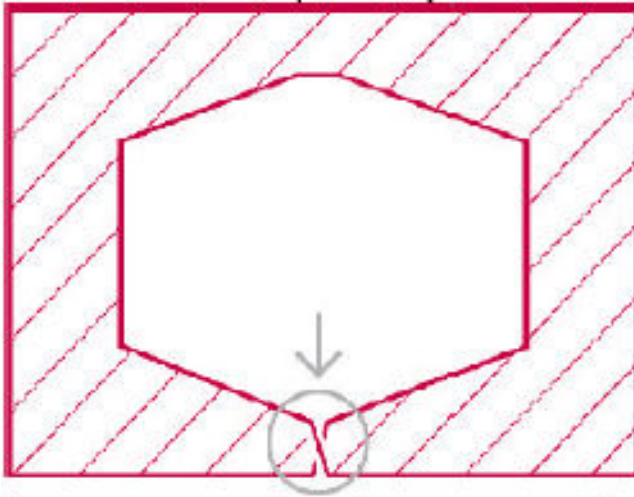


Fig. 1: Incorrect crossing block outline

- Nota: aceasta este o linie neintrerupta...
- Softul o trateaza ca pe un singur bloc
- Acum se pot folosi mai multe blocuri!

35

35

Problema majora

- Algoritmul de prezicere a dozajului este bazat pe blocuri in forma unui poligon dar in editorul grafic se puteau genera ne-poligoane.
- Cand rula pe blocuri "speciale" algoritmul daea predictii gresite care conduceau la dozaje extrem de mari

36

36

Principiile programarii orientate pe obiecte

- Obiecte
- Clase
- Mostenire
- Ascunderea informatiei
- Polimorfism
- Sabloane

Obiecte

- au stare si actiuni (metode/functii)
- au interfata (actiuni) si o parte ascunsa (starea)
- Sunt grupate in clase, obiecte cu aceleasi proprietati

Clase

- mentioneaza proprietatile generale ale obiectelor din clasa respectiva
- clasele nu se pot “rula”
- folositoare la encapsulare (ascunderea informatiei)
- reutilizare de cod: mostenire

Mostenire

- multe obiecte au proprietati similare
- reutilizare de cod

Ascunderea informatiei

- foarte importanta
- public, protected, private

Avem acces?	public	protected	private
Aceeasi clasa	da	da	da
Clase derivate	da	da	nu
Alte clase	da	nu	nu

Polimorfism

- tot pentru claritate/ cod mai sigur

Sabloane

- din nou cod mai sigur/reutilizare de cod
- putem implementa lista inlantuita de
 - intregi
 - caractere
 - float
 - obiecte

Sa ne cunoastem mai bine

- Raspundeti la urmatoarele intrebari pentru 0.05 din nota de la examen;
nume si grupa scris citet
- 1. Care e diferenta dintre obiecte si clasa
- 2. Ce este polimorfismul si dati un exemplu de polimorfism.
- 3. Care sunt avantajele programarii orientate pe obiecte. Care sunt dezavantajele POO
- 4. Descrieti ce face un constructor si un destructor.
- 5. Ce sunt functiile virtuale si la ce se folosesc

Cursul de programare orientata pe obiecte

Seria 13

Saptamana 2, 25 feb 2014

Andrei Paun

Cuprinsul cursului: 25 feb 2014

- Generalitati despre curs
- Reguli de comportament
- Generalitati despre OOP
- Sa ne cunoastem mai bine... rezultate

Principiile programarii orientate pe obiecte

- Obiecte
- Clase
- Mostenire
- Ascunderea informatiei
- Polimorfism
- Sabloane

Obiecte

- au stare si actiuni (metode/functii)
- au interfata (actiuni) si o parte ascunsa (starea)
- Sunt grupate in clase, obiecte cu aceleasi proprietati

Clase

- mentioneaza proprietatile generale ale obiectelor din clasa respectiva
- clasele nu se pot “rula”
- folosite la encapsulare (ascunderea informatiei)
- reutilizare de cod: mostenire

Mostenire

- multe obiecte au proprietati similare
- reutilizare de cod

Ascunderea informatiei

- foarte importanta
- public, protected, private

Avem acces?	public	protected	private
Aceeasi clasa	da	da	da
Clase derivate	da	da	nu
Alte clase	da	nu	nu

Polimorfism

- tot pentru claritate/ cod mai sigur
- Polimorfism la compilare: ex. max(int), max(float)
- Polimorfism la executie: RTTI

Sabloane

- din nou cod mai sigur/reutilizare de cod
- putem implementa lista inlantuita de
 - intregi
 - caractere
 - float
 - obiecte

Privire de ansamblu pentru C++

- Bjarne Stroustrup in 1979 la Bell Laboratories in Murray Hill, New Jersey
- 5 revizii: 1985, 1990, 1998 ANSI+ISO, 2003 (corrigendum), 2011 (C++11)
- Versiunea 1998: Standard C++

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    cout << "This is output.\n"; // this is a single line comment
    /* you can still use C style comments */
    // input a number using >>
    cout << "Enter a number: ";
    cin >> i;
    // now, output a number using <<
    cout << i << " squared is " << i*i << "\n";
    return 0;
}
```

Diferente cu C

- <iostream> (fara .h)
- int main() (fara void)
- using namespace std;
- cout, cin (fara &)
- // comentarii pe o linie
- declarare variabile

```
#include <iostream>
using namespace std;
int main( )
{
    float f;
    char str[80];
    double d;
    cout << "Enter two floating point numbers: ";
    cin >> f >> d;
    cout << "Enter a string: ";
    cin >> str;
    cout << f << " " << d << " " << str;
    return 0;
}
```

- citirea string-urilor se face pana la primul caracter alb
- se poate face afisare folosind toate caracterele speciale \n, \t, etc.

Variabile locale

```
/* Incorrect in C89. OK in C++. */  
int f()  
{  
    int i;  
    i = 10;  
    int j; /* won't compile as a C program */  
    j = i*2;  
    return j;  
}
```

```
#include <iostream>  
using namespace std;  
int main()  
{  
    float f;  
    double d;  
    cout << "Enter two floating point numbers: ";  
    cin >> f >> d;  
    cout << "Enter a string: ";  
    char str[80]; // str declared here, just before 1st use  
    cin >> str;  
    cout << f << " " << d << " " << str;  
    return 0;  
}
```

C++ vechi vs C++ nou

- fara conversie automata la int

<pre>func(int i) { return i*i; }</pre>	<pre>int func(int i) { return i*i; }</pre>
--	--

- nou tip de include
- using namespace

Tipul de date bool

- se definesc true si false (1 si 0)
- C99 nu il defineste ca bool ci ca _Bool (fara true/false)
- <stdbool.h> pentru compatibilitate

2 versiuni de C++; diferite: Noile include

- `<iostream>` `<fstream>` `<vector>` `<string>`
- **`math.h` este `<cmath>`**
- **`string.h` este `<cstring>`**
- **`math.h` deprecated, folositi `cmath`**

Clasele in C++

```
#define SIZE 100
// This creates the class stack.
class stack {
    int stck[SIZE];
    int tos;
public:
    void init();
    void push(int i);
    int pop();
};
```

- `init()`, `push()`, `pop()` sunt functii membru
- `stck`, `tos`: variabile membru

- se creeaza un tip nou de date `stack mystack;`
- un obiect instantiaza clasa
- functiile membru sunt date prin semnatura
- pentru definirea fiecărei functii se foloseste `::`

```
void stack::push(int i)
{
    if(tos==SIZE) {
        cout << "Stack is full.\n";
        return;
    }
    stck[tos] = i;
    tos++;
}
```

- `::` scope resolution operator
- si alte clase pot folosi numele `push()` si `pop()`
- dupa instantiere, pentru apelul `push()`

`stack mystack;`

- `mystack.push(5)`
- programul complet in continuare

```

#include <iostream>
using namespace std;
#define SIZE 100
// This creates the class stack.
class stack {
    int stck[SIZE];
    int tos;
public:
    void init();
    void push(int i);
    int pop();
};

void stack::init()
{
    tos = 0;
}

void stack::push(int i)
{
    if(tos==SIZE) {
        cout << "Stack is full.\n";
        return;
    }
    stck[tos] = i;
    tos++;
}

int stack::pop()
{
    if(tos==0) {
        cout << "Stack underflow.\n";
        return 0;
    }
    tos--;
    return stck[tos];
}

int main()
{
    stack stack1, stack2; // create two stack objects
    stack1.init();
    stack2.init();
    stack1.push(1);
    stack2.push(2);
    stack1.push(3);
    stack2.push(4);
    cout << stack1.pop() << " ";
    cout << stack1.pop() << " ";
    cout << stack2.pop() << " ";
    cout << stack2.pop() << "\n";
    return 0;
}

```

Encapsulare

- urmatorul cod nu poate fi folosit in main()

stack1.tos = 0; // Error, tos is private.

Overloading de functii

- polimorfism
- simplitate/corectitudine de cod

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

// abs is overloaded three ways
int abs(int i);
double abs(double d);
long abs(long l);

int main()
{
    cout << abs(-10) << "\n";
    cout << abs(-11.0) << "\n";
    cout << abs(-9L) << "\n";
    return 0;
}

int abs(int i)
{
    cout << "Using integer abs()\n";
    return i<0 ? -i : i;
}
```

```
double abs(double d)
{
    cout << "Using double abs()\n";
    return d<0.0 ? -d : d;
}

long abs(long l)
{
    cout << "Using long abs()\n";
    return l<0 ? -l : l;
}
```

```
Using integer abs()
10
Using double abs()
11
Using long abs()
9
```

overload de functii

- nume acelasi
- diferenta e in tipurile de parametrii
- tipul de intoarcere nu e suficient pentru diferenta
- se poate folosi si pentru functii complet diferite (nerecomandat)
- overload de operatori: mai tarziu

Mostenirea

- incorporarea componentelor unei clase in alta
- refolosire de cod
- detalii mai subtile pentru tipuri si subtipuri
- clasa de baza, clasa derivata
- clasa derivata contine toate elementele clasei de baza, mai adauga noi elemente

```

class building {
    int rooms;
    int floors;
    int area;
public:
    void set_rooms(int num);
    int get_rooms();
    void set_floors(int num);
    int get_floors();
    void set_area(int num);
    int get_area();
};

```

// house is derived from building

```

class house : public building {
    int bedrooms;
    int baths;
public:
    void set_bedrooms(int num);
    int get_bedrooms();
    void set_baths(int num);
    int get_baths();
};

```

tip acces: public, private, protected
mai multe mai tarziu

public: membrii publici ai building
devin publici pentru house

- house NU are acces la membrii privati ai lui building
- asa se realizeaza encapsularea
- clasa derivata are acces la membrii publici ai clasei de baza si la toti membrii sai (publici si privati)

```

#include <iostream>
using namespace std;

class building {
    int rooms;
    int floors;
    int area;
public:
    void set_rooms(int num);
    int get_rooms();
    void set_floors(int num);
    int get_floors();
    void set_area(int num);
    int get_area();
};

// house is derived from building
class house : public building {
    int bedrooms;
    int baths;
public:
    void set_bedrooms(int num);
    int get_bedrooms();
    void set_baths(int num);
    int get_baths();
};

// school is also derived from building
class school : public building {
    int classrooms;
    int offices;
public:
    void set_classrooms(int num);
    int get_classrooms();
    void set_offices(int num);
    int get_offices();
};

```

```

void building::set_rooms(int num)
{ rooms = num; }
void building::set_floors(int num)
{ floors = num; }
void building::set_area(int num)
{ area = num; }
int building::get_rooms()
{ return rooms; }
int building::get_floors()
{ return floors; }
int building::get_area()
{ return area; }
void house::set_bedrooms(int num)
{ bedrooms = num; }
void house::set_baths(int num)
{ baths = num; }
int house::get_bedrooms()
{ return bedrooms; }
int house::get_baths()
{ return baths; }
void school::set_classrooms(int num)
{ classrooms = num; }
void school::set_offices(int num)
{ offices = num; }
int school::get_classrooms()
{ return classrooms; }
int school::get_offices()
{ return offices; }

int main()
{
    house h;
    school s;
    h.set_rooms(12);
    h.set_floors(3);
    h.set_area(4500);
    h.set_bedrooms(5);
    h.set_baths(3);
    cout << "house has " << h.get_bedrooms();
    cout << " bedrooms\n";
    s.set_rooms(200);
    s.set_classrooms(180);
    s.set_offices(5);
    s.set_area(25000);
    cout << "school has " << s.get_classrooms();
    cout << " classrooms\n";
    cout << "Its area is " << s.get_area();
    return 0;
}

```

```

house has 5 bedrooms
school has 180 classrooms
Its area is 25000

```

Mostenire

- terminologie
 - clasa de baza, clasa derivata
 - superclasa subclasa
 - parinte, fiu
- mai tarziu: functii virtuale, identificare de tipuri in timpul rularii (RTTI)

Constructori/Destructor

- initializare automata
- obiectele nu sunt statice
- constructor: functie speciala, numele clasei
- constructorii nu pot intoarce valori (nu au tip de intoarcere)

```

// This creates the class stack.
class stack {
    int stck[SIZE];
    int tos;
public:
    stack(); // constructor
    void push(int i);
    int pop();
};

// stack's constructor
stack::stack()
{
    tos = 0;
    cout << "Stack Initialized\n";
}

```

- constructorii/destructorii sunt chemati de fiecare data o variabila de acel tip este creata/distrusa. Declaratii active nu pasive.
- Destructorii: reversul, executa operatii cand obiectul nu mai este folositor
- memory leak

stack::~~stack()


```
// This creates the class stack.
class stack {
    int stck[SIZE];
    int tos;
public:
    stack(); // constructor
    ~stack(); // destructor
    void push(int i);
    int pop();
};

// stack's constructor
stack::stack()
{
    tos = 0;
    cout << "Stack Initialized\n";
}

// stack's destructor
stack::~~stack()
{
    cout << "Stack Destroyed\n";
}
```

```
// Using a constructor and destructor.
#include <iostream>
using namespace std;
#define SIZE 100
```

```
// This creates the class stack.
```

```
class stack {
    int stck[SIZE];
    int tos;
public:
    stack(); // constructor
    ~stack(); // destructor
    void push(int i);
    int pop();
};
```

```
// stack's constructor
```

```
stack::stack()
{
    tos = 0;
    cout << "Stack Initialized\n";
}
```

```
// stack's destructor
```

```
stack::~~stack()
{
    cout << "Stack Destroyed\n";
}
```

```
void stack::push(int i){
    if(tos==SIZE) {
        cout << "Stack is full.\n";
        return;
    }
    stck[tos] = i;
    tos++;}

int stack::pop(){
    if(tos==0) {
        cout << "Stack underflow.\n";
        return 0;
    }
    tos--;
    return stck[tos];}
```

```
int main(){
    stack a, b; // create two stack objects
    a.push(1);
    b.push(2);
    a.push(3);
    b.push(4);
    cout << a.pop() << " ";
    cout << a.pop() << " ";
    cout << b.pop() << " ";
    cout << b.pop() << "\n";
    return 0;
}
```

Stack Initialized

Stack Initialized

3 1 4 2

Stack Destroyed

Stack Destroyed

Clasele in C++

- cu “class”
- obiectele instantiaza clase
- similare cu struct-uri si union-uri
- au functii
- specificatorii de acces: public, private, protected
- default: private
- protected: pentru mostenire, vorbim mai tarziu

```
class nume_clasa {  
    private variabile si functii membru  
    specificator_de_acces:  
        variabile si functii membru  
    specificator_de_acces:  
        variabile si functii membru  
    // ...  
    specificator_de_acces:  
        variabile si functii membru  
} lista_obiecte;
```

- putem trece de la public la private si iar la public, etc.

```
class employee {  
    char name[80]; // private by default  
public:  
    void putname(char *n); // these are public  
    void getname(char *n);  
private:  
    double wage; // now, private again  
public:  
    void putwage(double w); // back to public  
    double getwage();  
};
```

```
class employee {  
    char name[80];  
    double wage;  
public:  
    void putname(char *n);  
    void getname(char *n);  
    void putwage(double w);  
    double getwage();  
};
```

- se foloseste mai mult a doua varianta
- un membru (ne-static) al clasei nu poate avea initializare
- nu putem avea ca membri obiecte de tipul clasei (putem avea pointeri la tipul clasei)
- nu auto, extern, register

Cursul de programare orientata pe obiecte

Seria 14

Saptamana 3, 5 martie 2013

Andrei Paun

Cuprinsul cursului: 5 martie 2013

- Struct, Union si clase
- functii prieten
- Constructori/destructori
- supraincarcarea functiilor in C++
- supraincarcarea operatorilor in C++

Clasele in C++

- cu “class”
- obiectele instantiaza clase
- similare cu struct-uri si union-uri
- au functii
- specificatorii de acces: public, private, protected
- default: private
- protected: pentru mostenire, vorbim mai tarziu

```

class class-name {
    private data and functions
access-specifier:
    data and functions
access-specifier:
    data and functions
    // ...
access-specifier:
    data and functions
} object-list;

```

- putem trece de la public la private si iar la public, etc.

```

class employee {
    char name[80]; // private by default
public:
    void putname(char *n); // these are public
    void getname(char *n);
private:
    double wage; // now, private again
public:
    void putwage(double w); // back to public
    double getwage();
};

```

```

class employee {
    char name[80];
    double wage;
public:
    void putname(char *n);
    void getname(char *n);
    void putwage(double w);
    double getwage();
};

```

- se foloseste mai mult a doua varianta
- un membru (ne-static) al clasei nu poate avea initializare
- nu putem avea ca membri obiecte de tipul clasei (putem avea pointeri la tipul clasei)
- nu auto, extern, register

- variabilele de instanta (instance variables)
- membri de tip date ai clasei
 - in general private
 - pentru viteza se pot folosi “public”

Exemplu

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

class myclass {
public:
    int i, j, k; // accessible to entire program
};

int main()
{
    myclass a, b;
    a.i = 100; // access to i, j, and k is OK
    a.j = 4;
    a.k = a.i * a.j;
    b.k = 12; // remember, a.k and b.k are different
    cout << a.k << " " << b.k;
    return 0;
}
```

Struct si class

- singura diferenta: struct are default membri ca public iar class ca private
- struct defineste o clasa (tip de date)
- putem avea in struct si functii
- pentru compatibilitate cu cod vechi
- extensibilitate
- a nu se folosi struct pentru clase

```
// Using a structure to define a class.
```

```
#include <iostream>
```

```
#include <cstring>
```

```
using namespace std;
```

```
struct mystr {
```

```
    void buildstr(char *s); // public
```

```
    void showstr();
```

```
private: // now go private
```

```
    char str[255];
```

```
};
```

```
void mystr::buildstr(char *s){
```

```
    if(!*s) *str = '\0'; // initialize string
```

```
    else strcat(str, s);}
```

```
void mystr::showstr() {cout << str << "\n"; }
```

```
int main() {
```

```
    mystr s;
```

```
    s.buildstr(""); // init
```

```
    s.buildstr("Hello ");
```

```
    s.buildstr("there!");
```

```
    s.showstr();
```

```
    return 0;
```

```
}
```

```
class mystr {
```

```
    char str[255];
```

```
public:
```

```
    void buildstr(char *s); // public
```

```
    void showstr();
```

```
};
```

union si class

- la fel ca struct
- toate elementele de tip data folosesc aceeași locație de memorie
- membrii sunt publici (by default)

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

union swap_byte {
    void swap();
    void set_byte(unsigned short i);
    void show_word();
    unsigned short u;
    unsigned char c[2];
};

void swap_byte::swap()
{
    unsigned char t;
    t = c[0];
    c[0] = c[1];
    c[1] = t;
}

void swap_byte::show_word()
{
    cout << u;
}

void swap_byte::set_byte(unsigned short i)
{
    u = i;
}
```

```
int main()
{
    swap_byte b;
    b.set_byte(49034);
    b.swap();
    b.show_word();
    return 0;
}
```

35519

union ca si class

- union nu poate mosteni
- nu se poate mosteni din union
- nu poate avea functii virtuale (nu avem mostenire)
- nu avem variabile de instanta statice
- nu avem referinte in union
- nu avem obiecte care fac overload pe =
- obiecte cu (con/de)structor definiti nu pot fi membri in union

union anonime

- nu au nume pentru tip
- nu se pot declara obiecte de tipul respectiv
- folosite pentru a spune compilatorului cum se tin variabilele respective in memorie
 - folosesc aceeasi locatie de memorie
- variabilele din union sunt accesibile ca si cum ar fi declarate in blocul respectiv

```

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

int main()
{
    // define anonymous union
    union {
        long l;
        double d;
        char s[4];
    };

    // now, reference union elements directly
    l = 100000;
    cout << l << " ";
    d = 123.2342;
    cout << d << " ";
    strcpy(s, "hi");
    cout << s;
    return 0;
}

```

union anonime

- nu poate avea functii
- nu poate avea private sau protected (fara functii nu avem acces la altceva)
- union-uri anonime globale trebuie precizate ca static

functii prieten

- cuvantul: friend
- pentru accesarea campurilor protected, private din alta clasa
- folosite la overload-area operatorilor, pentru unele functii de I/O, si portiuni interconectate (exemplu urmeaza)
- in rest nu se prea folosesc

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

class myclass {
    int a, b;
public:
    friend int sum(myclass x);
    void set_ab(int i, int j);
};

void myclass::set_ab(int i, int j) { a = i; b = j; }

// Note: sum() is not a member function of any class.
int sum(myclass x)
{
    /* Because sum() is a friend of myclass, it can directly access a and b. */
    return x.a + x.b;
}

int main()
{
    myclass n;
    n.set_ab(3, 4);
    cout << sum(n);
    return 0;
}
```

```

#include <iostream>
using namespace std;

const int IDLE = 0;
const int INUSE = 1;
class C2; // forward declaration

class C1 {
    int status; // IDLE if off, INUSE if on screen
    // ...
public:
    void set_status(int state);
    friend int idle(C1 a, C2 b);
};

class C2 {
    int status; // IDLE if off, INUSE if on screen
    // ...
public:
    void set_status(int state);
    friend int idle(C1 a, C2 b);
};

void C1::set_status(int state)
{ status = state; }

void C2::set_status(int state)
{ status = state; }

int idle(C1 a, C2 b)
{
    if(a.status || b.status) return 0;
    else return 1;
}

int main()
{
    C1 x;
    C2 y;
    x.set_status(IDLE);
    y.set_status(IDLE);
    if(idle(x, y)) cout << "Screen can be used.\n";
    else cout << "In use.\n";
    x.set_status(INUSE);
    if(idle(x, y)) cout << "Screen can be used.\n";
    else cout << "In use.\n";
    return 0;
}

```

- functii prieten din alte obiecte

```

#include <iostream>
using namespace std;
const int IDLE = 0;
const int INUSE = 1;

class C2; // forward declaration

class C1 {
    int status; // IDLE if off, INUSE if on screen
// ...
public:
    void set_status(int state);
    int idle(C2 b); // now a member of C1
};

class C2 {
    int status; // IDLE if off, INUSE if on screen
// ...
public:
    void set_status(int state);
    friend int C1::idle(C2 b);
};

void C1::set_status(int state)
{
    status = state;
}

void C2::set_status(int state)
{
    status = state;
}

// idle() is member of C1, but friend of C2
int C1::idle(C2 b)
{
    if(THIS->status || b.status) return 0;
    else return 1;
}

int main()
{
    C1 x;
    C2 y;
    x.set_status(IDLE);
    y.set_status(IDLE);
    if(x.idle(y)) cout << "Screen can be used.\n";
    else cout << "In use.\n";
    x.set_status(INUSE);
    if(x.idle(y)) cout << "Screen can be used.\n";
    else cout << "In use.\n";
    return 0;
}

```

clase prieten

- daca avem o clasa prieten, toate functiile membre ale clasei prieten au acces la membrii privati ai clasei

```

// Using a friend class.
#include <iostream>
using namespace std;

class TwoValues {
    int a;
    int b;
public:
    TwoValues(int i, int j) { a = i; b = j; }
    friend class Min;
};

class Min {
public:
    int min(TwoValues x);
};

int Min::min(TwoValues x)
{ return x.a < x.b ? x.a : x.b; }

int main()
{
    TwoValues ob(10, 20);
    Min m;
    cout << m.min(ob);
    return 0;
}

```

functii inline

- foarte comune in clase
- doua tipuri: explicit (inline) si implicit

Explicit

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

```
inline int max(int a, int b)
{
    return a>b ? a : b;
}
```

```
int main()
{
    cout << max(10, 20);
    cout << " " << max(99, 88);
    return 0;
}
```

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

```
int main()
{
    cout << (10>20 ? 10 : 20);
    cout << " " << (99>88 ? 99 : 88);
    return 0;
}
```

functii inline

- executie rapida
- este o sugestie/cerere pentru compilator
- pentru functii foarte mici
- pot fi si membri ai unei clase

```

#include <iostream>
using namespace std;

class myclass {
    int a, b;
public:
    void init(int i, int j);
    void show();
};

// Create an inline function.
inline void myclass::init(int i, int j)
{   a = i; b = j; }

// Create another inline function.
inline void myclass::show()
{ cout << a << " " << b << "\n"; }

int main()
{
    myclass x;
    x.init(10, 20);
    x.show();
    return 0;
}

```

Definirea functiilor inline implicit (in clase)

```

#include <iostream>
using namespace std;

```

```

class myclass {
    int a, b;
public:
    // automatic inline
    void init(int i, int j) { a=i; b=j; }
    void show() { cout << a << " " << b << "\n"; }
};

int main()
{
    myclass x;
    x.init(10, 20);
    x.show();
    return 0;
}

```

```

#include <iostream>
using namespace std;

```

```

class myclass {
    int a, b;
public:
    // automatic inline
    void init(int i, int j)
    {
        a = i;
        b = j;
    }

    void show()
    {
        cout << a << " " << b << "\n";
    }
};

```


Constructorii parametrizati

- trimitem argumente la constructori
- putem defini mai multe variante cu mai multe numere si tipuri de parametrii
- overload de constructori

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

```
class myclass {
    int a, b;
public:
    myclass(int i, int j) {a=i; b=j;}
    void show() {cout << a << " " << b;}
};
```

```
int main()
{
    myclass ob(3, 5);      myclass ob = myclass(3, 4);
    ob.show();
    return 0;
}
```

- a doua forma implica “copy constructors”
- discutat mai tarziu

```

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

const int IN = 1;
const int CHECKED_OUT = 0;

class book {
    char author[40];
    char title[40];
    int status;
public:
    book(char *n, char *t, int s);
    int get_status() {return status;}
    void set_status(int s) {status = s;}
    void show();
};

book::book(char *n, char *t, int s)
{
    strcpy(author, n);
    strcpy(title, t);
    status = s;
}

void book::show()
{
    cout << title << " by " << author;
    cout << " is ";
    if(status==IN) cout << "in.\n";
    else cout << "out.\n";
}

int main()
{
    book b1("Twain", "Tom Sawyer", IN);
    book b2("Melville", "Moby Dick", CHECKED_OUT);
    b1.show();
    b2.show();
    return 0;
}

```

constructori cu un paramentru

- se creeaza o conversie implicita de date

```

#include <iostream>
using namespace std;

class X {
    int a;
public:
    X(int j) { a = j; }
    int geta() { return a; }
};

int main()
{
    X ob = 99; // passes 99 to j
    cout << ob.geta(); // outputs 99
    return 0;
}

```

Membrii statici ai claselor

- elemente statice: date, functii

Membrii statici: de tip date

- variabila precedata de “static”
 - o singura copie din acea variabila va exista pentru toata clasa
 - deci fiecare obiect din clasa poate accesa campul respectiv, dar in memorie nu avem decat o singura variabila definita astfel
 - variabilele initializate cu 0 inainte de crearea primului obiect

- o variabila statica declarata in clasa nu este definita (nu s-a alocat inca spatiu pentru ea)
- deci trebuie definita in mod global in afara clasei
- aceasta definitie din afara clasei ii aloca spatiu in memorie

```
#include <iostream>
using namespace std;
class shared {
    static int a;
    int b;
public:
    void set(int i, int j) {a=i; b=j;}
    void show();};

int shared::a; // define a

void shared::show(){
    cout << "This is static a: " << a;
    cout << "\nThis is non-static b: " << b;
    cout << "\n";}

int main(){
    shared x, y;
    x.set(1, 1); // set a to 1
    x.show();
    y.set(2, 2); // change a to 2
    y.show();
    x.show(); /* Here, a has been changed for both x and y
               because a is shared by both objects. */
    return 0;}
```

```
This is static a: 1
This is non-static b: 1
This is static a: 2
This is non-static b: 2
This is static a: 2
This is non-static b: 1
```

- C++ vechi NU cerea redeclararea lui “a” prin operatorul de rezolutie de scop ::
- daca intalniti cod C++ vechi de acest tip este RECOMANDABIL sa se redefineasca variabilele statice folosind sintaxa C++ modern
- DE CE? creeaza inconsistente

Variabile statice de instantă

- pot fi folosite înainte de a avea un obiect din clasa respectiva
- in continuare: variabila de instantă definită static și public
- ca să folosim o asemenea variabilă de instantă folosim operatorul de rezolutie de scop cu numele clasei

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

class shared {
public:
    static int a;
};

int shared::a; // define a

int main()
{
    // initialize a before creating any objects
    shared::a = 99;
    cout << "This is initial value of a: " << shared::a;
    cout << "\n";
    shared x;
    cout << "This is x.a: " << x.a;
    return 0;
}
```

Folosirea variabilelor statice de instanta

- semafoare: controlul asupra unei resurse pe care pot lucra mai multe obiecte

```

#include <iostream>
using namespace std;

class cl {
    static int resource;
public:
    int get_resource();
    void free_resource() {resource = 0;}
};

int cl::resource; // define resource

int cl::get_resource()
{
    if(resource) return 0; // resource already in use
    else {
        resource = 1;
        return 1; // resource allocated to this object
    }
}

int main()
{
    cl ob1, ob2;
    if(ob1.get_resource()) cout << "ob1 has resource\n";
    if(!ob2.get_resource()) cout << "ob2 denied resource\n";
    ob1.free_resource(); // let someone else use it
    if(ob2.get_resource())
        cout << "ob2 can now use resource\n";
    return 0;
}

```

Alte exemplu

- Numararea obiectelor dintr-o clasa

```

#include <iostream>
using namespace std;

class Counter {
public:
    static int count;
    Counter() { count++; }
    ~Counter() { count--; }
};

int Counter::count;

void f()
{
    Counter temp;
    cout << "Objects in existence: ";
    cout << Counter::count << "\n";
    // temp is destroyed when f() returns
}

int main(void)
{
    Counter o1;
    cout << "Objects in existence: ";
    cout << Counter::count << "\n";
    Counter o2;
    cout << "Objects in existence: ";
    cout << Counter::count << "\n";
    f();
    cout << "Objects in existence: ";
    cout << Counter::count << "\n";
    return 0;
}

```

```

Objects in existence: 1
Objects in existence: 2
Objects in existence: 3
Objects in existence: 2

```

- Folosirea variabilor statice de instanta elimina necesitatea variabilelor globale
- folosirea variabilelor globale aproape intotdeauna violeaza principiul encapsularii datelor, si deci nu este in concordanta cu OOP

functii statice pentru clase

- au dreptul sa foloseasca doar elemente statice din clasa (sau accesibile global)
- nu au pointerul “this”
- nu putem avea varianta statica si non-statica pentru o functie
- nu pot fi declarate ca “virtual” (legat de mostenire)
- folosire limitata (initializare de date statice)

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

class cl {
    static int resource;
public:
    static int get_resource();
    void free_resource() { resource = 0; }
};

int cl::resource; // define resource

int cl::get_resource()
{
    if(resource) return 0; // resource already in use
    else {
        resource = 1;
        return 1; // resource allocated to this object
    }
}
```

```
int main()
{
    cl ob1, ob2;
    /* get_resource() is static so may be called
    independent of any object. */

    if(cl::get_resource()) cout << "ob1 has resource\n";
    if(!cl::get_resource()) cout << "ob2 denied resource\n";
    ob1.free_resource();
    if(ob2.get_resource()) // can still call using object syntax
        cout << "ob2 can now use resource\n";
    return 0;
}
```

```

#include <iostream>
using namespace std;

class static_type {
    static int i;
public:
    static void init(int x) {i = x;}
    void show() {cout << i;}
};

int static_type::i; // define i

int main()
{
    // init static data before object creation
    static_type::init(100);
    static_type x;
    x.show(); // displays 100
    return 0;
}

```

- folosirea uzuala a functiilor statice

Chestiuni despre constructori si desctructori

- constructor: executat la crearea obiectului
- destructor : executat la distrugerea obiectului
- obiecte globale: constructorii executati in ordinea in care sunt definite obiectele
 - destructorii: dupa ce main s-a terminat in ordinea inversa a constructorilor

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

```
class myclass {
    int who;
public:
    myclass(int id);
    ~myclass();
} glob_ob1(1), glob_ob2(2);
```

```
myclass::myclass(int id)
{
    cout << "Initializing " << id << "\n";
    who = id;}

myclass::~~myclass()
{ cout << "Destructing " << who << "\n";}
```

```
int main()
{
    myclass local_ob1(3);
    cout << "This will not be first line displayed.\n";
    myclass local_ob2(4);
    return 0;
}
```

Initializing 1
Initializing 2
Initializing 3
This will not be first line displayed.
Initializing 4
Destructing 4
Destructing 3
Destructing 2
Destructing 1

Operatorul de rezolutie de scop ::

```
int i; // global i
void f()
{
    int i; // local i
    i = 10; // uses local i
    .
    .
    .
}
```

```
int i; // global i
void f()
{
    int i; // local i
    ::i = 10; // now refers to global i
    .
    .
    .
}
```

Clase locale

- putem defini clase in clase sau functii
- class este o declaratie, deci defineste un scop
- operatorul de rezolutie de scop ajuta in aceste cazuri
- rar utilizate clase in clase

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

```
void f();
```

```
int main()
{
    f();
    // myclass not known here
    return 0;
}
```

```
void f()
{
    class myclass {
        int i;
    public:
        void put_i(int n) { i=n; }
        int get_i() { return i; }
    } ob;
```

```
    ob.put_i(10);
    cout << ob.get_i();
}
```

- exemplu de clasa in functia f()
- restrictii: functii definite in clasa
- nu acceseaza variabilele locale ale functiei
- acceseaza variabilele definite static
- fara variabile static definite in clasa

transmitere de obiecte catre functii

- similar cu tipurile predefinite
- call-by-value
- constructori-destructori!

```
// Passing an object to a function.
#include <iostream>
using namespace std;

class myclass {
    int i;
public:
    myclass(int n);
    ~myclass();
    void set_i(int n) { i=n; }
    int get_i() { return i; }
};

myclass::myclass(int n)
{
    i = n;
    cout << "Constructing " << i << "\n";
}

myclass::~~myclass()
{
    cout << "Destroying " << i << "\n";
}

void f(myclass ob);
```

```
int main()
{
    myclass o(1);
    f(o);
    cout << "This is i in main: ";
    cout << o.get_i() << "\n";
    return 0;
}

void f(myclass ob)
{
    ob.set_i(2);
    cout << "This is local i: " << ob.get_i();
    cout << "\n";
}
```

```
Constructing 1
This is local i: 2
Destroying 2
This is i in main: 1
Destroying 1
```

Discutie

- apelam constructorul cand creem obiectul o
- apelam de DOUA ori destructorul
- la apel de functie: apel prin valoare, o copie a obiectului e creata
 - apelam constructorul?
 - la finalizare apelam destructorul?

- La apel de functie constructorul “normal”
NU ESTE APELAT
- se apeleaza asa-numitul “constructorul de copiere”
- un asemenea constructor defineste cum se copiaza un obiect
- se poate defini explicit de catre programator
 - daca nu e definit C++ il creeaza automat

Constructor de copiere

- C++ il definește pentru a face o copie identică pe date
- constructorul e folosit pentru initializare
- constr. de copiere e folosit pentru obiect deja initializat, doar copiaza
- vrem sa folosim starea curenta a obiectului, nu starea initiala a unui obiect din clasa respectiva

destructori pentru obiecte transmise ca la functii

- trebuie sa distrugem obiectul respectiv
- chemam destructorul
- putem avea probleme cu obiecte care folosesc memoria dinamic: la distrugere copia elibereaza memoria, obiectul din main este defect (nu mai are memorie alocata)
- in aceste cazuri trebuie sa redefinim operatorul de copiere (copy constructor)

Functii care intorc obiecte

- o functie poate intoarce obiecte
- un obiect temporar este creat automat pentru a tine informatiile din obiectul de intors
- acesta este obiectul care este intors
- dupa ce valoarea a fost intoarsa, acest obiect este distrus
- probleme cu memoria dinamica: solutie polimorfism pe = si pe constructorul de copiere

```
// Returning objects from a function.
#include <iostream>
using namespace std;

class myclass {
    int i;
public:
    void set_i(int n) { i=n; }
    int get_i() { return i; }
};

myclass f(); // return object of type myclass

int main()
{
    myclass o;
    o = f();
    cout << o.get_i() << "\n";
    return 0;
}

myclass f()
{
    myclass x;
    x.set_i(1);
    return x;
}
```


copierea prin operatorul =

- este posibil sa dam valoarea unui obiect altui obiect
- trebuie sa fie de acelasi tip (aceeasi clasa)

```
// Assigning objects.

#include <iostream>
using namespace std;

class myclass {
    int i;
public:
    void set_i(int n) { i=n; }
    int get_i() { return i; }
};

int main()
{
    myclass ob1, ob2;
    ob1.set_i(99);
    ob2 = ob1; // assign data from ob1 to ob2
    cout << "This is ob2's i: " << ob2.get_i();
    return 0;
}
```

```

#include<iostream.h>
class B
{ int i;
  public: B() { i=1; }
        virtual int get_i() { return i; }
};
class D: public B
{ int j;
  public: D() { j=2; }
        int get_i() {return B::get_i()+j; }
};
int main()
{ const int i = cin.get();
  if (i%3) { D o;}
  else {B o;}
  cout<<o.get_i();
  return 0;
}

```

```

#include <iostream.h>
class A
{ static int x;
  public: A(int i=0) {x=i; }
        int get_x() { return x; }
        int& set_x(int i) { x=i;}
        A operator=(A a1) { set_x(a1.get_x()); return a1;}
};
int main()
{ A a(212), b;
  cout<<(b=a).get_x();
  return 0;
}

```

```

#include<iostream.h>
class B
{ int i;
  public: B() { i=1; }
          virtual int get_i() { return i; } };
class D: virtual public B
{ int j;
  public: D() { j=2; }
          int get_i() {return B::get_i()+j; } };
class D2: virtual public B
{ int j2;
  public: D2() { j2=3; }
          int get_i() {return B::get_i()+j2; } };
class MM: public D, public D2
{ int x;
  public: MM() { x=D::get_i()+D2::get_i(); }
          int get_i() {return x; } };

int main()
{ B *o= new MM();
  cout<<o->get_i()<<"\n";
  MM *p= dynamic_cast<MM*>(o);
  if (p) cout<<p->get_i()<<"\n";
  D *p2= dynamic_cast<D*>(o);
  if (p2) cout<<p2->get_i()<<"\n";
  return 0;
}

```

```

#include<iostream.h>
class B
{ protected: int x;
  public: B(int i=28) { x=i; }
          virtual B f(B ob) { return x+ob.x+1; }
          void afisare(){ cout<<x; } };
class D: public B
{ public: D(int i=-32):B(i) {}
          B f(B ob) { return x+ob.x-1; } };

int main()
{ B *p1=new D, *p2=new B, *p3=new B(p1->f(*p2));
  p3->afisare();
  return 0;
}

```

Cursul de programare orientata pe obiecte

Seria 14

Saptamana 4, 19 martie 2013

Andrei Paun

Cuprinsul cursului: 19 martie 2013

- supraincarcarea functiilor in C++
- supraincarcarea operatorilor in C++

Constructor de copiere

- C++ il definește pentru a face o copie identică pe date
- constructorul e folosit pentru initializare
- constr. de copiere e folosit pentru obiect deja initializat, doar copiaza
- vrem sa folosim starea curenta a obiectului, nu starea initiala a unui obiect din clasa respectiva

destructori pentru obiecte transmise ca tre functii

- trebuie sa distrugem obiectul respectiv
- chemam destructorul
- putem avea probleme cu obiecte care folosesc memoria dinamic: la distrugere copia elibereaza memoria, obiectul din main este defect (nu mai are memorie alocata)
- in aceste cazuri trebuie sa redefinim operatorul de copiere (copy constructor)

Functii care intorc obiecte

- o functie poate intoarce obiecte
- un obiect temporar este creat automat pentru a tine informatiile din obiectul de intors
- acesta este obiectul care este intors
- dupa ce valoarea a fost intoarsa, acest obiect este distrus
- probleme cu memoria dinamica: solutie polimorfism pe = si pe constructorul de copiere

```
// Returning objects from a function.
#include <iostream>
using namespace std;

class myclass {
    int i;
public:
    void set_i(int n) { i=n; }
    int get_i() { return i; }
};

myclass f(); // return object of type myclass

int main()
{
    myclass o;
    o = f();
    cout << o.get_i() << "\n";
    return 0;
}

myclass f()
{
    myclass x;
    x.set_i(1);
    return x;
}
```

copierea prin operatorul =

- este posibil sa dam valoarea unui obiect altui obiect
- trebuie sa fie de acelasi tip (aceeasi clasa)

```
// Assigning objects.

#include <iostream>
using namespace std;

class myclass {
    int i;
public:
    void set_i(int n) { i=n; }
    int get_i() { return i; }
};

int main()
{
    myclass ob1, ob2;
    ob1.set_i(99);
    ob2 = ob1; // assign data from ob1 to ob2
    cout << "This is ob2's i: " << ob2.get_i();
    return 0;
}
```

Supraincercarea functiilor

- este folosirea aceluiasi nume pentru functii diferite
- functii diferite, dar cu inteles apropiat
- compilatorul foloseste numarul si tipul parametrilor pentru a diferentia apelurile

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

int myfunc(int i); // these differ in types of parameters
double myfunc(double i);

int main()
{
    cout << myfunc(10) << " "; // calls myfunc(int i)
    cout << myfunc(5.4); // calls myfunc(double i)
    return 0;
}

double myfunc(double i)
{
    return i;
}

int myfunc(int i)
{
    return i;
}
```

tipuri diferite pentru parametrul i


```

#include <iostream>
using namespace std;

int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);

int main()
{
    cout << myfunc(10) << " "; // calls myfunc(int i)
    cout << myfunc(4, 5); // calls myfunc(int i, int j)
    return 0;
}

int myfunc(int i)
{
    return i;
}

int myfunc(int i, int j)
{
    return i*j;
}

```

numar diferit de parametri

- daca diferenta este doar in tipul de date
intors: eroare la compilare

```

int myfunc(int i); // Error: differing return types are
float myfunc(int i); // insufficient when overloading.

```

- sau tipuri care `_par_` sa fie diferite

```

void f(int *p);
void f(int p[]); // error, *p is same as p[]

```

Polimorfism pe constructori

- foarte comun sa fie supraincarcati
- de ce?
 - flexibilitate
 - pentru a putea defini obiecte initializate si neinitializate
 - constructori de copiere: copy constructors

overload pe constructori: flexibilitate

- putem avea mai multe posibilitati pentru initializarea/construirea unui obiect
- definim constructori pentru toate modurile de initializare
- daca se incearca initializarea intr-un alt fel (decat cele definite): eroare la compilare

<pre> #include <iostream> #include <cstdio> using namespace std; class date { int day, month, year; public: date(char *d); date(int m, int d, int y); void show_date(); }; // Initialize using string. date::date(char *d) { sscanf(d, "%d%*c%d%*c%d", &month, &day, &year); } // Initialize using integers. date::date(int m, int d, int y) { day = d; month = m; year = y; } </pre>	<pre> void date::show_date() { cout << month << "/" << day; cout << "/" << year << "\n"; } int main() { date ob1(10, 2, 4, 2003), ob2("10/22/2003"); ob1.show_date(); ob2.show_date(); return 0; } </pre>
--	--

"%d%*c%d%*c%d" "%d%*c%d%*c%d"

citim din sir
 *: ignoram ce citim
 c: un singur caracter

citim 3 intregi sau
 luna/zi/an

polimorfism de constructori: obiecte initializate si ne-initializate

- important pentru array-uri dinamice de obiecte
- nu se pot initializa obiectele dintr-o lista alocata dinamic
- asadar avem nevoie de posibilitatea de a crea obiecte neinitializate (din lista dinamica) si obiecte initializate (definite normal)

```

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

class powers {
    int x;
public:
    // overload constructor two ways
    powers(int x) : x(x) {}
    powers() {}
};

int main()
{
    powers ofTwo[] = {1, 2, 4, 8, 16}; // initialized
    powers ofThree[5]; // uninitialized
    powers *p;
    int i;

    // show powers of two
    cout << "Powers of two: ";
    for(i=0; i<5; i++) {
        cout << ofTwo[i].getX() << " ";
    }
    cout << "\n\n";

    // set powers of three
    ofThree[0].setx(1); ofThree[1].setx(3);
    ofThree[2].setx(9); ofThree[3].setx(27);
    ofThree[4].setx(81);

    // show powers of three
    cout << "Powers of three: ";
    for(i=0; i<5; i++) { cout << ofThree[i].getX() << " ";}

    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;}

    // initialize dynamic array with powers of two
    for(i=0; i<5; i++) { p[i].setx(ofTwo[i].getX());}

    // show powers of two
    cout << "Powers of two: ";
    for(i=0; i<5; i++) {cout << p[i].getX() << " ";}
    cout << "\n\n";
    delete [] p;
    return 0;
}

```

- ofThree si lista p au nevoie de constructorul fara parametri

polimorfism de constructori: constructorul de copiere

- pot aparea probleme cand un obiect initializeaza un alt obiect

MyClass B = A;

- aici se copiaza toate campurile (starea) obiectului A in obiectul B
- problema apare la alocare dinamica de memorie: A si B folosesc aceeasi zona de memorie pentru ca pointerii arata in acelasi loc
- destructorul lui MyClass elibereaza aceeasi zona de memorie de doua ori (distruge A si B)

constructorul de copiere

- aceeași problemă
 - apel de funcție cu obiect ca parametru
 - apel de funcție cu obiect ca variabilă de întoarcere
 - în aceste cazuri un obiect temporar este creat, se copiază prin constructorul de copiere în obiectul temporar, și apoi se continuă
 - deci vor fi din nou două distrugeri de obiecte din clasa respectivă (una pentru parametru, una pentru obiectul temporar)

putem redefini constructorii de copiere

```
classname (const classname &o) {  
    // body of constructor  
}
```

- o este obiectul din dreapta
- putem avea mai mulți parametri (dar trebuie să definim valori implicite pentru ei)
- & este apel prin referință
- putem avea și atribuire (o1=o2;)
 - redefinim operatorii mai târziu, putem redefini =
 - = diferit de initializare

```

#include <iostream>
#include <new>
#include <cstdlib>
using namespace std;

class array {
    int *p;
    int size;
public:
    array(int sz) {
        try {
            p = new int[sz];
        } catch (bad_alloc xa) {
            cout << "Allocation Failure\n";
            exit(EXIT_FAILURE);
        }
        size = sz;
    }
    ~array() { delete [] p; }

    // copy constructor
    array(const array &a);
    void put(int i, int j) {if(i>=0 && i<size) p[i] = j;}
    int get(int i) { return p[i];}
};

// Copy Constructor
array::array(const array &a) {
    int i;
    try {
        p = new int[a.size];
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        exit(EXIT_FAILURE);
    }
    for(i=0; i<a.size; i++) p[i] = a.p[i];
}

int main()
{
    array num(10);
    int i;

    for(i=0; i<10; i++) num.put(i, i);
    for(i=9; i>=0; i--) cout << num.get(i);
    cout << "\n";

    // create another array and initialize with num
    array x(num); // invokes copy constructor
    for(i=0; i<10; i++) cout << x.get(i);

    return 0;
}

```

- Observatie: constructorul de copiere este folosit doar la initializari
- daca avem array a(10); array b(10); b=a;
 - nu este initializare, este copiere de stare
 - este posibil sa trebuiasca redefinit si operatorul = (mai tarziu)

pointeri catre functii polimorifice

- putem avea pointeri catre functii (C)
- putem avea pointeri catre functii polimorifice
- cum se defineste pointerul ne spune catre ce versiune a functiei cu acelasi nume aratam

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

```
int myfunc(int a);
int myfunc(int a, int b);
```

```
int main()
```

```
{
    int (*fp)(int a); // pointer to int f(int)
    fp = myfunc; // points to myfunc(int)
    cout << fp(5);
    return 0;
}
```

```
int myfunc(int a)
```

```
{
    return a;
}
```

```
int myfunc(int a, int b)
```

```
{
    return a*b;
}
```

- semnatura functiei din definitia pointerului ne spune ca mergem spre functia cu un parametru
 - trebuie sa existe una din variantele polimorifice care este la fel cu definitia pointerului

Argumente implicite pentru functii

- putem defini valori implicite pentru parametrii unei functii
- valorile implicite sunt folosite atunci cand acei parametri nu sunt dati la apel

```
void myfunc(double d = 0.0)
{
    // ...
}
...
myfunc(198.234); // pass an explicit value
myfunc(); // let function use default
```

-

Argumente implicite

- dau posibilitatea pentru flexibilitate
- majoritatea functiilor considera cel mai general caz, cu parametrii impliciti putem sa chemam o functie pentru cazuri particulare
- multe functii de I/O folosesc arg. implicite
- nu avem nevoie de overload


```

#include <iostream>
using namespace std;

void clrscr(int size=25);

int main()
{
    register int i;
    for(i=0; i<30; i++ ) cout << i << endl;
    cin.get();
    clrscr(); // clears 25 lines
    for(i=0; i<30; i++ ) cout << i << endl;
    cin.get();
    clrscr(10); // clears 10 lines

    return 0;
}

void clrscr(int size)
{
    for(; size; size--) cout << endl;
}

```

- se pot refolosi valorile unor parametri

```

void iputs(char *str, int indent)
{
    if(indent < 0) indent = 0;
    for( ; indent; indent--) cout << " ";
    cout << str << "\n";
}

```

```

#include <iostream>
using namespace std;

/* Default indent to -1. This value tells the function to reuse the
previous value. */
void iputs(char *str, int indent = -1);

int main()
{
    iputs("Hello there", 10);
    iputs("This will be indented 10 spaces by default");
    iputs("This will be indented 5 spaces", 5);
    iputs("This is not indented", 0);

    return 0;
}

void iputs(char *str, int indent)
{
    static i = 0; // holds previous indent value
    if(indent >= 0)
        i = indent;
    else // reuse old indent value
        indent = i;
    for( ; indent; indent--) cout << " ";
    cout << str << "\n";
}

```

Hello there
 This will be indented 10 spaces by default
 This will be indented 5 spaces
 This is not indented

parametrii impliciti

- se specifica o singura data
- pot fi mai multi
- toti sunt la dreapta
- putem avea param. impliciti in definitia constructorilor
 - nu mai facem overload pe constructor
 - nu trebuie sa ii precizam mereu la declarare

```

#include <iostream>
using namespace std;

class cube {
    int x, y, z;
public:
    cube(int i=0, int j=0, int k=0) {          cube() {x=0; y=0; z=0}
        x=i;
        y=j;
        z=k;
    }

    int volume() {
        return x*y*z;
    }
};

int main()
{
    cube a(2,3,4), b;
    cout << a.volume() << endl;
    cout << b.volume();

    return 0;
}

```

<pre> // A customized version of strcat(). #include <iostream> #include <cstring> using namespace std; void mystrcat(char *s1, char *s2, int len = -1); int main() { char str1[80] = "This is a test"; char str2[80] = "0123456789"; mystrcat(str1, str2, 5); // concatenate 5 chars cout << str1 << '\n'; strcpy(str1, "This is a test"); // reset str1 mystrcat(str1, str2); // concatenate entire string cout << str1 << '\n'; return 0; } </pre>	<pre> // A custom version of strcat(). void mystrcat(char *s1, char *s2, int len) { // find end of s1 while(*s1) s1++; if(len == -1) len = strlen(s2); while(*s2 && len) { *s1 = *s2; // copy chars s1++; s2++; len--; } *s1 = '\0'; // null terminate s1 } </pre>
---	---

parametrii impliciti

- modul corect de folosire este de a defini un asemenea parametru cand se subantelege valoarea implicita
- daca sunt mai multe posibilitati pentru valoarea implicita e mai bine sa nu se foloseasca (lizibilitate)
- cand se foloseste un param. implicit nu trebuie sa faca probleme in program

Ambiguitati pentru polimorfism de functii

- erori la compilare
- majoritatea datorita conversiilor implicite

```
int myfunc(double d);  
// ...  
cout << myfunc('c'); // not an error, conversion applied
```

```

#include <iostream>
using namespace std;

float myfunc(float i);
double myfunc(double i);

int main()
{
    cout << myfunc(10.1) << " "; // unambiguous, calls myfunc(double)
    cout << myfunc(10); // ambiguous
    return 0;
}

float myfunc(float i)
{
    return i;
}

double myfunc(double i)
{
    return -i;
}

```

- problema nu e de definire a functiilor myfunc,
- problema apare la apelul functiilor

```

#include <iostream>
using namespace std;

char myfunc(unsigned char ch);
char myfunc(char ch);

int main()
{
    cout << myfunc('c'); // this calls myfunc(char)
    cout << myfunc(88) << " "; // ambiguous

    return 0;
}

char myfunc(unsigned char ch)
{
    return ch-1;
}

char myfunc(char ch)
{
    return ch+1;
}

```

- ambiguitate intre char si unsigned char
- ambiguitate pentru functii cu param. impliciti

```

#include <iostream>
using namespace std;

int myfunc(int i);
int myfunc(int i, int j=1);

int main()
{
    cout << myfunc(4, 5) << " "; // unambiguous
    cout << myfunc(10); // ambiguous
    return 0;
}

int myfunc(int i)
{
    return i;
}

int myfunc(int i, int j)
{
    return i*j;
}

```

// This program contains an error.

```
#include <iostream>
```

```
using namespace std;
```

```
void f(int x);
```

```
void f(int &x); // error
```

```
int main()
```

```
{
```

```
    int a=10;
```

```
    f(a); // error, which f()?
```

```
    return 0;
```

```
}
```

```
void f(int x)
```

```
{
```

```
    cout << "In f(int)\n";
```

```
}
```

```
void f(int &x)
```

```
{
```

```
    cout << "In f(int &)\n";
```

```
}
```

- doua tipuri de apel: prin valoare si prin referinta, ambiguitate!
- mereu eroare de ambiguitate

Supraincercarea operatorilor in C++

- majoritatea operatorilor pot fi supraincarcati
- similar ca la functii
- una din proprietatile C++ care ii confera putere
- s-a facut supraincercarea operatorilor si pentru operatii de I/O (<<,>>)
- supraincercarea se face definind o functie operator: membru al clasei sau nu

functii operator membri ai clasei

```
ret-type class-name::operator#(arg-list)
{
  // operations
}
```

- # este operatorul supraincarcat (+ - * / ++ -- = , etc.)
- de obicei ret-type este tipul clasei, dar avem flexibilitate
- pentru operatori unari arg-list este vida
- pentru operatori binari: arg-list contine un element

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

```
class loc {
  int longitude, latitude;
public:
  loc() {}
  loc(int lg, int lt) {
    longitude = lg;
    latitude = lt;
  }
}
```

```
void show() {
  cout << longitude << " ";
  cout << latitude << "\n";
}
```

```
loc operator+(loc op2);
};
```

```
// Overload + for loc.
```

```
loc loc::operator+(loc op2)
{
  loc temp;
  temp.longitude = op2.longitude + longitude;
  temp.latitude = op2.latitude + latitude;
  return temp;
}
```

```
int main()
{
  loc ob1(10, 20), ob2( 5, 30);
  ob1.show(); // displays 10 20
  ob2.show(); // displays 5 30
  ob1 = ob1 + ob2;
  ob1.show(); // displays 15 50
}
```

```
return 0;
```

```
}
```

- un singur argument pentru ca avem **this**
- longitude==this->longitude
- obiectul din stanga face apelul la functia operator
 - ob1a chemat operatorul + redefinit in clasa lui ob1

- daca intoarcem acelasi tip de date in operator putem avea expresii
- daca intorceam alt tip nu puteam face
 - putem avea si $ob1 = ob1 + ob2;$
 - $(ob1+ob2).show();$ // displays outcome of $ob1+ob2$
- pentru ca functia show() este definita in clasa lui ob1
- se genereaza un obiect temporar
 - (constructor de copiere)

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

class loc {
    int longitude, latitude;
public:
    loc() {} // needed to construct temporaries
    loc(int lg, int lt) { longitude = lg; latitude = lt; }
    void show() { cout << longitude << " ";
                  cout << latitude << "\n"; }
    loc operator+(loc op2);
    loc operator-(loc op2);
    loc operator=(loc op2);
    loc operator++();
};

// Overload + for loc.
loc loc::operator+(loc op2){
    loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    return temp;}

loc loc::operator-(loc op2){
    loc temp;
    temp.longitude = longitude - op2.longitude;
    temp.latitude = latitude - op2.latitude;
    return temp;}

// Overload assignment for loc.
loc loc::operator=(loc op2){
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this; } // object that generated call

// Overload prefix ++ for loc.
loc loc::operator++(){
    longitude++;
    latitude++;
    return *this;}

int main(){
    loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
    ob1.show();
    ob2.show();
    ++ob1;
    ob1.show(); // displays 11 21
    ob2 = ++ob1;
    ob1.show(); // displays 12 22
    ob2.show(); // displays 12 22
    ob1 = ob2 = ob3; // multiple assignment
    ob1.show(); // displays 90 90
    ob2.show(); // displays 90 90

    return 0;
}
```


- apelul la functia operator se face din obiectul din stanga (pentru operatori binari)
 - din aceasta cauza pentru – avem functia definita asa
- operatorul = face copiere pe variabilele de instanta, intoarce *this
- se pot face atribuiiri multiple (dreapta spre stanga)

Formele prefix si postfix

- am vazut prefix, pentru postfix: definim un parametru int “dummy”

// Prefix increment

type operator++() {

// body of prefix operator

}

// Postfix increment

type operator++(int x) {

// body of postfix operator

}

supraincarcarea +=, *=, etc.

```
loc loc::operator+=(loc op2)
{
    longitude = op2.longitude + longitude;
    latitude = op2.latitude + latitude;
    return *this;
}
```

restrictii

- nu se poate redefini si precedenta operatorilor
- nu se poate redefini numarul de operanzi
 - rezonabil pentru ca redefinim pentru lizibilitate
 - putem ignora un operand daca vrem
- nu putem avea valori implicite; exceptie pentru ()
- nu putem face overload pe . :: .* ?
- e bine sa facem operatiuni apropiate de intelesul operatorilor respectivi

- Este posibil sa facem o decuplare completa intre intelesul initial al operatorului
 - exemplu: << >>
- mostenire: operatorii (mai putin =) sunt mosteniti de clasa derivata
- clasa derivata poate sa isi redefineasca operatorii

Supraincercarea operatorilor ca functii prieten

- operatorii pot fi definiti si ca functie nemembra a clasei
- o facem functie prietena pentru a putea accesa rapid campurile protejate
- nu avem pointerul “this”
- deci vom avea nevoie de toti operanzii ca parametri pentru functia operator
- primul parametru este operandul din stanga, al doilea parametru este operandul din dreapta

```

#include <iostream>
using namespace std;

class loc {
    int longitude, latitude;
public:
    loc() {} // needed to construct temporaries
    loc(int lg, int lt) {longitude = lg; latitude = lt;}
    void show() { cout << longitude << " ";
                  cout << latitude << "\n";}
    friend loc operator+(loc op1, loc op2); // friend
    loc operator-(loc op2);
    loc operator=(loc op2);
    loc operator++();
};

// Now, + is overloaded using friend function.
loc operator+(loc op1, loc op2)
{
    loc temp;

    temp.longitude = op1.longitude + op2.longitude;
    temp.latitude = op1.latitude + op2.latitude;

    return temp;
}

// Overload - for loc.
loc loc::operator-(loc op2){
    loc temp;
    // notice order of operands
    temp.longitude = longitude - op2.longitude;
    temp.latitude = latitude - op2.latitude;
    return temp;}

// Overload assignment for loc.
loc loc::operator=(loc op2){
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this;} //return obj. that generated call

// Overload ++ for loc.
loc loc::operator++(){
    longitude++;
    latitude++;
    return *this;}

int main(){
    loc ob1(10, 20), ob2( 5, 30);
    ob1 = ob1 + ob2;
    ob1.show();

    return 0;
}

```

Restricții pentru operatorii definiți ca prieten

- nu se pot supraîncărca = () [] sau -> cu funcții prieten
- pentru ++ sau -- trebuie să folosim referințe

functii prieten pentru operatori unari

- pentru ++, -- folosim referinta pentru a transmite operandul
 - pentru ca trebuie sa se modifice si nu avem pointerul this
 - apel prin valoare: primim o copie a obiectului si nu putem modifica operandul (ci doar copia)

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

class loc {
    int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {longitude = lg;latitude = lt;}
    void show() { cout << longitude << " ";
                  cout << latitude << "\n";}

    loc operator=(loc op2);
    friend loc operator++(loc &op);
    friend loc operator--(loc &op);
};

// Overload assignment for loc.
loc loc::operator=(loc op2){
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this; // return object that generated call
}

// Make op-- a friend; use reference.
loc operator--(loc &op)
{
    op.longitude--;
    op.latitude--;
    return op;
}

int main()
{
    loc ob1(10, 20), ob2;
    ob1.show();
    ++ob1;
    ob1.show(); // displays 11 21
    ob2 = ++ob1;
    ob2.show(); // displays 12 22
    --ob2;
    ob2.show(); // displays 11 21

    return 0;
}

// Now a friend; use a reference parameter.
loc operator++(loc &op) {
    op.longitude++;
    op.latitude++;
    return op;}
```

pentru varianta postfix ++ --

- la fel ca la supraincarcarea operatorilor prin functii membru ale clasei: parametru int

```
// friend, postfix version of ++  
friend loc operator++(loc &op, int x);
```

Diferente supraincarcarea prin membrii sau prieteni

- de multe ori nu avem diferente,
 - atunci e indicat sa folosim functii membru
- uneori avem insa diferente: pozitia operanzilor
 - pentru functii membru operandul din stanga apeleaza functia operator supraincarcata
 - daca vrem sa scriem expresie: 100+ob;
probleme la compilare=> functii prieten

- in aceste cazuri trebuie sa definim doua functii de supraincarcare:
 - `int + tipClasa`
 - `tipClasa + int`

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

class loc {
    int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {longitude = lg; latitude = lt;}
    void show() { cout << longitude << " ";
                 cout << latitude << "\n";}
    friend loc operator+(loc op1, int op2);
    friend loc operator+(int op1, loc op2);
};

// + is overloaded for loc + int.
loc operator+(loc op1, int op2){
    loc temp;
    temp.longitude = op1.longitude + op2;
    temp.latitude = op1.latitude + op2;
    return temp;}

// + is overloaded for int + loc.
loc operator+(int op1, loc op2){
    loc temp;
    temp.longitude = op1 + op2.longitude;
    temp.latitude = op1 + op2.latitude;
    return temp;}

int main()
{
    loc ob1(10, 20), ob2( 5, 30), ob3(7, 14);

    ob1.show();
    ob2.show();
    ob3.show();
    ob1 = ob2 + 10; // both of these
    ob3 = 10 + ob2; // are valid
    ob1.show();
    ob3.show();

    return 0;
}
```

supraincarcarea new si delete

- supraincarcare op. de folosire memorie in mod dinamic pentru cazuri speciale

```
// Allocate an object.  
void *operator new(size_t size)  
{  
/* Perform allocation. Throw bad_alloc on failure.  
Constructor called automatically. */  
return pointer_to_memory;
```

- `size_t`: predefinit Constructor called automatically. */
return pointer_to_memory;
- pentru `new`: constructorul este chemat automat
- pentru `delete`: destructorul // Delete an object.
void operator delete(void *p)este chemat automat
- supraincarcare la nivel {
@ Free memory pointed to by pde clasa sau globala

```
// Delete an object.
void operator delete(void *p)
{
    // Free memory pointed to by p
    Destructeur called automatically. */
}
```

#include <iostream>		// delete overloaded relative to loc.
#include <cstdlib>		void loc::operator delete(void *p){
#include <new>		cout << "In overloaded delete.\n";
using namespace std;		free(p);
class loc {		}
int longitude, latitude;		
public:	• In overloaded new.	
loc() {}		
loc(int lg, int lt) {longitude = lg; latitude = lt;}	• In overloaded new.);
void show() { cout << longitude << " " << latitude << " "; }	• 10 20	error for p1.\n";
void *operator new(size_t size) { return malloc(size); }	• -10 -20	-20);
void operator delete(void *p) { free(p); }	• In overloaded delete.	{
	• In overloaded delete.	error for p2.\n";
// new overloaded relative to loc.		
void *loc::operator new(size_t size) { return malloc(size); }		
void *p;		p = new loc(10, 20);
cout << "In overloaded new.\n";		delete p1;
p = malloc(size);		delete p2;
if(!p) { bad_alloc ba; throw ba; }		return 0;
return p;		}
}		

- daca new sau delete sunt folositi pentru alt tip de date in program, versiunile originale sunt folosite
- se poate face overload pe new si delete la nivel global
 - se declara in afara oricarei clase
 - pentru new/delete definiti si global si in clasa, cel din clasa e folosit pentru elemente de tipul clasei, si in rest e folosit cel redefinit global

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
class loc {
    int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {longitude = lg;latitude = lt;}
    void show() {cout << longitude << " ";
                cout << latitude << "\n";}
};

// Global new
void *operator new(size_t size)
{
    void *p;
    p = malloc(size);
    if(!p) {
        bad_alloc ba;
        throw ba;
    }
    return p;}

```

```
// Global delete
void operator delete(void *p)
{ free(p); }

int main(){
    loc *p1, *p2;
    float *f;
    try {p1 = new loc (10, 20);
        } catch (bad_alloc xa) {
            cout << "Allocation error for p1.\n";
            return 1;        }
    try {p2 = new loc (-10, -20);
        } catch (bad_alloc xa) {
            cout << "Allocation error for p2.\n";
            return 1;        }
    try {f = new float; // uses overloaded new, too
        } catch (bad_alloc xa) {
            cout << "Allocation error for f.\n";
            return 1;        }
    *f = 10.10F; cout << *f << "\n";
    p1->show(); p2->show();
    delete p1; delete p2; delete f;
    return 0; }

```

new si delete pentru array-uri

- facem overload de doua ori

```
// Allocate an array of objects.
void *operator new[](size_t size)
{
    /* Perform allocation. Throw bad_alloc on failure.
    Constructor for each element called automatically. */
    return pointer_to_memory;
}

// Delete an array of objects.
void operator delete[](void *p)
{
    /* Free memory pointed to by p.
    Destructor for each element called automatically.
    */
}
```

supraincarcarea []

- trebuie sa fie functii membru, (nestate)
- nu pot fi functii prieten
- este considerat operator binar
- o[3] se transforma in
type class-name::operator[](int i)
- o.operator[](3)
{
 // ...
}

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

class atype {
    int a[3];
public:
    atype(int i, int j, int k) { a[0] = i; a[1] = j; a[2] = k; }
    int operator[](int i) { return a[i]; }
};

int main()
{
    atype ob(1, 2, 3);
    cout << ob[1]; // displays 2
    return 0;
}
```

- operatorul [] poate fi folosit si la stanga unei atribuirii (obiectul intors este atunci referinta)

```

#include <iostream>
using namespace std;

class atype {
    int a[3];
public:
    atype(int i, int j, int k) { a[0] = i; a[1] = j; a[2] = k; }
    int &operator[](int i) { return a[i]; }
};

int main()
{
    atype ob(1, 2, 3);
    cout << ob[1]; // displays 2
    cout << " ";
    ob[1] = 25; // [] on left of =
    cout << ob[1]; // now displays 25
    return 0;
}

```

- putem in acest fel verifica array-urile
- exemplul urmator

// A safe array example.

```

#include <iostream>
#include <cstdlib>
using namespace std;
class atype {
    int a[3];
public:
    atype(int i, int j, int k) {a[0] = i;a[1] = j;a[2] = k;}
    int &operator[](int i);
};

```

// Provide range checking for atype.

```

int &atype::operator[](int i)
{
    if(i<0 || i> 2) {
        cout << "Boundary Error\n";
        exit(1);
    }
    return a[i];
}

```

```

int main()
{

```

```

    atype ob(1, 2, 3);
    cout << ob[1]; // displays 2
    cout << " ";
    ob[1] = 25; // [] appears on left
    cout << ob[1]; // displays 25
    ob[3] = 44;
        // generates runtime error, 3 out-of-range
    return 0;
}

```

supraincarcarea ()

- nu creem un nou fel de a chema functii
- definim un mod de a chema functii cu numar arbitrar de parametrii

```
double operator()(int a, float f, char *s);
```

```
O(10, 23.34, "hi");
```

echivalent cu `O.operator()(10, 23.34, "hi");`

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

```
class loc {
    int longitude, latitude;
public:
```

10 20

7 8

11 11

```
};
```

```
// Overload () for loc.
```

```
loc loc::operator()(int i, int j)
```

```
{
    longitude = i;
    latitude = j;
    return *this;
}
```

```
// Overload + for loc.
```

```
loc loc::operator+(loc op2)
```

```
{
    loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    return temp;
}
```

```
int main()
```

```
{
    loc ob1(10, 20), ob2(1, 1);
    ob1.show();
    ob1(7, 8); // can be executed by itself
    ob1.show();
    ob1 = ob2 + ob1(10, 10);
                // can be used in expressions
    ob1.show();
    return 0;
}
```

overload pe ->

- operator unar
- obiect->element
 - obiect genereaza apelul
 - element trebuie sa fie accesibil
 - intoarce un pointer catre un obiect din clasa

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

class myclass {
public:
    int i;
    myclass *operator->() {return this;}
};

int main()
{
    myclass ob;
    ob->i = 10; // same as ob.i
    cout << ob.i << " " << ob->i;
    return 0;
}
```

supraincarcarea operatorului ,

- operator binar
- ar trebui ignorate toate valorile mai putin a celui mai din dreapta operand

```
#include <iostream>    10 20    // Overload + for loc
using namespace std;   5 30    loc loc::operator+(loc op2)
                        1 1      {
class loc {             10 60    loc temp;
    int longitude, latitude; 1 1    temp.longitude = op2.longitude + longitude;
public:                 1 1    temp.latitude = op2.latitude + latitude;
    loc() {}           1 1    return temp;
    loc(int lg, int lt) {longitude = lg;latitude = lt;} }
    void show() {cout << longitude << " ";
                        cout << latitude << "\n";}
    loc operator+(loc op2);
    loc operator,(loc op2);
};

// overload comma for loc
loc loc::operator,(loc op2)
{
    loc temp;
    temp.longitude = op2.longitude;
    temp.latitude = op2.latitude;
    cout << op2.longitude << " " << op2.latitude << "\n";
    return temp;
}

int main()
{
    loc ob1(10, 20), ob2( 5, 30), ob3(1, 1);
    ob1.show();
    ob2.show();
    ob3.show();
    cout << "\n";
    ob1 = (ob1, ob2+ob2, ob3);
    ob1.show(); // displays 1 1, the value of ob3
    return 0;
}
```

Cursul de programare orientata pe obiecte

Seria 13

Saptamana 5, 18 martie 2014

Andrei Paun

Cuprinsul cursului: 18 martie 2014

- ~~supraincarcarea functiilor in C++~~
- supraincarcarea operatorilor in C++

pointeri catre functii polimorifice

- putem avea pointeri catre functii (C)
- putem avea pointeri catre functii polimorifice
- cum se defineste pointerul ne spune catre ce versiune a functiei cu acelasi nume aratam

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

```
int myfunc(int a);
int myfunc(int a, int b);
```

```
int main()
```

```
{
    int (*fp)(int a); // pointer to int f(int)
    fp = myfunc; // points to myfunc(int)
    cout << fp(5);
    return 0;
}
```

```
int myfunc(int a)
```

```
{
    return a;
}
```

```
int myfunc(int a, int b)
```

```
{
    return a*b;
}
```

- semnatura functiei din definitia pointerului ne spune ca mergem spre functia cu un parametru
 - trebuie sa existe una din variantele polimorifice care este la fel cu definitia pointerului

Argumente implicite pentru functii

- putem defini valori implicite pentru parametrii unei functii
- valorile implicite sunt folosite atunci cand acei parametri nu sunt dati la apel

```
void myfunc(double d = 0.0)
{
    // ...
}
...
myfunc(198.234); // pass an explicit value
myfunc(); // let function use default
```

-

Argumente implicite

- dau posibilitatea pentru flexibilitate
- majoritatea functiilor considera cel mai general caz, cu parametrii impliciti putem sa chemam o functie pentru cazuri particulare
- multe functii de I/O folosesc arg. implicite
- nu avem nevoie de overload

```

#include <iostream>
using namespace std;

void clrscr(int size=25);

int main()
{
    register int i;
    for(i=0; i<30; i++ ) cout << i << endl;
    cin.get();
    clrscr(); // clears 25 lines
    for(i=0; i<30; i++ ) cout << i << endl;
    cin.get();
    clrscr(10); // clears 10 lines

    return 0;
}

void clrscr(int size)
{
    for(; size; size--) cout << endl;
}

```

- se pot refolosi valorile unor parametri

```

void iputs(char *str, int indent)
{
    if(indent < 0) indent = 0;
    for( ; indent; indent--) cout << " ";
    cout << str << "\n";
}

```

```

#include <iostream>
using namespace std;

/* Default indent to -1. This value tells the function to reuse the
previous value. */
void iputs(char *str, int indent = -1);

int main()
{
    iputs("Hello there", 10);
    iputs("This will be indented 10 spaces by default");
    iputs("This will be indented 5 spaces", 5);
    iputs("This is not indented", 0);

    return 0;
}

void iputs(char *str, int indent)
{
    static i = 0; // holds previous indent value
    if(indent >= 0)
        i = indent;
    else // reuse old indent value
        indent = i;
    for( ; indent; indent--) cout << " ";
    cout << str << "\n";
}

```

Hello there
 This will be indented 10 spaces by default
 This will be indented 5 spaces
 This is not indented

parametrii impliciti

- se specifica o singura data
- pot fi mai multi
- toti sunt la dreapta
- putem avea param. impliciti in definitia constructorilor
 - nu mai facem overload pe constructor
 - nu trebuie sa ii precizam mereu la declarare

```

#include <iostream>
using namespace std;

class cube {
    int x, y, z;
public:
    cube(int i=0, int j=0, int k=0) {          cube() {x=0; y=0; z=0}
        x=i;
        y=j;
        z=k;
    }

    int volume() {
        return x*y*z;
    }
};

int main()
{
    cube a(2,3,4), b;
    cout << a.volume() << endl;
    cout << b.volume();

    return 0;
}

```

<pre> // A customized version of strcat(). #include <iostream> #include <cstring> using namespace std; void mystrcat(char *s1, char *s2, int len = -1); int main() { char str1[80] = "This is a test"; char str2[80] = "0123456789"; mystrcat(str1, str2, 5); // concatenate 5 chars cout << str1 << '\n'; strcpy(str1, "This is a test"); // reset str1 mystrcat(str1, str2); // concatenate entire string cout << str1 << '\n'; return 0; } </pre>	<pre> // A custom version of strcat(). void mystrcat(char *s1, char *s2, int len) { // find end of s1 while(*s1) s1++; if(len == -1) len = strlen(s2); while(*s2 && len) { *s1 = *s2; // copy chars s1++; s2++; len--; } *s1 = '\0'; // null terminate s1 } </pre>
---	---

parametrii impliciti

- modul corect de folosire este de a defini un asemenea parametru cand se subantelege valoarea implicita
- daca sunt mai multe posibilitati pentru valoarea implicita e mai bine sa nu se foloseasca (lizibilitate)
- cand se foloseste un param. implicit nu trebuie sa faca probleme in program

Ambiguitati pentru polimorfism de functii

- erori la compilare
- majoritatea datorita conversiilor implicite

```
int myfunc(double d);  
// ...  
cout << myfunc('c'); // not an error, conversion applied
```

```

#include <iostream>
using namespace std;

float myfunc(float i);
double myfunc(double i);

int main()
{
    cout << myfunc(10.1) << " "; // unambiguous, calls myfunc(double)
    cout << myfunc(10); // ambiguous
    return 0;
}

float myfunc(float i)
{
    return i;
}

double myfunc(double i)
{
    return -i;
}

```

- problema nu e de definire a functiilor myfunc,
- problema apare la apelul functiilor

```

#include <iostream>
using namespace std;

char myfunc(unsigned char ch);
char myfunc(char ch);

int main()
{
    cout << myfunc('c'); // this calls myfunc(char)
    cout << myfunc(88) << " "; // ambiguous

    return 0;
}

char myfunc(unsigned char ch)
{
    return ch-1;
}

char myfunc(char ch)
{
    return ch+1;
}

```

- ambiguitate intre char si unsigned char
- ambiguitate pentru functii cu param. impliciti

```

#include <iostream>
using namespace std;

int myfunc(int i);
int myfunc(int i, int j=1);

int main()
{
    cout << myfunc(4, 5) << " "; // unambiguous
    cout << myfunc(10); // ambiguous
    return 0;
}

int myfunc(int i)
{
    return i;
}

int myfunc(int i, int j)
{
    return i*j;
}

```

// This program contains an error.

```
#include <iostream>
```

```
using namespace std;
```

```
void f(int x);
```

```
void f(int &x); // error
```

```
int main()
```

```
{
```

```
    int a=10;
```

```
    f(a); // error, which f()?
```

```
    return 0;
```

```
}
```

```
void f(int x)
```

```
{
```

```
    cout << "In f(int)\n";
```

```
}
```

```
void f(int &x)
```

```
{
```

```
    cout << "In f(int &)\n";
```

```
}
```

- doua tipuri de apel: prin valoare si prin referinta, ambiguitate!
- mereu eroare de ambiguitate

Supraincercarea operatorilor in C++

- majoritatea operatorilor pot fi supraincarcati
- similar ca la functii
- una din proprietatile C++ care ii confera putere
- s-a facut supraincercarea operatorilor si pentru operatii de I/O (<<,>>)
- supraincercarea se face definind o functie operator: membru al clasei sau nu

functii operator membri ai clasei

```
ret-type class-name::operator#(arg-list)
{
    // operations
}
```

- # este operatorul supraincarcat (+ - * / ++ -- = , etc.)
- de obicei ret-type este tipul clasei, dar avem flexibilitate
- pentru operatori unari arg-list este vida
- pentru operatori binari: arg-list contine un element

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

```
class loc {
    int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {
        longitude = lg;
        latitude = lt;
    }
}
```

```
void show() {
    cout << longitude << " ";
    cout << latitude << "\n";
}
```

```
loc operator+(loc op2);
};
```

```
// Overload + for loc.
```

```
loc loc::operator+(loc op2)
{
    loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    return temp;
}
```

```
int main()
{
    loc ob1(10, 20), ob2( 5, 30);
    ob1.show(); // displays 10 20
    ob2.show(); // displays 5 30
    ob1 = ob1 + ob2;
    ob1.show(); // displays 15 50
}
```

```
return 0;
```

```
}
```

- un singur argument pentru ca avem **this**
- longitude==this->longitude
- obiectul din stanga face apelul la functia operator
 - ob1a chemat operatorul + redefinit in clasa lui ob1

- daca intoarcem acelasi tip de date in operator putem avea expresii
- daca intorceam alt tip nu puteam face
 - putem avea si $ob1 = ob1 + ob2;$
 - $(ob1+ob2).show();$ // displays outcome of $ob1+ob2$
- pentru ca functia show() este definita in clasa lui ob1
- se genereaza un obiect temporar
 - (constructor de copiere)

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

class loc {
    int longitude, latitude;
public:
    loc() {} // needed to construct temporaries
    loc(int lg, int lt) { longitude = lg; latitude = lt; }
    void show() { cout << longitude << " ";
                  cout << latitude << "\n"; }
    loc operator+(loc op2);
    loc operator-(loc op2);
    loc operator=(loc op2);
    loc operator++();
};

// Overload + for loc.
loc loc::operator+(loc op2){
    loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    return temp;}

loc loc::operator-(loc op2){
    loc temp;
    temp.longitude = longitude - op2.longitude;
    temp.latitude = latitude - op2.latitude;
    return temp;}

// Overload assignment for loc.
loc loc::operator=(loc op2){
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this; } // object that generated call

// Overload prefix ++ for loc.
loc loc::operator++(){
    longitude++;
    latitude++;
    return *this;}

int main(){
    loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
    ob1.show();
    ob2.show();
    ++ob1;
    ob1.show(); // displays 11 21
    ob2 = ++ob1;
    ob1.show(); // displays 12 22
    ob2.show(); // displays 12 22
    ob1 = ob2 = ob3; // multiple assignment
    ob1.show(); // displays 90 90
    ob2.show(); // displays 90 90

    return 0;
}
```

- apelul la functia operator se face din obiectul din stanga (pentru operatori binari)
 - din aceasta cauza pentru – avem functia definita asa
- operatorul = face copiere pe variabilele de instanta, intoarce *this
- se pot face atribuiiri multiple (dreapta spre stanga)

Formele prefix si postfix

- am vazut prefix, pentru postfix: definim un parametru int “dummy”

```
// Prefix increment
type operator++( ) {
// body of prefix operator
}
```

```
// Postfix increment
type operator++(int x) {
// body of postfix operator
}
```

supraincarcarea +=, *=, etc.

```
loc loc::operator+=(loc op2)
{
    longitude = op2.longitude + longitude;
    latitude = op2.latitude + latitude;
    return *this;
}
```

restrictii

- nu se poate redefini si precedenta operatorilor
- nu se poate redefini numarul de operanzi
 - rezonabil pentru ca redefinim pentru lizibilitate
 - putem ignora un operand daca vrem
- nu putem avea valori implicite; exceptie pentru ()
- nu putem face overload pe . :: .* ?
- e bine sa facem operatiuni apropiate de intelesul operatorilor respectivi

- Este posibil sa facem o decuplare completa intre intelesul initial al operatorului
 - exemplu: << >>
- mostenire: operatorii (mai putin =) sunt mosteniti de clasa derivata
- clasa derivata poate sa isi redefineasca operatorii

Supraincercarea operatorilor ca functii prieten

- operatorii pot fi definiti si ca functie nemembra a clasei
- o facem functie prietena pentru a putea accesa rapid campurile protejate
- nu avem pointerul “this”
- deci vom avea nevoie de toti operanzii ca parametri pentru functia operator
- primul parametru este operandul din stanga, al doilea parametru este operandul din dreapta

```

#include <iostream>
using namespace std;

class loc {
    int longitude, latitude;
public:
    loc() {} // needed to construct temporaries
    loc(int lg, int lt) {longitude = lg; latitude = lt;}
    void show() { cout << longitude << " ";
                  cout << latitude << "\n";}
    friend loc operator+(loc op1, loc op2); // friend
    loc operator-(loc op2);
    loc operator=(loc op2);
    loc operator++();
};

// Now, + is overloaded using friend function.
loc operator+(loc op1, loc op2)
{
    loc temp;

    temp.longitude = op1.longitude + op2.longitude;
    temp.latitude = op1.latitude + op2.latitude;

    return temp;
}

// Overload - for loc.
loc loc::operator-(loc op2){
    loc temp;
    // notice order of operands
    temp.longitude = longitude - op2.longitude;
    temp.latitude = latitude - op2.latitude;
    return temp;}

// Overload assignment for loc.
loc loc::operator=(loc op2){
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this;} //return obj. that generated call

// Overload ++ for loc.
loc loc::operator++(){
    longitude++;
    latitude++;
    return *this;}

int main(){
    loc ob1(10, 20), ob2( 5, 30);
    ob1 = ob1 + ob2;
    ob1.show();

    return 0;
}

```

Restricții pentru operatorii definiți ca prieten

- nu se pot supraîncărca = () [] sau -> cu funcții prieten
- pentru ++ sau -- trebuie să folosim referințe

functii prieten pentru operatori unari

- pentru ++, -- folosim referinta pentru a transmite operandul
 - pentru ca trebuie sa se modifice si nu avem pointerul this
 - apel prin valoare: primim o copie a obiectului si nu putem modifica operandul (ci doar copia)

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

class loc {
    int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {longitude = lg;latitude = lt;}
    void show() { cout << longitude << " ";
                 cout << latitude << "\n";}

    loc operator=(loc op2);
    friend loc operator++(loc &op);
    friend loc operator--(loc &op);
};

// Overload assignment for loc.
loc loc::operator=(loc op2){
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this; // return object that generated call
}

// Make op-- a friend; use reference.
loc operator--(loc &op)
{
    op.longitude--;
    op.latitude--;
    return op;
}

int main()
{
    loc ob1(10, 20), ob2;
    ob1.show();
    ++ob1;
    ob1.show(); // displays 11 21
    ob2 = ++ob1;
    ob2.show(); // displays 12 22
    --ob2;
    ob2.show(); // displays 11 21

    return 0;
}

// Now a friend; use a reference parameter.
loc operator++(loc &op) {
    op.longitude++;
    op.latitude++;
    return op;}
```

pentru varianta postfix ++ --

- la fel ca la supraincarcarea operatorilor prin functii membru ale clasei: parametru int

```
// friend, postfix version of ++  
friend loc operator++(loc &op, int x);
```

Diferente supraincarcarea prin membrii sau prieteni

- de multe ori nu avem diferente,
 - atunci e indicat sa folosim functii membru
- uneori avem insa diferente: pozitia operanzilor
 - pentru functii membru operandul din stanga apeleaza functia operator supraincarcata
 - daca vrem sa scriem expresie: 100+ob;
probleme la compilare=> functii prieten

- in aceste cazuri trebuie sa definim doua functii de supraincarcare:
 - `int + tipClasa`
 - `tipClasa + int`

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

class loc {
    int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {longitude = lg; latitude = lt;}
    void show() { cout << longitude << " ";
                  cout << latitude << "\n";}
    friend loc operator+(loc op1, int op2);
    friend loc operator+(int op1, loc op2);
};

// + is overloaded for loc + int.
loc operator+(loc op1, int op2){
    loc temp;
    temp.longitude = op1.longitude + op2;
    temp.latitude = op1.latitude + op2;
    return temp;}

// + is overloaded for int + loc.
loc operator+(int op1, loc op2){
    loc temp;
    temp.longitude = op1 + op2.longitude;
    temp.latitude = op1 + op2.latitude;
    return temp;}

int main()
{
    loc ob1(10, 20), ob2( 5, 30), ob3(7, 14);

    ob1.show();
    ob2.show();
    ob3.show();
    ob1 = ob2 + 10; // both of these
    ob3 = 10 + ob2; // are valid
    ob1.show();
    ob3.show();

    return 0;
}
```

supraincarcarea new si delete

- supraincarcare op. de folosire memorie in mod dinamic pentru cazuri speciale

```
// Allocate an object.  
void *operator new(size_t size)  
{  
/* Perform allocation. Throw bad_alloc on failure.  
Constructor called automatically. */  
return pointer_to_memory;
```

- `size_t`: predefinit `Constructor called automatically. */`
`return pointer_to_memory;`
- pentru `new`: constructorul este chemat automat `}`
- pentru `delete`: destructorul este chemat automat `// Delete an object.`
`void operator delete(void *p)`
- supraincarcare la nivel de clasa sau globala `{`
`@ Free memory pointed to by p`

```
// Delete an object.
void operator delete(void *p)
{
    // Free memory pointed to by p
    Destructeur called automatically. */
}
```

#include <iostream>		// delete overloaded relative to loc.
#include <cstdlib>		void loc::operator delete(void *p){
#include <new>		cout << "In overloaded delete.\n";
using namespace std;		free(p);
class loc {		}
int longitude, latitude;		
public:	• In overloaded new.	
loc() {}		
loc(int lg, int lt) {longitude = lg; latitude = lt;}	• In overloaded new.);
void show() { cout << longitude << " " << latitude << " "; }	• 10 20	error for p1.\n";
void *operator new(size_t size) { return malloc(size); }	• -10 -20	-20);
void operator delete(void *p) { free(p); }	• In overloaded delete.	{
	• In overloaded delete.	error for p2.\n";
// new overloaded relative to loc.		
void *loc::operator new(size_t size) { return malloc(size); }		
void *p;		p = new loc(10, 20);
cout << "In overloaded new.\n";		delete p1;
p = malloc(size);		delete p2;
if(!p) { bad_alloc ba; throw ba; }		return 0;
return p;		}
}		

- daca new sau delete sunt folositi pentru alt tip de date in program, versiunile originale sunt folosite
- se poate face overload pe new si delete la nivel global
 - se declara in afara oricarei clase
 - pentru new/delete definiti si global si in clasa, cel din clasa e folosit pentru elemente de tipul clasei, si in rest e folosit cel redefinit global

```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
class loc {
    int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {longitude = lg;latitude = lt;}
    void show() {cout << longitude << " ";
                cout << latitude << "\n";}
};

// Global new
void *operator new(size_t size)
{
    void *p;
    p = malloc(size);
    if(!p) {
        bad_alloc ba;
        throw ba;
    }
    return p;}

```

```
// Global delete
void operator delete(void *p)
{ free(p); }

int main(){
    loc *p1, *p2;
    float *f;
    try {p1 = new loc (10, 20);
        } catch (bad_alloc xa) {
            cout << "Allocation error for p1.\n";
            return 1;        }
    try {p2 = new loc (-10, -20);
        } catch (bad_alloc xa) {
            cout << "Allocation error for p2.\n";
            return 1;        }
    try {f = new float; // uses overloaded new, too
        } catch (bad_alloc xa) {
            cout << "Allocation error for f.\n";
            return 1;        }
    *f = 10.10F; cout << *f << "\n";
    p1->show(); p2->show();
    delete p1; delete p2; delete f;
    return 0; }

```

new si delete pentru array-uri

- facem overload de doua ori

```
// Allocate an array of objects.  
void *operator new[](size_t size)  
{  
    /* Perform allocation. Throw bad_alloc on failure.  
    Constructor for each element called automatically. */  
    return pointer_to_memory;  
}  
  
// Delete an array of objects.  
void operator delete[](void *p)  
{  
    /* Free memory pointed to by p.  
    Destructor for each element called automatically.  
    */  
}
```

supraincarcarea []

- trebuie sa fie functii membru, (nestate)
- nu pot fi functii prieten
- este considerat operator binar
- o[3] se transforma in
type class-name::operator[](int i)
- o.operator[](3)
{
 // ...
}

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

class atype {
    int a[3];
public:
    atype(int i, int j, int k) { a[0] = i; a[1] = j; a[2] = k; }
    int operator[](int i) { return a[i]; }
};

int main()
{
    atype ob(1, 2, 3);
    cout << ob[1]; // displays 2
    return 0;
}
```

- operatorul [] poate fi folosit si la stanga unei atribuirii (obiectul intors este atunci referinta)

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

```
class atype {
    int a[3];
public:
    atype(int i, int j, int k) { a[0] = i; a[1] = j; a[2] = k; }
    int &operator[](int i) { return a[i]; }
};
```

```
int main()
{
    atype ob(1, 2, 3);
    cout << ob[1]; // displays 2
    cout << " ";
    ob[1] = 25; // [] on left of =
    cout << ob[1]; // now displays 25
    return 0;
}
```

- putem in acest fel verifica array-urile
- exemplul urmator

// A safe array example.

```
#include <iostream>
```

```
#include <cstdlib>
```

```
using namespace std;
```

```
class atype {
    int a[3];
public:
    atype(int i, int j, int k) {a[0] = i;a[1] = j;a[2] = k;}
    int &operator[](int i);
};
```

// Provide range checking for atype.

```
int &atype::operator[](int i)
{
    if(i<0 || i> 2) {
        cout << "Boundary Error\n";
        exit(1);
    }
    return a[i];
}
```

```
int main()
```

```
{
    atype ob(1, 2, 3);
    cout << ob[1]; // displays 2
    cout << " ";
    ob[1] = 25; // [] appears on left
    cout << ob[1]; // displays 25
    ob[3] = 44;
        // generates runtime error, 3 out-of-range
    return 0;
}
```

supraincarcarea ()

- nu creem un nou fel de a apela functii
- definim un mod de apel de functii cu numar arbitrar de parametri

```
double operator()(int a, float f, char *s);
```

```
O(10, 23.34, "hi");
```

echivalent cu `O.operator()(10, 23.34, "hi");`

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

```
class loc {
    int longitude, latitude;
public:
```

```
10 20
```

```
7 8
```

```
11 11
```

```
};
```

```
// Overload () for loc.
```

```
loc loc::operator()(int i, int j)
```

```
{
    longitude = i;
    latitude = j;
    return *this;
}
```

```
// Overload + for loc.
```

```
loc loc::operator+(loc op2)
```

```
{
    loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    return temp;
}
```

```
int main()
```

```
{
    loc ob1(10, 20), ob2(1, 1);
    ob1.show();
    ob1(7, 8); // can be executed by itself
    ob1.show();
    ob1 = ob2 + ob1(10, 10);
                // can be used in expressions
    ob1.show();
    return 0;
}
```

overload pe ->

- operator unar
- obiect->element
 - obiect genereaza apelul
 - element trebuie sa fie accesibil
 - intoarce un pointer catre un obiect din clasa

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

class myclass {
public:
    int i;
    myclass *operator->() {return this;}
};

int main()
{
    myclass ob;
    ob->i = 10; // same as ob.i
    cout << ob.i << " " << ob->i;
    return 0;
}
```


supraincarcarea operatorului ,

- operator binar
- ar trebui ignorate toate valorile mai puțin a celui mai din dreapta operand

```
#include <iostream>    10 20    // Overload + for loc
using namespace std;   5 30    loc loc::operator+(loc op2)
                        1 1      {
class loc {             10 60    loc temp;
    int longitude, latitude; 1 1    temp.longitude = op2.longitude + longitude;
public:                 1 1    temp.latitude = op2.latitude + latitude;
    loc() {}            1 1    return temp;
    loc(int lg, int lt) {longitude = lg;latitude = lt;} }
    void show() {cout << longitude << " ";
                        cout << latitude << "\n";}
    loc operator+(loc op2);
    loc operator,(loc op2);
};

// overload comma for loc
loc loc::operator,(loc op2)
{
    loc temp;
    temp.longitude = op2.longitude;
    temp.latitude = op2.latitude;
    cout << op2.longitude << " " << op2.latitude << "\n";
    return temp;
}

int main()
{
    loc ob1(10, 20), ob2( 5, 30), ob3(1, 1);
    ob1.show();
    ob2.show();
    ob3.show();
    cout << "\n";
    ob1 = (ob1, ob2+ob2, ob3);
    ob1.show(); // displays 1 1, the value of ob3
    return 0;
}
```

Cursul de programare orientata pe obiecte

Serial3

Saptamana 6, 25 martie 2014

Andrei Paun

Cuprinsul cursului: 25 martie 2014

- exceptii si management de exceptii
- Pointeri si referinte in C si C++

exceptii in C++

- automatizarea procesarii erorilor
- try, catch throw
- block try arunca exceptie cu throw care este prinsa cu catch
- dupa ce este prinsa se termina executia din blocul catch si se da controlul “mai sus, nu se revine la locul unde s-a facut throw (nu e apel de functie)

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

- care bloc catch este executat este dat de tipul expresiei
- orice tip de date poate fi folosit ca argument pentru catch
- daca nu este generata exceptie, nu se executa nici un bloc catch
- generare de exceptie: throw exc;

- daca se face throw si nu exista un bloc try din care a fost aruncata exceptia sau o functie apelata dintr-un bloc try: eroare
- daca nu exista un catch care sa fie asociat cu throw-ul respectiv (tipuri de date egale) atunci programul se termina prin terminate()
- terminate() poate sa fie redefinita sa faca altceva

// A simple exception handling example.

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

```
int main()
{
    cout << "Start\n";
    try { // start a try block
        cout << "Inside try block\n";
        throw 100; // throw an error
        cout << "This will not execute";
    }
    catch (int i) { // catch an error
        cout << "Caught an exception -- value is: ";
        cout << i << "\n";
    }
    cout << "End";
    return 0;
}
```

Start

Inside try block

Caught an exception -- value is: 100

End

//

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

```
int main()
{
    cout << "Start\n";
    try { // start a try block
        cout << "Inside try block\n";
        throw 100; // throw an error
        cout << "This will not execute";
    }
    catch (double i) { // catch an error
        cout << "Caught an exception -- value is: ";
        cout << i << "\n";
    }
    cout << "End";
    return 0;
}
```

Start

Inside try block

Abnormal program termination

```
/* Throwing an exception from a function outside the  
try block.*/
```

```
#include <iostream>
```

```
using namespace std;
```

```
void Xtest(int test)
```

```
{ cout << "Inside Xtest, test is: " << test << "\n";  
  if(test) throw test;}
```

```
int main()
```

```
{ cout << "Start\n";
```

```
  try { // start a try block
```

```
    cout << "Inside try block\n";
```

```
    Xtest(0);
```

```
    Xtest(1);
```

```
    Xtest(2);
```

```
  }
```

```
  catch (int i) { // catch an error
```

```
    cout << "Caught an exception -- value is: ";
```

```
    cout << i << "\n";
```

```
  }
```

```
  cout << "End";
```

```
  return 0;
```

```
}
```

Start

Inside try block

Inside Xtest, test is: 0

Inside Xtest, test is: 1

Caught an exception -- value is: 1

End

```
#include <iostream>
```

```
using namespace std;
```

```
// Localize a try/catch to a function.
```

```
void Xhandler(int test)
```

```
{ try{
```

```
    if(test) throw test;
```

```
  }
```

```
  catch(int i) {
```

```
    cout << "Caught Exception #: " << i << '\n';
```

```
  }
```

```
}
```

```
int main()
```

```
{ cout << "Start\n";
```

```
  Xhandler(1);
```

```
  Xhandler(2);
```

```
  Xhandler(0);
```

```
  Xhandler(3);
```

```
  cout << "End";
```

```
  return 0;
```

```
}
```

Start

Caught Exception #: 1

Caught Exception #: 2

Caught Exception #: 3

End

- dupa prima exceptie programul se continua
- nu ca mai sus
- try/catch local

- instructiunile catch sunt verificate in ordinea in care sunt scrise, primul de tipul erorii este folosit

- aruncarea de erori din clase de baza si derivate
- un catch pentru tipul de baza va fi executat pentru un obiect aruncat de tipul derivat
- sa se puna catch-ul pe tipul derivat primul si apoi catchul pe tipul de baza

// Catching derived classes.

```
#include <iostream>
```

```
using namespace std;
```

```
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";
```

```
    }
```

```
    return 0;
```

```
}
```

// This example catches all exceptions.

```
#include <iostream>
```

```
using namespace std;
```

```
void Xhandler(int test)
```

```
{ try{
```

```
    if(test==0) throw test; // throw int
```

```
    if(test==1) throw 'a'; // throw char
```

```
    if(test==2) throw 123.23; // throw double
```

```
    }
```

```
    catch(...) { // catch all exceptions
```

```
        cout << "Caught One!\n";
```

```
    }
```

```
}
```

```
int main()
```

```
{ cout << "Start\n";
```

```
    Xhandler(0);
```

Start

```
    Xhandler(1);
```

Caught One!

```
    Xhandler(2);
```

Caught One!

```
    cout << "End";
```

Caught One!

```
    return 0;
```

End

```
}
```

- catch p
catch

- se poate specifica ce exceptii arunca o functie
- se restrictioneaza tipurile de exceptii care se pot arunca din functie
- un alt tip nespecificat termina programul:
 - apel la unexpected() care apeleaza abort()
 - se poate redefini

`void Xhandler(int test) throw(int, char, double)`

re-aruncarea unei exceptii

- `throw;` // fara exceptie din catch
- pentru a procesa eroarea in mai multe handlers
- evident avem try in try

- in <exception> avem

void terminate();

void unexpected();

terminate: cand nu exista catch compatibil

unexpected: cand o functie nu da voie sa fie
aruncat tipul respectiv de exceptie

XVIII. Spuneți dacă programul de mai jos este corect. În caz afirmativ, spuneți ce afișează pentru o valoare întreagă citită egală cu 7, în caz negativ spuneți de ce nu este corect.

```
#include <iostream.h>
float f(int y)
{ try
  { if (y%2) throw y/2;
  }
  catch (int i)
  { if (i%2) throw;
    cout<<"Numarul "<<i<<" nu e bun!"<<endl;
  }
  return y/2;
}
int main()
{ int x;
  try
  { cout<<"Da-mi un numar intreg: ";
    cin>>x;
    if (x) f(x);
    cout<<"Numarul "<<x<<" nu e bun!"<<endl;
  }
  catch (int i)
  { cout<<"Numarul "<<i<<" e bun!"<<endl;
  }
  return 0;
}
```

Pointeri in C

- Recapitulare
- &, *, array
- Operatii pe pointeri: =, ++, --, +int, -
- Pointeri catre pointeri, pointeri catre functii
- Alocare dinamica: malloc, free
- Diferente cu C++

pointerii

- O variabila care tine o adresa din memorie
- Are un tip, compilatorul stie tipul de date catre care se pointeaza
- Operatiile aritmetice tin cont de tipul de date din memorie
- `Pointer ++ == pointer+sizeof(tip)`
- Definitie: `tip *nume_pointer;`
 - Merge si `tip* nume_pointer;`

Operatori pe pointeri

- *, &, schimbare de tip
- *== “la adresa”
- &==“adresa lui”

```
int i=7, *j;
```

```
j=&i;
```

```
*j=9;
```

```
#include <stdio.h>
int main(void)
{
    double x = 100.1, y;
    int *p;
    /* The next statement causes p (which is an
    integer pointer) to point to a double. */
    p = (int *)&x;
    /* The next statement does not operate as
    expected. */
    y = *p;
    printf("%f", y); /* won't output 100.1 */
    return 0;
}
```

- Schimbarea de tip nu e controlata de compiler
- In C++ conversiile trebuiesc facute cu schimbarea de tip

Aritmetica pe pointeri

- `pointer++`; `pointer--`;
- `pointer+7`;
- `pointer-4`;
- `pointer1-pointer2`; intoarce un intreg
- comparatii: `<`, `>`, `==`, etc.

pointeri si array-uri

- numele array-ului este pointer
- `lista[5]==*(lista+5)`
- array de pointeri, numele listei este un pointer catre pointeri (dubla indirectare)
- `int **p`; (dubla indirectare)

alocare dinamica

- `void *malloc(size_t bytes);`
 - aloca in memorie dinamic bytes si intoarce pointer catre zona respectiva

`char *p;`

`p=malloc(100);`

intoarce null daca alocarea nu s-a putut face

pointer `void*` este convertit AUTOMAT la orice tip

- diferenta la C++: trebuie sa se faca schimbare de tip dintre `void*` in `tip*`

`p=(char *) malloc(100);`

`sizeof`: a se folosi pentru portabilitate

a se verifica daca alocarea a fost fara eroare (daca se intoarce null sau nu)

`if (!p) ...`

eliberarea de memorie alocata dinamic

`void free(void *p);`

unde p a fost alocat dinamic cu `malloc()`

a nu se folosi cu argumentul p invalid pentru
ca rezulta probleme cu lista de alocare
dinamica

C++: Array-uri de obiecte

- o clasa da un tip
- putem crea array-uri cu date de orice tip (inclusiv obiecte)
- se pot defini neinitializate sau initializate

`clasa lista[10];`

sau

`clasa lista[10]={ 1,2,3,4,5,6,7,8,9,0};`

pentru cazul initializat dat avem nevoie de
constructor care primeste un parametru intreg.

```

#include <iostream>
using namespace std;

class cl {
    int i;
public:
    cl(int j) { i=j; } // constructor
    int get_i() { return i; }
};

int main()
{
    cl ob[3] = {1, 2, 3}; // initializers
    int i;

    for(i=0; i<3; i++)
        cout << ob[i].get_i() << "\n";
    return 0;
}

```

- initializare pentru constructori cu mai multi parametri

```

clasa lista[3]={clasa(1,5), clasa(2,4), clasa(3,3)};

```

- pentru definirea listelor de obiecte neinitializate: constructor fara parametri
- daca in program vrem si initializare si neinitializare: overload pe constructor (cu si fara parametri)

pointeri catre obiecte

- obiectele sunt in memorie
- putem avea pointeri catre obiecte
- &obiect;
- accesarea membrilor unei clase:
-> in loc de .


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

```
class cl {
    int i;
public:
    cl(int j) { i=j; }
    int get_i() { return i; }
};
```

```
int main()
{
    cl ob(88), *p;
    p = &ob; // get address of ob
    cout << p->get_i(); // use -> to call get_i()
    return 0;
}
```

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

```
class cl {
public:
    int i;
    cl(int j) { i=j; }
};
```

```
int main()
{
    cl ob(1);
    int *p;
    p = &ob.i; // get address of ob.i
    cout << *p; // access ob.i via p
    return 0;
}
```

- in C++ tipurile pointerilor trebuie sa fie la fel

```
int *p;
```

```
float *q;
```

```
p=q; //eroare
```

se poate face cu schimbarea de tip (type casting) dar
iesim din verificarile automate facute de C++

pointerul **this**

- orice functie membru are pointerul **this** (definit ca argument implicit) care arata catre obiectul asociat cu functia respectiva
- (pointer catre obiecte de tipul clasei)
- functiile prieten nu au pointerul this
- functiile statice nu au pointerul this

```
#include <iostream>
using namespace std;
class pwr {
    double b;
    int e;
    double val;
public:
    pwr(double base, int exp);
    double get_pwr() { return this->val; }
};

pwr::pwr(double base, int exp)
{
    b = base;
    e = exp;
    val = 1;
    if(exp==0) return;
    for( ; exp>0; exp--) val = val * b;
}

int main()
{
    pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);
    cout << x.get_pwr() << " ";
    cout << y.get_pwr() << " ";
    cout << z.get_pwr() << "\n";
    return 0;
}
```

pointeri catre clase derivate

- clasa de baza B si clasa derivata D
- un pointer catre B poate fi folosit si cu D;

```
B *p, o(1);
```

```
D oo(2);
```

```
p=&o;
```

```
p=&oo;
```

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

class base {
    int i;
public:
    void set_i(int num) { i=num; }
    int get_i() { return i; }
};

class derived: public base {
    int j;
public:
    void set_j(int num) { j=num; }
    int get_j() { return j; }
};
```

```
int main()
{
    base *bp;
    derived d;
    bp = &d; // base pointer points to derived object
             // access derived object using base pointer
    bp->set_i(10);
    cout << bp->get_i() << " ";
    /* The following won't work. You can't access elements of
       a derived class using a base class pointer.

    ((derived *)bp)->set_j(88);
    cout << ((derived *)bp)->get_j();
    */
    return 0;
}
```

pointeri catre clase derivate

- de ce merge si pentru clase derivate?
 - pentru ca acea clasa derivata functioneaza ca si clasa de baza plus alte detalii
- aritmetica pe pointeri: nu functioneaza daca incrementam un pointer catre baza si suntem in clasa derivata
- se folosesc pentru polimorfism la executie (functii virtuale)

// This program contains an error.

```
#include <iostream>
```

```
using namespace std;
```

```
class base {
```

```
    int i;
```

```
public:
```

```
    void set_i(int num) { i=num; }
```

```
    int get_i() { return i; }
```

```
};
```

```
class derived: public base {
```

```
    int j;
```

```
public:
```

```
    void set_j(int num) {j=num;}
```

```
    int get_j() {return j;}
```

```
};
```

```
int main()
```

```
{
```

```
    base *bp;
```

```
    derived d[2];
```

```
    bp = d;
```

```
    d[0].set_i(1);
```

```
    d[1].set_i(2);
```

```
    cout << bp->get_i() << " ";
```

```
    bp++; // relative to base, not derived
```

```
    cout << bp->get_i(); // garbage value displayed
```

```
    return 0;
```

```
}
```

pointeri catre membri in clase

- pointer catre membru
- nu sunt pointeri normali (catre un membru dintr-un obiect) ci specifica un offset in clasa
- nu putem sa aplicam . si ->
- se folosesc .* si ->*

```
#include <iostream>
using namespace std;
class cl {
public:
    cl(int i) { val=i; }
    int val;
    int double_val() { return val+val; }
};

int main()
{
    int cl::*data; // data member pointer
    int (cl::*func)(); // function member pointer
    cl ob1(1), ob2(2); // create objects
    data = &cl::val; // get offset of val
    func = &cl::double_val; // get offset of double_val()
    cout << "Here are values: ";
    cout << ob1.*data << " " << ob2.*data << "\n";
    cout << "Here they are doubled: ";
    cout << (ob1.*func)() << " ";
    cout << (ob2.*func)() << "\n";
    return 0;
}
```

```
#include <iostream>
using namespace std;
class cl {
public: cl(int i) { val=i; }
        int val;
        int double_val() { return val+val; };
};

int main(){
    int cl::*data; // data member pointer
    int (cl::*func)(); // function member pointer
    cl ob1(1), ob2(2), *p1, *p2;
    p1 = &ob1; // access objects through a pointer
    p2 = &ob2;
    data = &cl::val; // get offset of val
    func = &cl::double_val;
    cout << "Here are values: ";
    cout << p1->*data << " " << p2->*data << "\n";
    cout << "Here they are doubled: ";
    cout << (p1->*func)() << " ";
    cout << (p2->*func)() << "\n";
    return 0;
}
```

```
int cl::*d;  
int *p;  
cl o;
```

```
p = &o.val // this is address of a specific val  
d = &cl::val // this is offset of generic val
```

- pointeri la membri nu sunt folositi decat rar in cazuri speciale

parametri referinta

- nou la C++
- la apel prin valoare se adauga si apel prin referinta la C++
- nu mai e nevoie sa folosim pointeri pentru a simula apel prin referinta, limbajul ne da acest lucru
- sintaxa: in functie & inaintea parametrului formal

// Manually: call-by-reference using a pointer.

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

```
void neg(int *i);
```

```
int main()
{
    int x;
    x = 10;
    cout << x << " negated is ";
    neg(&x);
    cout << x << "\n";
    return 0;
}
```

```
void neg(int *i)
{
    *i = -*i;
}
```

// Use a reference parameter.

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

```
void neg(int &i); // i now a reference
```

```
int main()
{
    int x;
    x = 10;
    cout << x << " negated is ";
    neg(x); // no longer need the & operator
    cout << x << "\n";
    return 0;
}
```

```
void neg(int &i)
{
    i = -i; // i is now a reference, don't need *
}
```

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

```
void swap(int &i, int &j);
```

```
int main()
{
    int a, b, c, d;
    a = 1;    b = 2;    c = 3;    d = 4;
    cout << "a and b: " << a << " " << b << "\n";
    swap(a, b); // no & operator needed
    cout << "a and b: " << a << " " << b << "\n";
    cout << "c and d: " << c << " " << d << "\n";
    swap(c, d);
    cout << "c and d: " << c << " " << d << "\n";
    return 0;
}
```

```
void swap(int &i, int &j)
{
    int t;
    t = i; // no * operator needed
    i = j;
    j = t;
}
```

referinte catre obiecte

- daca transmitem obiecte prin apel prin referinta la functii nu se mai creeaza noi obiecte temporare, se lucreaza direct pe obiectul transmis ca parametru
- deci copy-constructorul si destructorul nu mai sunt apelate
- la fel si la intoarcerea din functie a unei referinte

```
#include <iostream>
using namespace std;
class cl {
    int id;
public:
    int i;
    cl(int i);
    ~cl(){ cout << "Destructing " << id << "\n"; }
    void neg(cl &o) { o.i = -o.i; } // no temporary created
};

cl::cl(int num)
{   cout << "Constructing " << num << "\n";
    id = num;
}

int main()
{   cl o(1);
    o.i = 10;
    o.neg(o);
    cout << o.i << "\n";
    return 0;
}
```

Constructing 1
-10
Destructing 1


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

intoarcere de referinte

```
char &replace(int i); // return a reference
```

```
char s[80] = "Hello There";
```

```
int main()
{
    replace(5) = 'X'; // assign X to space after Hello
    cout << s;
    return 0;
}
```

```
char &replace(int i)
{
    return s[i];
}
```

- putem face atribuiiri catre apel de functie
- `replace(5)` este un element din `s` care se schimba
- e nevoie de atentie ca obiectul referit sa nu iasa din scopul de vizibilitate

referinte independente

- nu e asociat cu apelurile de functii
- se creeaza un alt nume pentru un obiect
- referintele independente trebuiesc initializate la definire pentru ca ele nu se schimba in timpul programului

```

#include <iostream>
using namespace std;

int main()
{
    int a;
    int &ref = a; // independent reference
    a = 10;
    cout << a << " " << ref << "\n";
    ref = 100;
    cout << a << " " << ref << "\n";
    int b = 19;
    ref = b; // this puts b's value into a
    cout << a << " " << ref << "\n";
    ref--; // this decrements a
    // it does not affect what ref refers to
    cout << a << " " << ref << "\n";
    return 0;
}

```

10 10
100 100
19 19
18 18

referinte catre clase derivate

- putem avea referinte definite catre clasa de baza si apelata functia cu un obiect din clasa derivata
- exact la la pointeri

Alocare dinamica in C++

- new, delete
- operatori nu functii
- se pot folosi inca malloc() si free() dar vor fi deprecated in viitor

operatorii new, delete

- new: aloca memorie si intoarce un pointer la inceputul zonei respective
- delete: sterge zona respectiva de memorie

```
p= new tip;  
delete p;
```

la eroare se “arunca” exceptia bad_alloc din <new>

```

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

int main()
{
    int *p;
    try {
        p = new int; // allocate space for an int
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }
    *p = 100;
    cout << "At " << p << " ";
    cout << "is the value " << *p << "\n";
    delete p;
    return 0;
}

```

```

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

int main()
{
    int *p;
    try {
        p = new int(100); // initialize with 100
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }
    cout << "At " << p << " ";
    cout << "is the value " << *p << "\n";
    delete p;
    return 0;
}

```

alocare de array-uri

```

p_var = new array_type [size];

delete [ ] p_var;

```

```

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

int main()
{
    int *p, i;
    try {
        p = new int [10]; // allocate 10 integer array
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }
    for(i=0; i<10; i++ )
        p[i] = i;
    for(i=0; i<10; i++)
        cout << p[i] << " ";
    delete [ ] p; // release the array
    return 0;
}

```

alocare de obiecte

- cu new
- dupa creare, new intoarce un pointer catre obiect
- dupa creare se executa constructorul obiectului
- cand obiectul este sters din memorie (delete) se executa destructorul

obiecte create dinamic cu constructori parametrizati

```
class balance {...}  
...  
  
balance *p;  
// this version uses an initializer  
try {  
    p = new balance (12387.87, "Ralph Wilson");  
} catch (bad_alloc xa) {  
    cout << "Allocation Failure\n";  
    return 1;  
}
```

- array-uri de obiecte alocate dinamic
 - nu se pot initializa
 - trebuie sa existe un constructor fara parametri
 - delete poate fi apelat pentru fiecare element din array

- new si delete sunt operatori
- pot fi suprascrisi pentru o anumita clasa
- pentru argumente suplimentare exista o forma speciala
 - `p_var = new (lista_argumente) tip;`
- exista forma nothrow pentru new: similar cu malloc:
 - `p=new(nothrow) int[20]; // intoarce null la eroare`

Cursul de programare orientata pe obiecte

Seria 13

Saptamana 7, 1 aprilie 2014

Andrei Paun

Cuprinsul cursului: 1 aprilie 2014

- Mostenirea in C++
- Polimorfism la executie prin functii virtuale

Mostenirea in C++

- in C: copiere de cod si re-utilizare
 - nu a functionat prea bine istoric
- in C++: re-utilizare de cod prin mostenire
 - mostenire de clase create anterior
 - clasele mostenite de multe ori sunt create de alti programatori si verificate deja (codul “merge”)
 - marea putere: modificam codul dar garantam ca portiunile de cod vechi functioneaza

- clasa de baza si clasa derivata
- pentru functii tipul derivat poate substitui tipul de baza
- sintaxa: pentru o clasa definita deja “baza”
class derivata: public baza { definitii noi }


```

#include <iostream>
using namespace std;

class X {
    int i;
public:
    X() { i = 0; }
    void set(int ii) { i = ii; }
    int read() const { return i; }
    int permute() { return i = i * 47; }
};

class Y : public X {
    int i; // Different from X's i
public:
    Y() { i = 0; }
    int change() {
        i = permute(); // Different name call
        return i;
    }
    void set(int ii) {
        i = ii;
        X::set(ii); // Same-name function call
    }
};

int main() {
    cout << "sizeof(X) = " << sizeof(X) << endl;
    cout << "sizeof(Y) = "
        << sizeof(Y) << endl;
    Y D;
    D.change();
    // X function interface comes through:
    D.read();
    D.permute();
    // Redefined functions hide base versions:
    D.set(12);
}

```

- la mostenire: toti membrii “private” ai clasei de baza sunt privati si neaccesibili in clasa derivata, folosesc spatiu, doar ca nu putem sa ii folosim
- daca se face mostenire cu “public” toti membrii publici ai clasei de baza sunt publici in clasa derivata,
- mostenire cu “private” toti membrii publici ai bazei devin privati (si accesibili) in derivata
- aproape tot timpul se face mostenire cu “public”
 - daca nu se precizeaza specificatorul de mostenire, default este PRIVATE

- In Java nici nu se poate “transforma” un membru public in private
- am vazut ca putem redefini functii (set) sau adauga noi functii si variabile
- daca vroiam sa chemam din derivata::set pe baza::set trebuia sa folosim operatorul de rezolutie de scop ::

Initializare de obiecte

- probleme: daca avem obiecte ca variabile de instanta ai unei clase sau la mostenire
 - de obicei ascundem informatiile, deci sunt campuri neaccesibile pentru constructorul clasei curenate (derivata)
 - raspuns: se cheama intai constructorul pentru baza si apoi pentru clasa derivata

lista de initializare pentru constructori

- pentru constructorul din clasa derivata care mosteneste pe baza

derivata:: derivata(int i): baza(i) { ... }

- spunem astfel ca acest constructor are un param. intreg care e transmis mai departe catre constructorul din baza

- lista de initializare se poate folosi si pentru obiecte incluse in clasa respectiva
 - fie un obiect gigi de tip Student in clasa Derivata
 - pentru apelarea constructorului pentru gigi cu un param. de initializare i
- Derivata::Derivata(int i): Baza(i), gigi(i+3)
{ ... }

pseudo-constructor pentru tipuri de baza

```
class X {
    int i;
    float f;
    char c;
    char* s;
public:
    X() : i(7), f(1.4), c('x'), s("howdy") {}
};

int main() {
    X x;
    int i(100); // Applied to ordinary definition
    int* ip = new int(47);
}
```

ordinea chemarii constructorilor si destructorilor

- constructorii sunt chemati in ordinea definirii obiectelor ca membri ai clasei si in ordinea mostenirii:
 - radacina arborelui de mostenire este primul constructor chemat, constructorii din obiectele membru in clasa respectiva sunt chemati in ordinea definirii
 - apoi se merge pe urmatorul nivel in ordinea mostenirii
- destructorii sunt chemati in ordinea inversa a constructorilor

```

#include <iostream>
using namespace std;

class cls
{
    int x;
public: cls(int i=0) {cout << "Inside constructor 1" << endl; x=i; }
    ~cls(){ cout << "Inside destructor 1" << endl;} };

class cls2
{
    int x;
    cls xx;
public: cls2(int i=0) {cout << "Inside constructor 2" << endl; x=i; }
    ~cls2(){ cout << "Inside destructor 2" << endl;} };

class cls3
{
    int x;
    cls2 yy;
    cls3 zz;
public: cls3(int i=0) {cout << "Inside constructor 3" << endl; x=i; }
    ~cls3(){ cout << "Inside destructor 3" << endl;} };

main()
{
    cls3 s;
}

```

```

Inside constructor 1
Inside constructor 2
Inside constructor 1
Inside constructor 3
Inside destructor 3
Inside destructor 1
Inside destructor 2
Inside destructor 1

```

particularitati la functii

- constructorii si destructorii nu sunt mosteniti (se redefiniesc noi constr. si destr. pentru clasa derivata)
- similar operatorul = (un fel de constructor)

mostenire cu specificatorul private

- toate componentele private din clasa de baza sunt ascunse in clasa derivata
- componentele public si protected sunt acum private si accesibile in derivata
- nu mai putem trata un obiect de tip derivat ca un obiect din clasa de baza!!!
- acelasi efect cu definirea unui obiect de tip baza in interiorul clasei noi (fara mostenire)

- uneori vrem doar portiuni din interfata bazei sa fie publice
- putem mosteni cu private si apoi in zona de definitie pentru clasa derivata spunem ce componente vrem sa le tinem publice:

```
derivata{
...
public:
    baza::functie1();
    baza::functie2();
}
```

```
class Pet {
public:
    char eat() const { return 'a'; }
    int speak() const { return 2; }
    float sleep() const { return 3.0; }
    float sleep(int) const { return 4.0; }
};

class Goldfish : Pet { // Private inheritance
public:
    Pet::eat; // Name publicizes member
    Pet::sleep; // Both overloaded members exposed
};

int main() {
    Goldfish bob;
    bob.eat();
    bob.sleep();
    bob.sleep(1);
    //! bob.speak();// Error: private member function
}
```

specificatorul protected

- este între private și public
- secțiuni definite ca protected sunt similare ca definire cu private (sunt ascunse de restul programului)*
- cu singura observație * sunt accesibile la moștenire

- în proiecte mari avem nevoie de acces la zone “protejate” din clase derivate
- cel mai bine este ca variabilele de instanță să fie PRIVATE și funcții care le modifică să fie protected
- în acest fel va mențineți dreptul de a implementa în alt fel proprietățile clasei respective


```

#include <fstream>
using namespace std;

class Base {
    int i;
protected:
    int read() const { return i; }
    void set(int ii) { i = ii; }
public:
    Base(int ii = 0) : i(ii) {}
    int value(int m) const { return m*i; }
};

class Derived : public Base {
    int j;
public:
    Derived(int jj = 0) : j(jj) {}
    void change(int x) { set(x); }
};

int main() {
    Derived d;
    d.change(10);
}

```

- mostenire: derivata1 din baza (public)
- derivata2 din derivata1 (public)
- daca in baza avem zone “protected” ele sunt transmise si in derivata1,2 tot ca protected
- mostenire derivata1 din baza (private)
atunci zonele protected devin private in derivata1 si neaccesibile in derivata2

Mostenire de tip protected

- `class derivata1: protected baza { ... };`
- atunci toti membrii publici si protected din baza devin protected in derivata.
- nu se prea foloseste, inclusa in limbaj pentru completitudine

Mostenire

- Daca in clasa derivata se redefineste o functie suprascrisa (overloaded) in baza, toate variantele functiei respective sunt ascunse (nu mai sunt accesibile in derivata)
- Putem avea polimorfism de nume in clasa derivata pe o functie overload din baza (dar semnatura noii functii trebuie sa fie distincta de cele din baza)

```

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

class Base {
public:
    int f() const {
        cout << "Base::f()\n";
        return 1;
    }
    int f(string) const { return 1; }
    void g() {}
};

class Derived1 : public Base {
public:
    void g() const {}
};

class Derived2 : public Base {
public:
    // Redefinition:
    int f() const {
        cout << "Derived2::f()\n";
        return 2;
    }
};

class Derived3 : public Base {
public:
    // Change return type:
    void f() const { cout << "Derived3::f()\n"; };

class Derived4 : public Base {
public:
    // Change argument list:
    int f(int) const {
        cout << "Derived4::f()\n";
        return 4;
    }
};

int main() {
    string s("hello");
    Derived1 d1;
    int x = d1.f();
    d1.f(s);
    Derived2 d2;
    x = d2.f();
    //! d2.f(s); // string version hidden
    Derived3 d3;
    //! x = d3.f(); // return int version hidden
    Derived4 d4;
    //! x = d4.f(); // f() version hidden
    x = d4.f(1);
}

```

Mostenire multipla

- putine limbaje au MM
- se mosteneste in acelasi timp din mai multe clase

```

class derivata: public Baza1, public Baza2 {
    ...
};

```

- mostenirea multipla e complicata:
ambiguitate
- nu e nevoie de MM (se simuleaza cu
mostenire simpla)
- majoritatea limbajelor OO nu au MM

mostenire multipla: ambiguitate

- clasa baza este mostenita de derivata1 si
derivata2 iar apoi
- clasa derivata3 mosteneste pe derivata1 si 2
- in derivata3 avem de doua ori variabilele
din baza!!!

```

#include <iostream>
using namespace std;

class base {
public:
    int i;
};

// derived1 inherits base.
class derived1 : public base {
public:
    int j;
};

// derived2 inherits base.
class derived2 : public base {
public:
    int k;
};

/* derived3 inherits both derived1 and derived2.
This means that there are two copies of base
in derived3! */
class derived3 : public derived1, public derived2 {
public:
    int sum;
};

int main()
{
    derived3 ob;
    ob.derived1::i = 10; ' this is ambiguous, which i???
    ob.j = 20;
    ob.k = 30;
    // i ambiguous here, too
    ob.sum = ob.derived1::i + ob.j + ob.k;
    // also ambiguous, which i?
    cout << ob.derived::i << " ";
    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;
    return 0;
}

```

- dar daca avem nevoie doar de o copie lui i?
 - nu vrem sa consumam spatiu in memorie
- folosim mostenire virtuala

```

class derivata1: virtual public baza{
...
}

```

Cursul de programare orientata pe obiecte

Seria 13

Saptamana 7 si 8, 1 si 8 aprilie 2014

Andrei Paun

Cuprinsul cursului: 1/8 aprilie 2014

- Mostenirea in C++
- Polimorfism la executie prin functii virtuale

Mostenirea in C++

- in C: copiere de cod si re-utilizare
 - nu a functionat prea bine istoric
- in C++: re-utilizare de cod prin mostenire
 - mostenire de clase create anterior
 - clasele mostenite de multe ori sunt create de alti programatori si verificate deja (codul “merge”)
 - marea putere: modificam codul dar garantam ca portiunile de cod vechi functioneaza

- clasa de baza si clasa derivata
- pentru functii tipul derivat poate substitui tipul de baza
- sintaxa: pentru o clasa definita deja “baza”
class derivata: public baza { definitii noi }

```

#include <iostream>
using namespace std;

class X {
    int i;
public:
    X() { i = 0; }
    void set(int ii) { i = ii; }
    int read() const { return i; }
    int permute() { return i = i * 47; }
};

class Y : public X {
    int i; // Different from X's i
public:
    Y() { i = 0; }
    int change() {
        i = permute(); // Different name call
        return i;
    }
    void set(int ii) {
        i = ii;
        X::set(ii); // Same-name function call
    }
};

int main() {
    cout << "sizeof(X) = " << sizeof(X) << endl;
    cout << "sizeof(Y) = "
        << sizeof(Y) << endl;
    Y D;
    D.change();
    // X function interface comes through:
    D.read();
    D.permute();
    // Redefined functions hide base versions:
    D.set(12);
}

```

- la mostenire: toti membrii “private” ai clasei de baza sunt privati si neaccesibili in clasa derivata, folosesc spatiu, doar ca nu putem sa ii folosim
- daca se face mostenire cu “public” toti membrii publici ai clasei de baza sunt publici in clasa derivata,
- mostenire cu “private” toti membrii publici ai bazei devin privati (si accesibili) in derivata
- aproape tot timpul se face mostenire cu “public”
 - daca nu se precizeaza specificatorul de mostenire, default este PRIVATE

- In Java nici nu se poate “transforma” un membru public in private
- am vazut ca putem redefini functii (set) sau adauga noi functii si variabile
- daca vroiam sa chemam din derivata::set pe baza::set trebuia sa folosim operatorul de rezolutie de scop ::

Initializare de obiecte

- probleme: daca avem obiecte ca variabile de instanta ai unei clase sau la mostenire
 - de obicei ascundem informatiile, deci sunt campuri neaccesibile pentru constructorul clasei curente (derivata)
 - raspuns: se cheama intai constructorul pentru baza si apoi pentru clasa derivata

lista de initializare pentru constructori

- pentru constructorul din clasa derivata care mosteneste pe baza

derivata:: derivata(int i): baza(i) {...}

- spunem astfel ca acest constructor are un param. intreg care e transmis mai departe catre constructorul din baza

- lista de initializare se poate folosi si pentru obiecte incluse in clasa respectiva
- fie un obiect gigi de tip Student in clasa Derivata
- pentru apelarea constructorului pentru gigi cu un param. de initializare i

Derivata::Derivata(int i): Baza(i), gigi(i+3)
{...}

pseudo-constructor pentru tipuri de baza

```
class X {  
    int i;  
    float f;  
    char c;  
    char* s;  
public:  
    X() : i(7), f(1.4), c('x'), s("howdy") {}  
};  
  
int main() {  
    X x;  
    int i(100); // Applied to ordinary definition  
    int* ip = new int(47);  
}
```

ordinea chemarii constructorilor si destructorilor

- constructorii sunt chemati in ordinea definirii obiectelor ca membri ai clasei si in ordinea mostenirii:
 - radacina arborelui de mostenire este primul constructor chemat, constructorii din obiectele membru in clasa respectiva sunt chemati in ordinea definirii
 - apoi se merge pe urmatorul nivel in ordinea mostenirii
- destructorii sunt chemati in ordinea inversa a constructorilor

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

class cls
{
    int x;
public:
    cls(int i=0) {cout << "Inside constructor 1" << endl; x=i; }
    ~cls(){ cout << "Inside destructor 1" << endl;} };

class cls2
{
    int x;
    cls xx;
public:
    cls2(int i=0) {cout << "Inside constructor 2" << endl; x=i; }
    ~cls2(){ cout << "Inside destructor 2" << endl;} };

class cls3
{
    int x;
    cls2 xx;
    cls3 xxx;
public:
    cls3(int i=0) {cout << "Inside constructor 3" << endl; x=i; }
    ~cls3(){ cout << "Inside destructor 3" << endl;} };

main()
{
    cls3 s;
}
```

Inside constructor 1
Inside constructor 2
Inside constructor 1
Inside constructor 3
Inside destructor 3
Inside destructor 1
Inside destructor 2
Inside destructor 1

particularitati la functii

- constructorii si destructorii nu sunt mosteniti (se redefiniesc noi constr. si destr. pentru clasa derivata)
- similar operatorul = (un fel de constructor)

mostenire cu specificatorul private

- toate componentele private din clasa de baza sunt ascunse in clasa derivata
- componentele public si protected sunt acum private si accesibile in derivata
- nu mai putem trata un obiect de tip derivat ca un obiect din clasa de baza!!!
- acelasi efect cu definirea unui obiect de tip baza in interiorul clasei noi (fara mostenire)

- uneori vrem doar portiuni din interfata bazei sa fie publice
- putem mosteni cu private si apoi in zona de definitie pentru clasa derivata spunem ce componente vrem sa le tinem publice:

```
derivata{  
    ...  
    public:  
        baza::functie1();  
        baza::functie2();  
}
```

```
class Pet {  
public:  
    char eat() const { return 'a'; }  
    int speak() const { return 2; }  
    float sleep() const { return 3.0; }  
    float sleep(int) const { return 4.0; }  
};  
  
class Goldfish : Pet { // Private inheritance  
public:  
    Pet::eat; // Name publicizes member  
    Pet::sleep; // Both overloaded members exposed  
};  
  
int main() {  
    Goldfish bob;  
    bob.eat();  
    bob.sleep();  
    bob.sleep(1);  
    //! bob.speak();// Error: private member function  
}
```

specificatorul protected

- este între private și public
- secțiuni definite ca protected sunt similare ca definire cu private (sunt ascunse de restul programului)*
- cu singura observație * sunt accesibile la moștenire

- în proiecte mari avem nevoie de acces la zone “protejate” din clase derivate
- cel mai bine este ca variabilele de instanță să fie PRIVATE și funcții care le modifică să fie protected
- în acest fel vă mențineți dreptul de a implementa în alt fel proprietățile clasei respective

```

#include <fstream>
using namespace std;

class Base {
    int i;
protected:
    int read() const { return i; }
    void set(int ii) { i = ii; }
public:
    Base(int ii = 0) : i(ii) {}
    int value(int m) const { return m*i; }
};

class Derived : public Base {
    int j;
public:
    Derived(int jj = 0) : j(jj) {}
    void change(int x) { set(x); }
};

int main() {
    Derived d;
    d.change(10);
}

```

- mostenire: derivata1 din baza (public)
- derivata2 din derivata1 (public)
- daca in baza avem zone “protected” ele sunt transmise si in derivata1,2 tot ca protected
- mostenire derivata1 din baza (private)
atunci zonele protected devin private in derivata1 si neaccesibile in derivata2

Mostenire de tip protected

- `class derivata1: protected baza {...};`
- atunci toti membrii publici si protected din baza devin protected in derivata.
- nu se prea foloseste, inclusa in limbaj pentru completitudine

Mostenire

- Daca in clasa derivata se redefineste o functie suprascrisa (overloaded) in baza, toate variantele functiei respective sunt ascunse (nu mai sunt accesibile in derivata)
- Putem face overload in clasa derivata pe o functie overloduita din baza (dar semnatura noii functii trebuie sa fie distincta de cele din baza)

```

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

class Base {
public:
    int f() const {
        cout << "Base::f()\n";
        return 1;
    }
    int f(string) const { return 1; }
    void g() {}
};

class Derived1 : public Base {
public:
    void g() const {}
};

class Derived2 : public Base {
public:
    // Redefinition:
    int f() const {
        cout << "Derived2::f()\n";
        return 2;
    }
};

class Derived3 : public Base {
public:
    // Change return type:
    void f() const { cout << "Derived3::f()\n"; };

class Derived4 : public Base {
public:
    // Change argument list:
    int f(int) const {
        cout << "Derived4::f()\n";
        return 4;
    };

int main() {
    string s("hello");
    Derived1 d1;
    int x = d1.f();
    d1.f(s);
    Derived2 d2;
    x = d2.f();
    //! d2.f(s); // string version hidden
    Derived3 d3;
    //! x = d3.f(); // return int version hidden
    Derived4 d4;
    //! x = d4.f(); // f() version hidden
    x = d4.f(1);
}

```

Mostenire multipla

- putine limbaje au MM
- se mosteneste in acelasi timp din mai multe clase

```

class derivata: public Baza1, public Baza2 {
    ...
};

```

- mostenirea multipla e complicata:
ambiguitate
- nu e nevoie de MM (se simuleaza cu
mostenire simpla)
- majoritatea limbajelor OO nu au MM

mostenire multipla: ambiguitate

- clasa baza este mostenita de derivata1 si
derivata2 iar apoi
- clasa derivata3 mosteneste pe derivata1 si 2
- in derivata3 avem de doua ori variabilele
din baza!!!

```

#include <iostream>
using namespace std;

class base {
public:
    int i;
};

// derived1 inherits base.
class derived1 : public base {
public:
    int j;
};

// derived2 inherits base.
class derived2 : public base {
public:
    int k;
};

/* derived3 inherits both derived1 and derived2.
This means that there are two copies of base
in derived3! */
class derived3 : public derived1, public derived2 {
public:
    int sum;
};

int main()
{
    derived3 ob;
    ob.derived1::i = 10; ' this is ambiguous, which i???
    ob.j = 20;
    ob.k = 30;
    // i ambiguous here, too
    ob.sum = ob.derived1::i + ob.j + ob.k;
    // also ambiguous, which i?
    cout << ob.derived::i << " ";
    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;
    return 0;
}

```

- dar daca avem nevoie doar de o copie lui i?
 - nu vrem sa consumam spatiu in memorie
- folosim mostenire virtuala

```

class derivata1: virtual public baza{

...

}

```

- astfel daca avem mostenire de doua sau mai multe ori dintr-o clasa de baza (fiecare mostenire trebuie sa fie virtuala) atunci compilatorul alocă spațiu pentru o singură copie
- în clasele derivat1 și 2 mostenirea e la fel ca mai înainte (nici un efect pentru virtual în acel caz)

```
// This program uses virtual base classes.
#include <iostream>
using namespace std;

class base { public: int i; };

// derived1 inherits base as virtual.
class derived1 : virtual public base {
public:
    int j;
};

// derived2 inherits base as virtual.
class derived2 : virtual public base {
public:
    int k;
};

/* derived3 inherits both derived1 and derived2.
This time, there is only one copy of base class. */
class derived3 : public derived1, public derived2 {
public:
    int sum;
};

int main()
{
    derived3 ob;
    ob.i = 10; // now unambiguous
    ob.j = 20;
    ob.k = 30;

    // unambiguous
    ob.sum = ob.i + ob.j + ob.k;

    // unambiguous
    cout << ob.i << " ";
    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;

    return 0;
}
```

putem avea si functii virtuale

- folosit pentru polimorfism la executie
- functii virtuale sunt definite in baza si redefinite in clasa derivata
- pointer de tip baza care arata catre obiect de tip derivat si cheama o functie virtuala definita in baza si in derivata executa

FUNCTIA DIN CLASA DERIVATA

Mostenire: discutie

- tipul derivat este un subtip pentru tipul de baza; putem folosi obiecte de tip derivat in loc de obiecte de tip de baza
- mostenirea este foarte folositoare la proiecte mari: un bug este identificat in clasa nou creata
- nu ne trebuie definitia clasei de baza ci doar descrierea clasei (probabil in fisier .h) si codul deja compilat

- problemele din proiecte mari se rezolva incremental
 - clasele si mostenirea sunt utile in acest caz
- structura de clase trebuie sa fie “rezonabila” cu proiectul respectiv
 - clasele trebuie sa nu fie prea mici si nici prea mari
 - clase mici: obliga programatorul la mostenire fara a folosi direct clasa respectiva
 - clase mari: prea multe functii incat nu se poate tine minte ce face

Funcții virtuale

- proprietate fundamentala la C++
- in afara de compilator mai “rau” care verifica mai mult decat un compilator de C
- si obiecte (gruparea datelor cu functiile aferente si ascundere de informatii)
- functiile virtuale dau polimorfism

- Cod mai bine organizat cu polimorfism
- Codul poate “crește” fara schimbări semnificative: programe extensibile
- Poate fi văzută și ca exemplu de separare dintre interfață și implementare

- Pana acum:
 - Encapsulare (date și metode împreună)
 - Controlul accesului (private/public)
 - Acum: decuplare în privința tipurilor
 - Tipul derivat poate lua locul tipului de bază (foarte important pentru procesarea mai multor tipuri prin același cod)
 - Funcții virtuale: ne lasă să chemăm funcțiile pentru tipul derivat

- Functiile virtuale si felul lor de folosire:
componenta **IMPORTANTA** a limbajului
OOP

- upcasting: clasa derivata poate lua locul clasei de baza
- problema cand facem apel la functie prin pointer (tipul pointerului ne da functia apelata)

```

#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

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

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
    // Redefine interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};

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

int main() {
    Wind flute;
    tune(flute); // Upcasting
}

printeaza pe ecran: Instrument::play
nu Wind::play

```

- in C avem early binding la apel de functii
 - se face la compilare
- in C++ putem defini late binding prin functii virtuale (late, dynamic, runtime binding)
 - se face apel de functie bazat pe tipul obiectului, la rulare (nu se poate face la compilare)
- C++ nu e interpretat ca Java, cum compilam si avem late binding? cod care verifica la rulare tipul obiectului si apeleaza functia corecta

- Late binding pentru o functie: se scrie virtual inainte de definirea functiei.
- Pentru clasa de baza: nu se schimba nimic!
- Pentru clasa derivata: late binding insemna ca un obiect derivat folosit in locul obiectului de baza isi va folosi functia sa, nu cea din baza (din cauza de late binding)

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.
```

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

```
// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
    // Override interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};
```

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

```
int main() {
    Wind flute;
    tune(flute); // Upcasting
}
```

printeaza pe ecran: Wind::play
nu Instrument::play

- in acest fel putem defini functii pentru tipul de baza si daca le apelam cu parametri obiecte de tip derivat se apeleaza corect functiile pentru tipurile derivate
- functiile care lucreaza cu tipul de baza nu trebuiesc schimbate pentru considerarea particularitatilor fiecarui nou tip derivat

Si ce e asa de util?

- Intrebare: de ce atata fanfara pentru o chestie simpla?
- Pentru ca putem extinde codul precedent **fara schimbări** in codul deja scris.

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.
```

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

```
// Wind objects are Instruments
// because they have the same interface:
```

```
class Wind : public Instrument {
public:
    // Override interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};
```

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

```
class Percussion : public Instrument {
public:
    void play(note) const {
        cout << "Percussion::play" << endl; } };
class Stringed : public Instrument {
public:
    void play(note) const {
        cout << "Stringed::play" << endl; } };
class Brass : public Wind {
public:
    void play(note) const {
        cout << "Brass::play" << endl; } };
class Woodwind : public Wind {
public:
    void play(note) const {
        cout << "Woodwind::play" << endl; } };
```

```
int main() {
    Wind flute;
    Percussion drum;
    Stringed violin;
    Brass flugelhorn;
    Woodwind recorder;
    tune(flute); tune(flugehorn); tune(violin);
}
```

- Daca o functie virtuala nu e redefinita in clasa derivata: cea mai apropiata redefinire este folosita
- Trebuie sa privim conceperea structurii unui program prin prisma functiilor virtuale
- De multe ori realizam ca structura de clase aleasa initial nu e buna si trebuie redefinita

- Redefinirea structurii de clase si chestiunile legate de acest lucru (refactoring) sunt uzuale si nu sunt considerate greseli
- Pur si simplu cu informatia initiala s-a conceput o structura
- Dupa ce s-a obtinut mai multa informatie despre proiect si problemele legate de implementare se ajunge la alta structura

Cum se face late binding

- Tipul obiectului este tinut in obiect pentru clasele cu functii virtuale
- Demonstratie: urmatorul program

```

#include <iostream>
using namespace std;

class NoVirtual {
    int a;
public:
    void x() const {}
    int i() const { return 1; }
};

class OneVirtual {
    int a;
public:
    virtual void x() const {}
    int i() const { return 1; }
};

class TwoVirtuals {
    int a;
public:
    virtual void x() const {}
    virtual int i() const { return 1; }
};

int main() {
    cout << "int: " << sizeof(int) << endl;
    cout << "NoVirtual: "
        << sizeof(NoVirtual) << endl;
    cout << "void* : " << sizeof(void*) << endl;
    cout << "OneVirtual: "
        << sizeof(OneVirtual) << endl;
    cout << "TwoVirtuals: "
        << sizeof(TwoVirtuals) << endl;
}

```

- Late binding se face (uzual) cu o tabela de pointeri: vptr catre functii
- In tabela sunt adresele functiilor clasei respective (functiile virtuale sunt din clasa, celelalte pot fi mostenite, etc.)
- Fiecare obiect din clasa are pointerul acesta in componenta

- La apel de functie membru se merge la obiect, se apeleaza functia prin vptr
- Apel functie play: este functia membru nr. 7
- Chem functia o.vptr+7
- Vptr este initializat in constructor (automat)

Functii virtuale si obiecte

- Pentru obiecte accesate direct stim tipul, deci nu ne trebuie late binding
- Early binding se foloseste in acest caz
- Polimorfism la executie: prin pointeri!


```

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

class Pet {
public:
    virtual string speak() const { return ""; }
};

class Dog : public Pet {
public:
    string speak() const { return "Bark!"; }
};

int main() {
    Dog ralph;
    Pet* p1 = &ralph;
    Pet& p2 = ralph;
    Pet p3;
    // Late binding for both:
    cout << "p1->speak() = " << p1->speak() << endl;
    cout << "p2.speak() = " << p2.speak() << endl;
    // Early binding (probably):
    cout << "p3.speak() = " << p3.speak() << endl;
}

```

- Daca functiile virtuale sunt asa de importante de ce nu sunt toate functiile definite virtuale (din oficiu)
- deoarece “costa” in viteza programului
- In Java sunt “default”, dar Java e mai lent
- Nu mai putem avea functii inline (ne trebuie adresa functiei pentru VPTR)

Clase abstracte si functii virtuale pure

- Uneori vrem clase care dau doar interfata (nu vrem obiecte din clasa abstracta ci upcasting la ea)
- Compilatorul da eroare cand incercam sa instantiem o clasa abstracta
- Clasa abstracta=clasa cu cel putin o functie virtuala PURA

Functii virtuale pure

- Functie virtuala urmata de =0
- Ex: `virtual int pura(int i)=0;`
- La mostenire se defineste functia pura si clasa derivata poate fi folosita normal, daca nu se defineste functia pura, clasa derivata este si ea clasa abstracta
- In felul acesta nu trebuie definita functie care nu se executa niciodata

- In exemplul cu instrumentele: clasa instrument a fost folosita pentru a crea o interfata comuna pentru toate clasele derivate
- Nu planuim sa creem obiecte de tip instrument, putem genera eroare daca se ajunge sa se apeleze o functie membru din clasa instrument
- Dar trebuie sa asteptam la rulare sa se faca acest apel; mai simplu: verificam la compilare prin functii virtuale pure

Functii virtuale pure

- Putem defini functia play din instrument ca virtuala pura
- Daca se instantiaza instrument in program=eroare de compilare
- pot preveni “object slicing” (mai tarziu)
 - Aceasta este probabil cea mai importanta utilizare a lor

Clase abstracte

- Nu pot fi trimise catre functii (prin valoare)
- Trebuie folositi pointeri sau referintele pentru clase abstracte (ca sa putem avea upcasting)
- Daca vrem o functie sa fie comuna pentru toata ierarhia o putem declara virtuala si o si definim. Apel prin operatorul de rezolutie de scop:
cls_abs::functie_pura();
- Putem trece de la func. normale la pure; compilatorul da eroare la clasele care nu au redefinit functia respectiva

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

class Pet {
    string pname;
public:
    Pet(const string& name) : pname(name) {}
    virtual string name() const { return pname; }
    virtual string description() const {
        return "This is " + pname;
    }
};

class Dog : public Pet {
    string favoriteActivity;
public:
    Dog(const string& name, const string& activity)
        : Pet(name), favoriteActivity(activity) {}
    string description() const {
        return Pet::name() + " likes to " +
            favoriteActivity;
    }
};

void describe(Pet p) { // Slicing
    cout << p.description() << endl;
}

int main() {
    Pet p("Alfred");
    Dog d("Fluffy", "sleep");
    describe(p);
    describe(d);
}
```

Overload pe functii virtuale

- Nu e posibil overload prin schimbarea tipului param. de intoarcere (e posibil pentru ne-virtuale)
- Pentru ca se vrea sa se garanteze ca se poate chema baza prin apelul respectiv, deci daca baza a specificat int ca intoarcere si derivata trebuie sa mentina int la intoarcere
- Exceptie: pointer catre baza intors in baza, pointer catre derivata in derivata

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

```
class Base {
public:
    virtual int f() const {
        cout << "Base::f()\n";
        return 1;
    }
    virtual void f(string) const {}
    virtual void g() const {}
};
```

```
class Derived1 : public Base {
public:
    void g() const {}
};
```

```
class Derived2 : public Base {
public:
    // Overriding a virtual function:
    int f() const {
        cout << "Derived2::f()\n";
        return 2;
    }
};
```

```
class Derived3 : public Base {
public:
    //! void f() const{ cout << "Derived3::f()\n";};
```

```
class Derived4 : public Base {
public:
    // Change argument list:
    int f(int) const {
        cout << "Derived4::f()\n";
        return 4;
    }
};
```

```
int main() {
    string s("hello");
    Derived1 d1;
    int x = d1.f();
    d1.f(s);
    Derived2 d2;
    x = d2.f();
    //! d2.f(s); // string version hidden
    Derived4 d4;
    x = d4.f(1);
    //! x = d4.f(); // f() version hidden
    //! d4.f(s); // string version hidden
    Base& br = d4; // Upcast
    //! br.f(1); // Derived version unavailable
    br.f(); br.f(s); // Base version available}
```

Apeluri de functii virtuale in constructori

- Varianta locala este folosita (early binding), nu e ceea ce se intampla in mod normal cu functiile virtuale
- De ce?
 - Pentru ca functia virtuala din clasa derivata ar putea crede ca obiectul e initializat deja
 - Pentru ca la nivel de compilator in acel moment doar VPTR local este cunoscut
- Nu putem avea constructori virtuali

Destructorii si virtual-izare

- Putem avea destructori virtuali
- Este uzual sa se intalneasca
- Se cheama in ordine inversa decat constructorii
- Daca vrem sa eliminam portiuni alocate dinamic si pentru clasa derivata dar facem upcasting trebuie sa folosim destructori virtuali

```

#include <iostream>
using namespace std;

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; // Upcast
    delete bp;
    Base2* b2p = new Derived2; // Upcast
    delete b2p;
}

```

Destructorii virtuali puri

- Putem avea destructori virtuali puri
- Restrictie: trebuiesc definiti in clasa (chiar daca este abstracta)
- La mostenire nu mai trebuiesc redefiniti (se construiesc un destructor din oficiu)
- De ce? Pentru a preveni instantierea clasei

```
class AbstractBase {  
public:  
    virtual ~AbstractBase() = 0;  
};  
  
AbstractBase::~~AbstractBase() {}  
  
class Derived : public AbstractBase {};  
// No overriding of destructor necessary?  
  
int main() { Derived d; }
```

- Sugestie: oricand se defineste un destructor se defineste virtual daca mai sunt alte functii virtuale
- In felul asta nu exista surprize mai tarziu
- Nu are nici un efect daca nu se face upcasting, etc.

Functii virtuale in destructori

- La apel de functie virtuala din functii normale se apeleaza conform VPTR
- In destructori se face early binding! (apeluri locale)
- De ce? Pentru ca acel apel poate sa se bazeze pe portiuni deja distruse din obiect

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

class Base {
public:
    virtual ~Base() {
        cout << "Base1()\n";
        f();
    }
    virtual void f() { cout << "Base::f()\n"; }
};

class Derived : public Base {
public:
    ~Derived() { cout << "~Derived()\n"; }
    void f() { cout << "Derived::f()\n"; }
};

int main() {
    Base* bp = new Derived; // Upcast
    delete bp;
}
```

Cursul de programare orientata pe obiecte

Seria 13

Saptamana 9, 15 aprilie 2014

Andrei Paun

Cuprinsul cursului: 15 aprilie 2014

- Recapitulare functii virtuale
- Upcasting si downcasting
- Template-uri in C++

Functii virtuale

- proprietate fundamentala la C++
- in afara de compilator mai “rau” care verifica mai mult decat un compilator de C
- si obiecte (gruparea datelor cu functiile aferente si ascundere de informatii)
- functiile virtuale dau polimorfism

- Cod mai bine organizat cu polimorfism
- Codul poate “creste” fara schimbari semnificative: programe extensibile
- Poate fi vazuta si ca exemplu de separare dintre interfata si implementare

- Pana acum:
 - Encapsulare (date si metode impreuna)
 - Controlul accesului (private/public)
 - Acum: decuplare in privinta tipurilor
 - Tipul derivat poate lua locul tipului de baza (foarte important pentru procesarea mai multor tipuri prin acelasi cod)
 - Functii virtuale: ne lasa sa chemam functiile pentru tipul derivat

- upcasting: clasa derivata poate lua locul clasei de baza
- problema cand facem apel la functie prin pointer (tipul pointerului ne da functia apelata)

```

#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

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

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
    // Redefine interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};

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

int main() {
    Wind flute;
    tune(flute); // Upcasting
}

printeaza pe ecran: Instrument::play
nu Wind::play

```

- in C avem early binding la apel de functii
 - se face la compilare
- in C++ putem defini late binding prin functii virtuale (late, dynamic, runtime binding)
 - se face apel de functie bazat pe tipul obiectului, la rulare (nu se poate face la compilare)
- C++ nu e interpretat ca Java, cum compilam si avem late binding? cod care verifica la rulare tipul obiectului si apeleaza functia corecta

- Late binding pentru o functie: se scrie virtual inainte de definirea functiei.
- Pentru clasa de baza: nu se schimba nimic!
- Pentru clasa derivata: late binding insemna ca un obiect derivat folosit in locul obiectului de baza isi va folosi functia sa, nu cea din baza (din cauza de late binding)

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.
```

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

```
// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
    // Override interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};
```

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

```
int main() {
    Wind flute;
    tune(flute); // Upcasting
}
```

printeaza pe ecran: Wind::play
nu Instrument::play

- in acest fel putem defini functii pentru tipul de baza si daca le apelam cu parametri obiecte de tip derivat se apeleaza corect functiile pentru tipurile derivate
- functiile care lucreaza cu tipul de baza nu trebuiesc schimbate pentru considerarea particularitatilor fiecarui nou tip derivat

Si ce e asa de util?

- Intrebare: de ce atata fanfara pentru o chestie simpla?
- Pentru ca putem extinde codul precedent **fara schimbări** in codul deja scris.

```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.
```

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

```
// Wind objects are Instruments
// because they have the same interface:
```

```
class Wind : public Instrument {
public:
    // Override interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};
```

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

```
class Percussion : public Instrument {
public:
    void play(note) const {
        cout << "Percussion::play" << endl; } };
class Stringed : public Instrument {
public:
    void play(note) const {
        cout << "Stringed::play" << endl; } };
class Brass : public Wind {
public:
    void play(note) const {
        cout << "Brass::play" << endl; } };
class Woodwind : public Wind {
public:
    void play(note) const {
        cout << "Woodwind::play" << endl; } };
```

```
int main() {
    Wind flute;
    Percussion drum;
    Stringed violin;
    Brass flugelhorn;
    Woodwind recorder;
    tune(flute); tune(flugehorn); tune(violin);
}
```

- Daca o functie virtuala nu e redefinita in clasa derivata: cea mai apropiata redefinire este folosita
- Trebuie sa privim conceperea structurii unui program prin prisma functiilor virtuale
- De multe ori realizam ca structura de clase aleasa initial nu e buna si trebuie redefinita

- Redefinirea structurii de clase si chestiunile legate de acest lucru (refactoring) sunt uzuale si nu sunt considerate greseli
- Pur si simplu cu informatia initiala s-a conceput o structura
- Dupa ce s-a obtinut mai multas informatie despre proiect si problemele legate de implementare se ajunge la alta structura

Cum se face late binding

- Tipul obiectului este tinut in obiect pentru clasele cu functii virtuale

- Late binding se face (uzual) cu o tabela de pointeri: vptr catre functii
- In tabela sunt adresele functiilor clasei respective (functiile virtuale sunt din clasa, celelalte pot fi mostenite, etc.)
- Fiecare obiect din clasa are pointerul acesta in componenta

Functii virtuale si obiecte

- Pentru obiecte accesate direct stim tipul, deci nu ne trebuie late binding
- Early binding se foloseste in acest caz
- Polimorfism la executie: prin pointeri!

```

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

class Pet {
public:
    virtual string speak() const { return ""; }
};

class Dog : public Pet {
public:
    string speak() const { return "Bark!"; }
};

int main() {
    Dog ralph;
    Pet* p1 = &ralph;
    Pet& p2 = ralph;
    Pet p3;
    // Late binding for both:
    cout << "p1->speak() = " << p1->speak() << endl;
    cout << "p2.speak() = " << p2.speak() << endl;
    // Early binding (probably):
    cout << "p3.speak() = " << p3.speak() << endl;
}

```

- Daca functiile virtuale sunt asa de importante de ce nu sunt toate functiile definite virtuale (din oficiu)
- deoarece “costa” in viteza programului
- In Java sunt “default”, dar Java e mai lent
- Nu mai putem avea functii inline (ne trebuie adresa functiei pentru VPTR)

Clase abstracte si functii virtuale pure

- Uneori vrem clase care dau doar interfata (nu vrem obiecte din clasa abstracta ci upcasting la ea)
- Compilatorul da eroare cand incercam sa instantiem o clasa abstracta
- Clasa abstracta=clasa cu cel putin o functie virtuala PURA

Functii virtuale pure

- Functie virtuala urmata de =0
- Ex: `virtual int pura(int i)=0;`
- La mostenire se defineste functia pura si clasa derivata poate fi folosita normal, daca nu se defineste functia pura, clasa derivata este si ea clasa abstracta
- In felul acesta nu trebuie definita functie care nu se executa niciodata

- In exemplul cu instrumentele: clasa instrument a fost folosita pentru a crea o interfata comuna pentru toate clasele derivate
- Nu planuim sa creem obiecte de tip instrument, putem genera eroare daca se ajunge sa se apeleze o functie membru din clasa instrument
- Dar trebuie sa asteptam la rulare sa se faca acest apel; mai simplu: verificam la compilare prin functii virtuale pure

Functii virtuale pure

- Putem defini functia play din instrument ca virtuala pura
- Daca se instantiaza instrument in program=eroare de compilare
- pot preveni “object slicing” (mai tarziu)
 - Aceasta este probabil cea mai importanta utilizare a lor

Clase abstracte

- Nu pot fi trimise catre functii (prin valoare)
- Trebuie folositi pointeri sau referintele pentru clase abstracte (ca sa putem avea upcasting)
- Daca vrem o functie sa fie comuna pentru toata ierarhia o putem declara virtuala si o si definim. Apel prin operatorul de rezolutie de scop:
cls_abs::functie_pura();
- Putem trece de la func. Normale la pure; compilatorul da eroare la clasele care nu au redefinit functia respectiva

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

class Pet {
    string pname;
public:
    Pet(const string& name) : pname(name) {}
    virtual string name() const { return pname; }
    virtual string description() const {
        return "This is " + pname;
    }
};

class Dog : public Pet {
    string favoriteActivity;
public:
    Dog(const string& name, const string& activity)
        : Pet(name), favoriteActivity(activity) {}
    string description() const {
        return Pet::name() + " likes to " +
            favoriteActivity;
    }
};

void describe(Pet p) { // Slicing
    cout << p.description() << endl;
}

int main() {
    Pet p("Alfred");
    Dog d("Fluffy", "sleep");
    describe(p);
    describe(d);
}
```

Apeluri de functii virtuale in constructori

- Varianta locala este folosita (early binding), nu e ceea ce se intampla in mod normal cu functiile virtuale
- De ce?
 - Pentru ca functia virtuala din clasa derivata ar putea crede ca obiectul e initializat deja
 - Pentru ca la nivel de compilator in acel moment doar VPTR local este cunoscut
- Nu putem avea constructori virtuali

Destructorii si virtual-izare

- Putem avea destructori virtuali
- Este uzual sa se intalneasca
- Se cheama in ordine inversa decat constructorii
- Daca vrem sa eliminam portiuni alocate dinamic si pentru clasa derivata dar facem upcasting trebuie sa folosim destructori virtuali

Destructorii virtuali puri

- Putem avea destructori virtuali puri
- Restricție: trebuie să fie definiți în clasă (chiar dacă este abstractă)
- La moștenire nu mai trebuie să fie redefiniți (se construiește un destructor din oficiu)
- De ce? Pentru a preveni instanțierea clasei

```
class AbstractBase {  
public:  
    virtual ~AbstractBase() = 0;  
};  
  
AbstractBase::~~AbstractBase() {}  
  
class Derived : public AbstractBase {};  
// No overriding of destructor necessary?  
  
int main() { Derived d; }
```


- Sugestie: oricand se defineste un destructor se defineste virtual
- In felul asta nu exista surprize mai tarziu
- Nu are nici un efect daca nu se face upcasting, etc.

Functii virtuale in destructori

- La apel de functie virtuala din functii normale se apeleaza conform VPTR
- In destructori se face early binding! (apeluri locale)
- De ce? Pentru ca acel apel poate sa se bazeze pe portiuni deja distruse din obiect

Downcasting

- Cum se face upcasting putem sa facem si downcasting
- Problema: upcasting e sigur pentru ca respectivele functii trebuie sa fie definite in baza, downcasting e problematic
- Avem explicit cast prin: `dynamic_cast` (cuvant cheie)

downcasting

- Folosit in ierarhii polimorifice (cu functii virtuale)
- `Static_cast` intoarce pointer catre obiectul care satiface cerintele sau 0
- Foloseste tabelele VTABLE pentru determinarea tipului

```
//: C15:DynamicCast.cpp
#include <iostream>
using namespace std;

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

int main() {
    Pet* b = new Cat; // Upcast
    // Try to cast it to Dog*:
    Dog* d1 = dynamic_cast<Dog*>(b);
    // Try to cast it to Cat*:
    Cat* d2 = dynamic_cast<Cat*>(b);
    cout << "d1 = " << (long)d1 << endl;
    cout << "d2 = " << (long)d2 << endl;
} ///:~
```

- Daca stim cu siguranta tipul obiectului putem folosi “static_cast”

```

//: C15:StaticHierarchyNavigation.cpp
// Navigating class hierarchies with static_cast
#include <iostream>
#include <typeinfo>
using namespace std;

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
} ///:~

```

Template-uri

- Mostenirea: refolosire de cod din clase
- Compunerea de obiecte: similar
- Template-uri: refolosire de cod din afara claselor
- Chestiune sofisticata in C++, nu apare in C++ initial

Exemplu clasic: cozi si stive

- Ideea de stiva este aceeaasi pentru orice tip de elemente
- De ce sa implementez stiva de intregi, stiva de caractere, stiva de float-uri?
 - Implementez ideea de stiva prin templates (sabioane) si apoi refolosesc acel cod pentru intregi, caractere, float-uri

- Creem functii generice care pot fi folosite cu orice tip de date
- Vom vedea: tipul de date este specificat ca parametru pentru functie
- Putem avea template-uri (sabioane) si pentru functii si pentru clase

Functii generice

- Multi algoritmi sunt generici (nu conteaza pe ce tip de date opereaza)
- Inlaturam bug-uri si marim viteza implementarii daca reusim sa refolosim aceeasi implementare pentru un algoritm care trebuie folosit cu mai multe tipuri de date
- O singura implementare, mai multe folosiri

exemplu

- Algoritm de sortare (heapsort)
- $O(n \log n)$ in timp si nu necesita spatiu suplimentar (mergesort necesita $O(n)$)
- Se poate aplica la intregi, float, nume de familie
- Implementam cu sabloane si informam compilatorul cu ce tip de date sa il apeleze

- In esenta o functie generica face auto overload (pentru diverse tipuri de date)

```
template <class Ttype> ret-type func-name(parameter list)
{
    // body of function
}
```

- Ttype este un nume pentru tipul de date folosit (inca indecis), compilatorul il va inlocui cu tipul de date folosit

- In mod traditional folosim “class Ttype”
- Se poate folosi si “typename Ttype”

// Function template example.

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

// This is a function template.

```
template <class X> void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

```
template <class X>
void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

int main()

```
{
    int i=10, j=20;
    double x=10.1, y=23.3;
    char a='x', b='z';
    cout << "Original i, j: " << i << ' ' << j << '\n';
    cout << "Original x, y: " << x << ' ' << y << '\n';
    cout << "Original a, b: " << a << ' ' << b << '\n';
    swapargs(i, j); // swap integers
    swapargs(x, y); // swap floats
    swapargs(a, b); // swap chars
    cout << "Swapped i, j: " << i << ' ' << j << '\n';
    cout << "Swapped x, y: " << x << ' ' << y << '\n';
    cout << "Swapped a, b: " << a << ' ' << b << '\n';
    return 0;
}
```

- Compilatorul va crea 3 functii diferite swapargs (pe intregi, double si caractere)
- Swapargs este o functie generica, sau functie sablon
- Cand o chemam cu parametri se “specializeaza” devine “functie generata”
- Compilatorul “instantiaza” sablonul
- O functie generata este o instanta a unui sablon


```
template <class X>
void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

- Alta forma de a defini template-urile

```
template <class X>
int i; // this is an error
void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

- Specificatia de template trebuie sa fie imediat inaintea definitiei functiei

Putem avea functii cu mai mult de un tip generic

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

template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
    cout << x << ' ' << y << '\n';
}

int main()
{
    myfunc(10, "I like C++");
    myfunc(98.6, 19L);
    return 0;
}
```

- Cand creem un sablon ii dam voie compilatorului sa creeze atatea functii cu acelasi nume cate sunt necesare (d.p.d.v. al parametrilor folositi)

Overload pe sabloane

- Sablon: overload implicit
- Putem face overload explicit
- Se numeste “specializare explicita”
- In cazul specializarii explicite versiunea sablonului care s-ar fi format in cazul tipului de parametrii respectivi nu se mai creeaza (se foloseste versiunea explicita)

```
// Overriding a template function.
```

```
#include <iostream>
```

```
using namespace std;
```

```
template <class X> void swapargs(X &a, X &b)
```

```
{
    X temp;
    temp = a;
    a = b;
    b = temp;
    cout << "Inside template swapargs.\n";
}
```

```
//overrides the generic ver. of swapargs() for ints.
```

```
void swapargs(int &a, int &b)
```

```
{
    int temp;
    temp = a;
    a = b;
    b = temp;
    cout << "Inside swapargs int specialization.\n";
}
```

```
int main()
```

```
{
```

```
int i=10, j=20;
```

```
double x=10.1, y=23.3;
```

```
char a='x', b='z';
```

```
cout << "Original i, j: " << i << ' ' << j << '\n';
```

```
cout << "Original x, y: " << x << ' ' << y << '\n';
```

```
cout << "Original a, b: " << a << ' ' << b << '\n';
```

```
swapargs(i, j); // explicitly overloaded swapargs()
```

```
swapargs(x, y); // calls generic swapargs()
```

```
swapargs(a, b); // calls generic swapargs()
```

```
cout << "Swapped i, j: " << i << ' ' << j << '\n';
```

```
cout << "Swapped x, y: " << x << ' ' << y << '\n';
```

```
cout << "Swapped a, b: " << a << ' ' << b << '\n';
```

```
return 0;
```

```
}
```

Sintaxa noua pentru specializare explicita

```
// Use new-style specialization syntax.  
template<> void swapargs<int>(int &a, int &b)  
{  
    int temp;  
    temp = a;  
    a = b;  
    b = temp;  
    cout << "Inside swapargs int specialization.\n";  
}
```

- Daca se folosesc functii diferite pentru tipuri de date diferite e mai bine de folosit overloading decat sabloane

Putem avea overload si pe sabloane

- Diferita de specializare explicita
- Similar cu overload pe functii (doar ca acum sunt functii generice)
- Simplu: la fel ca la functiile normale

```

// Overload a function template declaration.
#include <iostream>
using namespace std;

// First version of f() template.
template <class X> void f(X a)
{
    cout << "Inside f(X a)\n";
}

// Second version of f() template.
template <class X, class Y> void f(X a, Y b)
{
    cout << "Inside f(X a, Y b)\n";
}

int main()
{
    f(10); // calls f(X)
    f(10, 20); // calls f(X, Y)
    return 0;
}

```

```

// Using standard parameters in a template function.
#include <iostream>
using namespace std;
const int TABWIDTH = 8;

```

```

// Display data at specified tab position.
template<class X> void tabOut(X data, int tab)
{
    for(; tab; tab--)
        for(int i=0; i<TABWIDTH; i++) cout << ' ';
    cout << data << "\n";
}

```

```

int main()
{
    tabOut("This is a test", 0);
    tabOut(100, 1);
    tabOut('X', 2);
    tabOut(10/3, 3);
    return 0;
}

```

```

This is a test
100      X      3

```

Parametri normali in sabloane

- E posibil
- E util
- E uzual

Functii generale: restrictii

- Nu putem inlocui orice multime de functii overloaduite cu un sablon (sabloanele fac aceleasi actiuni pe toate tipurile de date)

Exemplu pentru functii generice

- Bubble sort

```
// A Generic bubble sort.
#include <iostream>
using namespace std;

template <class X> void bubble(
X *items, // pointer to array to be sorted
int count) // number of items in array
{
    register int a, b;
    X t;
    for(a=1; a<count; a++)
        for(b=count-1; b>=a; b--)
            if(items[b-1] > items[b]) {
                // exchange elements
                t = items[b-1];
                items[b-1] = items[b];
                items[b] = t;
            }
}
```

```
int main()
{
    int iarray[7] = {7, 5, 4, 3, 9, 8, 6};
    double darray[5] = {4.3, 2.5, -0.9, 100.2, 3.0};
    int i;
    cout << "Here is unsorted integer array: ";
    for(i=0; i<7; i++)
        cout << iarray[i] << ' ';
    cout << endl;
    cout << "Here is unsorted double array: ";
    for(i=0; i<5; i++)
        cout << darray[i] << ' ';
    cout << endl;
    bubble(iarray, 7);
    bubble(darray, 5);
    cout << "Here is sorted integer array: ";
    for(i=0; i<7; i++)
        cout << iarray[i] << ' ';
    cout << endl;
    cout << "Here is sorted double array: ";
    for(i=0; i<5; i++)
        cout << darray[i] << ' ';
    cout << endl;
    return 0;
}
```

Clase generice

- Sabloane pentru clase nu pentru functii
- Clasa contine toti algoritmi necesari sa lucreze pe un anumit tip de date
- Din nou algoritmi pot fi generalizati, sabloane
- Specificam tipul de date pe care lucram cand obiectele din clasa respectiva sunt create

exemple

- Cozi, stive, liste inlantuite, arbori de sortare

```
template <class Ttype> class class-name {  
    ...  
}  
  
class-name <type> ob;
```

- Ttype este tipul de date parametrizat
- Ttype este precizat cand clasa e instantiata
- Putem avea mai multe tipuri (separate prin virgula)

- Functiile membru ale unei clase generice sunt si ele generice (in mod automat)
- Nu e necesar sa le specificam cu template

```
// This function demonstrates a generic stack.
#include <iostream>
using namespace std;

const int SIZE = 10;

// Create a generic stack class
template <class StackType> class stack {
    StackType stk[SIZE]; // holds the stack
    int tos; // index of top-of-stack
public:
    stack() { tos = 0; } // initialize stack
    void push(StackType ob); // push object
    StackType pop(); // pop object from stack
};

// Push an object.
template <class StackType> void
stack<StackType>::push(StackType ob)
{
    if(tos==SIZE) {
        cout << "Stack is full.\n";
        return;
    }
    stk[tos] = ob;
    tos++;
}

// Pop an object.
template <class StackType> StackType
stack<StackType>::pop()
{
    if(tos==0) {
        cout << "Stack is empty.\n";
        return 0; // return null on empty stack
    }
    tos--;
    return stk[tos];
}

int main(){
    // Demonstrate character stacks.
    stack<char> s1, s2; // create two character stacks
    int i; s1.push('a'); s2.push('x'); s1.push('b');
    s2.push('y'); s1.push('c'); s2.push('z');
    for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
    for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";

    // demonstrate double stacks
    stack<double> ds1, ds2; // create two double stacks
    ds1.push(1.1); ds2.push(2.2); ds1.push(3.3); ds2.push(4.4);
    ds1.push(5.5); ds2.push(6.6);
    for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop();
    for(i=0; i<3; i++) cout << "Pop ds2: " << ds2.pop();
    return 0;
}
```


Mai multe tipuri de date generice intr-o clasa

- Putem folosi dupa “template” in definitie cate tipuri generice vrem

```
/* This example uses two generic data types in a
class definition.
*/
#include <iostream>
using namespace std;

template <class Type1, class Type2> class myclass
{
    Type1 i;
    Type2 j;
public:
    myclass(Type1 a, Type2 b) { i = a; j = b; }
    void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
    myclass<int, double> ob1(10, 0.23);
    myclass<char, char *> ob2('X', "Templates add power.");
    ob1.show(); // show int, double
    ob2.show(); // show char, char *
    return 0;
}
```

- Sabloanele se folosesc cu operatorii suprascrisi
- Exemplul urmator suprascrie operatorul [] pentru creare de array “sigure”

// A generic safe array example.

#include <iostream>

#include <cstdlib>

using namespace std;

const int SIZE = 10;

template <class AType> class atype {

AType a[SIZE];

public:

atype() {

register int i;

for(i=0; i<SIZE; i++) a[i] = i;

}

AType &operator[](int i);

};

// Provide range checking for atype.

template <class AType> AType

&atype<AType>::operator[](int i)

{

if(i<0 || i> SIZE-1) {

cout << "\nIndex value of ";

cout << i << " is out-of-bounds.\n";

exit(1);

}

return a[i];

}

int main()

{

atype<int> intob; // integer array

atype<double> doubleob; // double array

int i;

cout << "Integer array: ";

for(i=0; i<SIZE; i++) intob[i] = i;

for(i=0; i<SIZE; i++) cout << intob[i] << " ";

cout << "\n";

cout << "Double array: ";

for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;

for(i=0; i<SIZE; i++) cout << doubleob[i] << " ";

cout << "\n";

intob[12] = 100; // generates runtime error

return 0;

}

- Se pot specifica si argumente valori in definirea claselor generalizate
- Dupa “template” dam tipurile parametrizate cat si “parametri normali” (ca la functii)
- Acesti “param. normali” pot fi int, pointeri sau referinte; trebuiesc sa fie cunoscuti la compilare: tratati ca si constante
- `template <class tip1, class tip2, int i>`

```
// Demonstrate non-type template arguments.
#include <iostream>
#include <cstdlib>
using namespace std;

// Here, int size is a non-type argument.
template <class AType, int size> class atype {
    AType a[size]; // length of array is passed in size
public:
    atype() {
        register int i;
        for(i=0; i<size; i++) a[i] = i;
    }
    AType &operator[](int i);
};

// Provide range checking for atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
    if(i<0 || i> size-1) {
        cout << "\nIndex value of ";
        cout << i << " is out-of-bounds.\n";
        exit(1);
    }
    return a[i];
}

int main()
{
    atype<int, 10> intob; // integer array of size 10
    atype<double, 15> doubleob; // 15 double array
    int i;
    cout << "Integer array: ";
    for(i=0; i<10; i++) intob[i] = i;
    for(i=0; i<10; i++) cout << intob[i] << " ";
    cout << '\n';
    cout << "Double array: ";
    for(i=0; i<15; i++) doubleob[i] = (double) i/3;
    for(i=0; i<15; i++) cout << doubleob[i] << " ";
    cout << '\n';
    intob[12] = 100; // generates runtime error
    return 0;
}
```

Argumente default si sabloane

- Putem avea valori default pentru tipurile parametrizate

```
template <class X=int> class myclass { //...
```

- Daca instantiem myclass fara sa precizam un tip de date atunci int este tipul de date folosit pentru sablon
- Este posibil sa avem valori default si pentru argumentele valori (nu tipuri)

```
// Demonstrate default template arguments.
```

```
#include <iostream>
```

```
#include <cstdlib>
```

```
using namespace std;
```

```
// Here, AType defaults to int and size defaults to 10.
```

```
template <class AType=int, int size=10>
```

```
class atype {
```

```
    AType a[size]; // size of array is passed in size
```

```
public:
```

```
    atype() {
```

```
        register int i;
```

```
        for(i=0; i<size; i++) a[i] = i;
```

```
    }
```

```
    AType &operator[](int i);
```

```
};
```

```
// Provide range checking for atype.
```

```
template <class AType, int size>
```

```
AType &atype<AType, size>::operator[](int i)
```

```
{
```

```
    if(i<0 || i> size-1) {
```

```
        cout << "\nIndex value of ";
```

```
        cout << i << " is out-of-bounds.\n";
```

```
        exit(1);
```

```
    }
```

```
    return a[i];
```

```
}
```

```
int main()
```

```
{
```

```
    atype<int, 100> intarray; // integer array, size 100
```

```
    atype<double> doublearray; // double array,  
    default size
```

```
    atype<> defarray; // default to int array of size 10
```

```
    int i;
```

```
    cout << "int array: ";
```

```
    for(i=0; i<100; i++) intarray[i] = i;
```

```
    for(i=0; i<100; i++) cout << intarray[i] << " ";
```

```
    cout << '\n';
```

```
    cout << "double array: ";
```

```
    for(i=0; i<10; i++) doublearray[i] = (double) i/3;
```

```
    for(i=0; i<10; i++) cout << doublearray[i] << " ";
```

```
    cout << '\n';
```

```
    cout << "defarray array: ";
```

```
    for(i=0; i<10; i++) defarray[i] = i;
```

```
    for(i=0; i<10; i++) cout << defarray[i] << " ";
```

```
    cout << '\n';
```

```
    return 0;
```

```
}
```

Specializari explicite pentru clase

- La fel ca la sabloanele pentru functii
- Se foloseste `template<>`

// Demonstrate class specialization.

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

```
template <class T> class myclass {
    T x;
public:
    myclass(T a) {
        cout << "Inside generic myclass\n";
        x = a;
    }
    T getx() { return x; }
};
```

// Explicit specialization for int.

```
template <> class myclass<int> {
    int x;
public:
    myclass(int a) {
        cout << "Inside myclass<int> specialization\n";
        x = a * a;
    }
    int getx() { return x; }
};
```

```
int main()
```

```
{
    myclass<double> d(10.1);
    cout << "double: " << d.getx() << "\n\n";
    myclass<int> i(5);
    cout << "int: " << i.getx() << "\n";
    return 0;
}
```

Inside generic myclass

double: 10.1

Inside myclass<int> specialization

int: 25

Typename, export

- Doua cuvinte cheie adaugate la C++ recent
- Se leaga de sabloane (template-uri)
- Typename: 2 folosiri
 - Se poate folosi in loc de class in definitia sablonului `template <typename X> void swapargs(X &a, X &b)`
 - Informeaza compilatorul ca un nume folosit in declaratia template este un tip nu un obiect
`typename X::Name someObject;`

- Deci `typename X::Name someObject;`
- Spune compilatorului ca `X::Name` sa fie tratat ca si tip
- Export: poate preceda declaratia sablonului
- Astfel se poate folosi sablonul din alte fisiere fara a duplica definitia

Chestiuni finale despre sabloane

- Ne da voie sa realizam refolosire de cod
 - Este una dintre cele mai eluzive idealuri in programare
- Sabloanele incep sa devina un concept standard in programare
- Aceasta tendinta este in crestere

Cursul de programare orientata pe obiecte

Seria 13

Saptamana 10, 6 mai 2014

Andrei Paun

Birocratice

- Examen 20 iunie ora 14:00, **semnături**
- Examen LFA 13 iunie ora 9:00
- Bomberman specificatii
- Laboratoare, etc.
- Cursul de pe 29 aprilie 2014 se recupereaza pe 8 mai 2014, ora 8:00 in amfiteatrul II.

Cuprinsul cursului: 23 aprilie 2013

- Recapitulare
- RTTI

Template-uri

- Mostenirea: refolosire de cod din clase
- Compunerea de obiecte: similar
- Template-uri: refolosire de cod din afara claselor
- Chestiune sofisticata in C++, nu apare in C++ initial

Exemplu clasic: cozi si stive

- Ideea de stiva este aceeaasi pentru orice tip de elemente
- De ce sa implementez stiva de intregi, stiva de caractere, stiva de float-uri?
 - Implementez ideea de stiva prin templates (sabloane) si apoi refolosesc acel cod pentru intregi, caractere, float-uri

- Creem functii generice care pot fi folosite cu orice tip de date
- Vom vedea: tipul de date este specificat ca parametru pentru functie
- Putem avea template-uri (sabloane) si pentru functii si pentru clase

Functii generice

- Multi algoritmi sunt generici (nu conteaza pe ce tip de date opereaza)
- Inlaturam bug-uri si marim viteza implementarii daca reusim sa refolosim aceeasi implementare pentru un algoritm care trebuie folosit cu mai multe tipuri de date
- O singura implementare, mai multe folosiri

exemplu

- Algoritm de sortare (heapsort)
- $O(n \log n)$ in timp si nu necesita spatiu suplimentar (mergesort necesita $O(n)$)
- Se poate aplica la intregi, float, nume de familie
- Implementam cu sabloane si informam compilatorul cu ce tip de date sa il apeleze

- In esenta o functie generica face auto overload (pentru diverse tipuri de date)

```
template <class Ttype> ret-type func-name(parameter list)
{
    // body of function
}
```

- Ttype este un nume pentru tipul de date folosit (inca indecis), compilatorul il va inlocui cu tipul de date folosit

- In mod traditional folosim “class Ttype”
- Se poate folosi si “typename Ttype”

// Function template example.

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

// This is a function template.

```
template <class X> void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

```
template <class X>
void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

int main()

```
{
    int i=10, j=20;
    double x=10.1, y=23.3;
    char a='x', b='z';
    cout << "Original i, j: " << i << ' ' << j << '\n';
    cout << "Original x, y: " << x << ' ' << y << '\n';
    cout << "Original a, b: " << a << ' ' << b << '\n';
    swapargs(i, j); // swap integers
    swapargs(x, y); // swap floats
    swapargs(a, b); // swap chars
    cout << "Swapped i, j: " << i << ' ' << j << '\n';
    cout << "Swapped x, y: " << x << ' ' << y << '\n';
    cout << "Swapped a, b: " << a << ' ' << b << '\n';
    return 0;
}
```

- Compilatorul va creea 3 functii diferite swapargs (pe intregi, double si caractere)
- Swapargs este o functie generica, sau functie sablon
- Cand o chemam cu parametri se “specializeaza” devine “functie generata”
- Compilatorul “instantiaza” sablonul
- O functie generata este o instanta a unui sablon

```
template <class X>
void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

- Alta forma de a defini template-urile

```
template <class X>
int i; // this is an error
void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
}
```

- Specificatia de template trebuie sa fie imediat inaintea definitiei functiei

Putem avea functii cu mai mult de un tip generic

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

template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
    cout << x << ' ' << y << '\n';
}

int main()
{
    myfunc(10, "I like C++");
    myfunc(98.6, 19L);
    return 0;
}
```

- Cand creem un sablon ii dam voie compilatorului sa creeze atatea functii cu acelasi nume cate sunt necesare (d.p.d.v. al parametrilor folositi)

Overload pe sabloane

- Sablon: overload implicit
- Putem face overload explicit
- Se numeste “specializare explicita”
- In cazul specializarii explicite versiunea sablonului care s-ar fi format in cazul tipului de parametrii respectivi nu se mai creeaza (se foloseste versiunea explicita)

```
// Overriding a template function.
#include <iostream>
using namespace std;

template <class X> void swapargs(X &a, X &b)
{
    X temp;
    temp = a;
    a = b;
    b = temp;
    cout << "Inside template swapargs.\n";
}

//overrides the generic ver. of swapargs() for ints.
void swapargs(int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
    cout << "Inside swapargs int specialization.\n";
}
```

```
int main()
{
    int i=10, j=20;
    double x=10.1, y=23.3;
    char a='x', b='z';
    cout << "Original i, j: " << i << ' ' << j << '\n';
    cout << "Original x, y: " << x << ' ' << y << '\n';
    cout << "Original a, b: " << a << ' ' << b << '\n';
    swapargs(i, j); // explicitly overloaded swapargs()
    swapargs(x, y); // calls generic swapargs()
    swapargs(a, b); // calls generic swapargs()
    cout << "Swapped i, j: " << i << ' ' << j << '\n';
    cout << "Swapped x, y: " << x << ' ' << y << '\n';
    cout << "Swapped a, b: " << a << ' ' << b << '\n';
    return 0;
}
```

Sintaxa noua pentru specializare explicita

```
// Use new-style specialization syntax.
template<> void swapargs<int>(int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
    cout << "Inside swapargs int specialization.\n";
}
```

- Daca se folosesc functii diferite pentru tipuri de date diferite e mai bine de folosit overloading decat sabloane

Putem avea overload si pe sabloane

- Diferita de specializare explicita
- Similar cu overload pe functii (doar ca acum sunt functii generice)
- Simplu: la fel ca la functiile normale

```
// Overload a function template declaration.
#include <iostream>
using namespace std;

// First version of f() template.
template <class X> void f(X a)
{
    cout << "Inside f(X a)\n";
}

// Second version of f() template.
template <class X, class Y> void f(X a, Y b)
{
    cout << "Inside f(X a, Y b)\n";
}

int main()
{
    f(10); // calls f(X)
    f(10, 20); // calls f(X, Y)
    return 0;
}
```



```
// Using standard parameters in a template function.
```

```
#include <iostream>
```

```
using namespace std;
```

```
const int TABWIDTH = 8;
```

```
// Display data at specified tab position.
```

```
template<class X> void tabOut(X data, int tab)
```

```
{
```

```
    for(; tab; tab--)
```

```
        for(int i=0; i<TABWIDTH; i++) cout << ' ';
```

```
    cout << data << "\n";
```

```
}
```

```
int main()
```

```
{
```

```
    tabOut("This is a test", 0);
```

```
    tabOut(100, 1);
```

```
    tabOut('X', 2);
```

```
    tabOut(10/3, 3);
```

```
    return 0;
```

```
}
```

This is a test

100

X

3

Parametri normali in sabloane

- E posibil
- E util
- E uzual

Functii generale: restrictii

- Nu putem inlocui orice multime de functii overloaduite cu un sablon (sabloanele fac aceleasi actiuni pe toate tipurile de date)

Exemplu pentru functii generice

- Bubble sort

```
// A Generic bubble sort.
```

```
#include <iostream>
```

```
using namespace std;
```

```
template <class X> void bubble(
```

```
X *items, // pointer to array to be sorted
```

```
int count) // number of items in array
```

```
{
```

```
    register int a, b;
```

```
    X t;
```

```
    for(a=1; a<count; a++)
```

```
        for(b=count-1; b>=a; b--)
```

```
            if(items[b-1] > items[b]) {
```

```
                // exchange elements
```

```
                t = items[b-1];
```

```
                items[b-1] = items[b];
```

```
                items[b] = t;
```

```
            }
```

```
}
```

```
int main()
```

```
{
```

```
    int iarray[7] = {7, 5, 4, 3, 9, 8, 6};
```

```
    double darray[5] = {4.3, 2.5, -0.9, 100.2, 3.0};
```

```
    int i;
```

```
    cout << "Here is unsorted integer array: ";
```

```
    for(i=0; i<7; i++)
```

```
        cout << iarray[i] << ' ';
```

```
    cout << endl;
```

```
    cout << "Here is unsorted double array: ";
```

```
    for(i=0; i<5; i++)
```

```
        cout << darray[i] << ' ';
```

```
    cout << endl;
```

```
    bubble(iarray, 7);
```

```
    bubble(darray, 5);
```

```
    cout << "Here is sorted integer array: ";
```

```
    for(i=0; i<7; i++)
```

```
        cout << iarray[i] << ' ';
```

```
    cout << endl;
```

```
    cout << "Here is sorted double array: ";
```

```
    for(i=0; i<5; i++)
```

```
        cout << darray[i] << ' ';
```

```
    cout << endl;
```

```
    return 0;
```

```
}
```

Clase generice

- Sabloane pentru clase nu pentru functii
- Clasa contine toti algoritmi necesari sa lucreze pe un anumit tip de date
- Din nou algoritmi pot fi generalizati, sabloane
- Specificam tipul de date pe care lucram cand obiectele din clasa respectiva sunt create

exemple

- Cozi, stive, liste inlantuite, arbori de sortare

```
template <class Ttype> class class-name {  
    ...  
}  
  
class-name <type> ob;
```

- Ttype este tipul de date parametrizat
- Ttype este precizat cand clasa e instantiata
- Putem avea mai multe tipuri (separate prin virgula)

- Functiile membru ale unei clase generice sunt si ele generice (in mod automat)
- Nu e necesar sa le specificam cu template

// This function demonstrates a generic stack.

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

```
const int SIZE = 10;
```

// Create a generic stack class

```
template <class StackType> class stack {
    StackType stk[SIZE]; // holds the stack
    int tos; // index of top-of-stack
public:
    stack() { tos = 0; } // initialize stack
    void push(StackType ob); // push object
    StackType pop(); // pop object from stack
};
```

// Push an object.

```
template <class StackType> void
stack<StackType>::push(StackType ob)
{
    if(tos==SIZE) {
        cout << "Stack is full.\n";
        return;
    }
    stk[tos] = ob;
    tos++;
}
```

// Pop an object.

```
template <class StackType> StackType
stack<StackType>::pop()
{ if(tos==0) {
    cout << "Stack is empty.\n";
    return 0; // return null on empty stack
}
    tos--;
    return stk[tos];}
```

```
int main(){
```

// Demonstrate character stacks.

```
stack<char> s1, s2; // create two character stacks
int i; s1.push('a'); s2.push('x'); s1.push('b');
s2.push('y'); s1.push('c'); s2.push('z');
for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";
```

// demonstrate double stacks

```
stack<double> ds1, ds2; // create two double stacks
ds1.push(1.1); ds2.push(2.2); ds1.push(3.3); ds2.push(4.4);
ds1.push(5.5); ds2.push(6.6);
for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop();
for(i=0; i<3; i++) cout << "Pop ds2: " << ds2.pop();
return 0;
}
```

Mai multe tipuri de date generice intr-o clasa

- Putem folosi dupa “template” in definitie cate tipuri generice vrem

```

/* This example uses two generic data types in a
class definition.
*/
#include <iostream>
using namespace std;

template <class Type1, class Type2> class myclass
{
    Type1 i;
    Type2 j;
public:
    myclass(Type1 a, Type2 b) { i = a; j = b; }
    void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
    myclass<int, double> ob1(10, 0.23);
    myclass<char, char *> ob2('X', "Templates add power.");
    ob1.show(); // show int, double
    ob2.show(); // show char, char *
    return 0;
}

```

- Sabloanele se folosesc cu operatorii suprascrisi
- Exemplul urmator suprascrie operatorul [] pentru creare de array “sigure”

```
// A generic safe array example.
```

```
#include <iostream>
```

```
#include <cstdlib>
```

```
using namespace std;
```

```
const int SIZE = 10;
```

```
template <class AType> class atype {
```

```
    AType a[SIZE];
```

```
public:
```

```
    atype() {
```

```
        register int i;
```

```
        for(i=0; i<SIZE; i++) a[i] = i;
```

```
    }
```

```
    AType &operator[](int i);
```

```
};
```

```
// Provide range checking for atype.
```

```
template <class AType> AType
```

```
&atype<AType>::operator[](int i)
```

```
{
```

```
    if(i<0 || i> SIZE-1) {
```

```
        cout << "\nIndex value of ";
```

```
        cout << i << " is out-of-bounds.\n";
```

```
        exit(1);
```

```
    }
```

```
    return a[i];
```

```
}
```

```
int main()
```

```
{
```

```
    atype<int> intob; // integer array
```

```
    atype<double> doubleob; // double array
```

```
    int i;
```

```
    cout << "Integer array: ";
```

```
    for(i=0; i<SIZE; i++) intob[i] = i;
```

```
    for(i=0; i<SIZE; i++) cout << intob[i] << " ";
```

```
    cout << "\n";
```

```
    cout << "Double array: ";
```

```
    for(i=0; i<SIZE; i++) doubleob[i] = (double) i/3;
```

```
    for(i=0; i<SIZE; i++) cout << doubleob[i] << " ";
```

```
    cout << "\n";
```

```
    intob[12] = 100; // generates runtime error
```

```
    return 0;
```

```
}
```

- Se pot specifica si argumente valori in definirea claselor generalizate
- Dupa “template” dam tipurile parametrizate cat si “parametri normali” (ca la functii)
- Acesti “param. normali” pot fi int, pointeri sau referinte; trebuiesc sa fie cunoscuti la compilare: tratati ca si constante
- `template <class tip1, class tip2, int i>`

```

// Demonstrate non-type template arguments.
#include <iostream>
#include <cstdlib>
using namespace std;

// Here, int size is a non-type argument.
template <class AType, int size> class atype {
    AType a[size]; // length of array is passed in size
public:
    atype() {
        register int i;
        for(i=0; i<size; i++) a[i] = i;
    }
    AType &operator[](int i);
};

// Provide range checking for atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
    if(i<0 || i> size-1) {
        cout << "\nIndex value of ";
        cout << i << " is out-of-bounds.\n";
        exit(1);
    }
    return a[i];
}

int main()
{
    atype<int, 10> intob; // integer array of size 10
    atype<double, 15> doubleob; // 15 double array
    int i;
    cout << "Integer array: ";
    for(i=0; i<10; i++) intob[i] = i;
    for(i=0; i<10; i++) cout << intob[i] << " ";
    cout << '\n';
    cout << "Double array: ";
    for(i=0; i<15; i++) doubleob[i] = (double) i/3;
    for(i=0; i<15; i++) cout << doubleob[i] << " ";
    cout << '\n';
    intob[12] = 100; // generates runtime error
    return 0;
}

```

Argumente default si sabloane

- Putem avea valori default pentru tipurile parametrizate

```
template <class X=int> class myclass { //...
```

- Daca instantiem myclass fara sa precizam un tip de date atunci int este tipul de date folosit pentru sablon
- Este posibil sa avem valori default si pentru argumentele valori (nu tipuri)


```

// Demonstrate default template arguments.
#include <iostream>
#include <cstdlib>
using namespace std;
// Here, AType defaults to int and size defaults to 10.
template <class AType=int, int size=10>
class atype {
    AType a[size]; // size of array is passed in size
public:
    atype() {
        register int i;
        for(i=0; i<size; i++) a[i] = i;
    }
    AType &operator[](int i);
};

// Provide range checking for atype.
template <class AType, int size>
AType &atype<AType, size>::operator[](int i)
{
    if(i<0 || i> size-1) {
        cout << "\nIndex value of ";
        cout << i << " is out-of-bounds.\n";
        exit(1);
    }
    return a[i];
}

int main()
{
    atype<int, 100> intarray; // integer array, size 100
    atype<double> doublearray; // double array,
    // default size
    atype<> defarray; // default to int array of size 10
    int i;
    cout << "int array: ";
    for(i=0; i<100; i++) intarray[i] = i;
    for(i=0; i<100; i++) cout << intarray[i] << " ";
    cout << "\n";
    cout << "double array: ";
    for(i=0; i<10; i++) doublearray[i] = (double) i/3;
    for(i=0; i<10; i++) cout << doublearray[i] << " ";
    cout << "\n";
    cout << "defarray array: ";
    for(i=0; i<10; i++) defarray[i] = i;
    for(i=0; i<10; i++) cout << defarray[i] << " ";
    cout << "\n";
    return 0;
}

```

Specializari explicite pentru clase

- La fel ca la sabloanele pentru functii
- Se foloseste template<>

// Demonstrate class specialization.

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

```
template <class T> class myclass {
    T x;
public:
    myclass(T a) {
        cout << "Inside generic myclass\n";
        x = a;
    }
    T getx() { return x; }
};
```

```
int main()
{
    myclass<double> d(10.1);
    cout << "double: " << d.getx() << "\n\n";
    myclass<int> i(5);
    cout << "int: " << i.getx() << "\n";
    return 0;
}
```

// Explicit specialization for int.

```
template <> class myclass<int> {
    int x;
public:
    myclass(int a) {
        cout << "Inside myclass<int> specialization\n";
        x = a * a;
    }
    int getx() { return x; }
};
```

Inside generic myclass

double: 10.1

Inside myclass<int> specialization

int: 25

Typename, export

- Doua cuvinte cheie adaugate la C++ recent
- Se leaga de sabloane (template-uri)

- Typename: 2 folosiri

- Se poate folosi in loc de class in definitia
sablonului `template <typename X> void swapargs(X &a, X &b)`

- Informeaza compilatorul ca un nume folosit in
declaratia template este un tip nu un obiect

```
typename X::Name someObject;
```

- Deci `typename X::Name someObject;`
- Spune compilatorului ca `X::Name` sa fie tratat ca si tip
- Export: poate preceda declaratia sablonului
- Astfel se poate folosi sablonul din alte fisiere fara a duplica definitia

Chestiuni finale despre sabloane

- Ne da voie sa realizam refolosire de cod
 - Este una dintre cele mai eluzive idealuri in programare
- Sabloanele incep sa devina un concept standard in programare
- Aceasta tendinta este in crestere

Conversii implicite in C++

Controlul tipului in timpul rularii programului in C++

- se numeste: “runtime type identification”
- sau RTTI
- este prezenta nu numai in C++ (Java, etc.)
- poate fi privita ca o facilitate a limbajului (facilitate secundara)
- se foloseste in situatii dificile

RTTI

- in mod normal ignoram tipul obiectului si lasam functiile virtuale sa proceseze datele
- uneori dorim sa aflam tipul cel mai derivat al unui obiect catre care nu avem decat pointer de tip baza
 - se pot face operatii mai eficiente cu aceasta informatie
 - tipul de baza nu trebuie extins prea mult

RTTI

- apare suficient de frecvent incat majoritatea librariilor cu clase contin functii virtuale pt a produce informatii RTTI
- cand s-a adaugat la C++ procesarea exceptiilor compilatorul a trebuit sa introduca informatii despre tipul obiectului la rulare, asadar RTTI este o extensie mica

casturi la executie

- daca vrem sa schimbam tipul unui obiect la executie
- se verifica daca un downcast este posibil (si deci valid)
- pointer catre baza este castat catre una din derivate

```
// Checks casts at runtime.
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;
class Security {
protected: enum { BASEID = 0 };
public: virtual ~Security() {}
    virtual bool isA(int id) { return (id == BASEID); }
};
class Stock : public Security {
    typedef Security Super;
protected: enum { OFFSET = 1, TYPEID =
    BASEID + OFFSET };
public: bool isA(int id) {
    return id == TYPEID || Super::isA(id); }
    static Stock* dynacast(Security* s) {
        return (s->isA(TYPEID)) ?
            static_cast<Stock*>(s) : 0; }
};
class Bond : public Security {
protected: enum { OFFSET = 2, ... };
public: static Bond* dynacast(Security* s) {
    return (s->isA(TYPEID)) ?
        static_cast<Bond*>(s) : 0; }
};
```

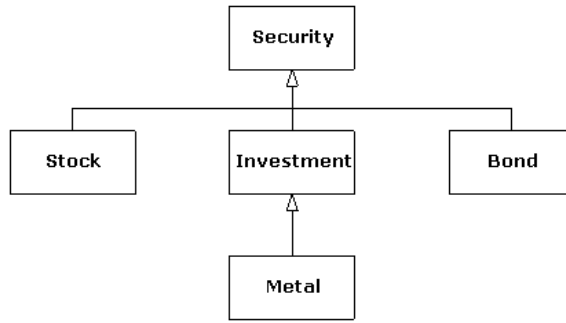
```
class Investment : public Security {
    typedef Security Super;
protected:
    enum { OFFSET = 3, TYPEID = BASEID +
        OFFSET };
public:
    bool isA(int id) {
        return id == TYPEID || Super::isA(id); }
    static Investment* dynacast(Security* s) {
        return (s->isA(TYPEID)) ?
            static_cast<Investment*>(s) : 0; }
    void special() {
        cout << "special Investment function" << endl; }
};

class Metal : public Investment {
    typedef Investment Super;
protected:
    enum { OFFSET = 4, TYPEID = BASEID +
        OFFSET };
public: bool isA(int id) {
    return id == TYPEID || Super::isA(id); }
    static Metal* dynacast(Security* s) {
        return (s->isA(TYPEID)) ?
            static_cast<Metal*>(s) : 0;
    }
};
```

```

int main() {
    vector<Security*> portfolio;
    portfolio.push_back(new Metal);
    portfolio.push_back(new Investment);
    portfolio.push_back(new Bond);
    portfolio.push_back(new Stock);
    for(vector<Security*>::iterator it =
        portfolio.begin();
        it != portfolio.end(); ++it) {
        Investment* cm = Investment::dynacast(*it);
        if(cm)
            cm->special();
        else
            cout << "not an Investment" << endl;
    }
    cout << "cast from intermediate pointer:" <<
        endl;
    Security* sp = new Metal;
    Investment* cp = Investment::dynacast(sp);
    if(cp) cout << " it's an Investment" << endl;
    Metal* mp = Metal::dynacast(sp);
    if(mp) cout << " it's a Metal too!" << endl;
    purge(portfolio);
}

```



downcasting

- in mod normal este descurajat si nu este necesar in mod uzual
- se foloseste in baze de date, browsere de clase si debuggere
- C++ are un cast verificat “dynamic_cast”

```

#include <iostream>
class Security {public: virtual ~Security() {}};
class Stock : public Security {};
class Bond : public Security {};

class Investment : public Security {
public:
    void special() {
        std::cout << "special Investment function"
        <<std::endl; }
};

class Metal : public Investment {};

#include <vector>
#include "../purge.h"
#include "Security.h"
using namespace std;

int main() {
    vector<Security*> portfolio;
    portfolio.push_back(new Metal);
    portfolio.push_back(new Investment);
    portfolio.push_back(new Bond);
    portfolio.push_back(new Stock);
    for(vector<Security*>::iterator it =
        portfolio.begin(); it != portfolio.end(); ++it) {
        Investment* cm =dynamic_cast<Investment*>(*it);
        if(cm)    cm->special();
        else      cout << "not a Investment" << endl;
    }
    cout << "cast from intermediate pointer:" << endl;
    Security* sp = new Metal;
    Investment* cp = dynamic_cast<Investment*>(sp);
    if(cp) cout << " it's an Investment" << endl;
    Metal* mp = dynamic_cast<Metal*>(sp);
    if(mp) cout << " it's a Metal too!" << endl;
    purge(portfolio);
}

```

//: C08:CatchBadCast.cpp

```

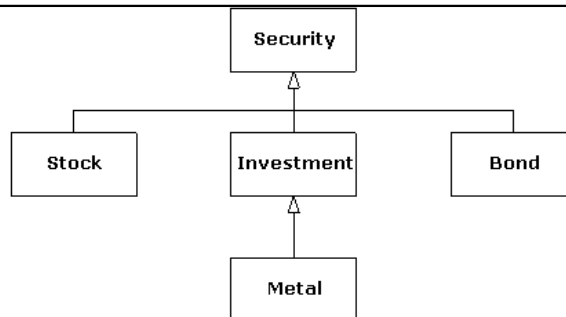
#include <typeinfo>
#include "Security.h"
using namespace std;

```

```

int main() {
    Metal m;
    Security& s = m;
    try {
        Investment& c =
            dynamic_cast<Investment&>(s);
        cout << "It's an Investment" << endl;
    } catch(bad_cast&) {
        cout << "s is not an Investment type" << endl;
    }
    try {
        Bond& b = dynamic_cast<Bond&>(s);
        cout << "It's a Bond" << endl;
    } catch(bad_cast&) {
        cout << "It's not a Bond type" << endl;
    }
}

```



- casting prin referinta
- nu exista referinta null
- se face catch

operatorul typeid

- alt fel de a obtine informatie despre tipul obiectului la rulare
- typeid intoarce un obiect type_info
- daca tipul este polimorfic da informatie despre cea mai derivata clasa (tipul dinamic)
- altfel da informatii statice

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

struct PolyBase { virtual ~PolyBase() {} };
struct PolyDer : PolyBase { PolyDer() {} };
struct NonPolyBase {};
struct NonPolyDer : NonPolyBase { NonPolyDer(int) {} };

int main() { // Test polymorphic Types
    const PolyDer pd;
    const PolyBase* ppb = &pd;
    cout << typeid(ppb).name() << endl;
    cout << typeid(*ppb).name() << endl;
    cout << boolalpha << (typeid(*ppb) == typeid(pd)) << endl;
    cout << (typeid(PolyDer) == typeid(const PolyDer)) << endl;
    // Test non-polymorphic Types
    const NonPolyDer npd(1);
    const NonPolyBase* nppb = &npd;
    cout << typeid(nppb).name() << endl;
    cout << typeid(*nppb).name() << endl;
    cout << (typeid(*nppb) == typeid(npd)) << endl;
    // Test a built-in type
    int i;
    cout << typeid(i).name() << endl;
} ///:~
```

```
struct PolyBase const *
struct PolyDer
true
true
struct NonPolyBase const *
struct NonPolyBase
false
int
```

- prima folosire intoarce tipul pointerului
- pentru RTTI trebuie sa ne uitam la un obiect
 - a doua linie
- RTTI ignora const pentru nivelul superior
- pentru chestii statice se obtine tipul pointerului
- tipurile de baza sunt si ele procesate

- rezultatul unei operatii typeid nu se poate tine intr-un obiect type_info (nu exista constructori accesibili, atribuirea nu e valida)
- se foloseste in comparatii directe
- sirul exact intors depinde de compilator
- typeid(null) genereaza exceptie: bad_typeid

```
//: C08:RTTIandNesting.cpp
```

```
#include <iostream>
```

```
#include <typeinfo>
```

```
using namespace std;
```

One::Nested

```
class One {
```

```
    class Nested {};
```

```
    Nested* n;
```

```
public:
```

```
    One() : n(new Nested) {}
```

```
    ~One() { delete n; }
```

```
    Nested* nested() { return n; }
```

```
};
```

```
int main() {
```

```
    One o;
```

```
    cout << typeid(*o.nested()).name() << endl;
```

```
}
```

- putem verifica daca un type_info precede un alt type_info

```
if(typeid(me).before(typeid(you)))
```

- verifica daca me apare inainte de you
- se foloseste in implementarea type_info ca si chei pentru o structura

castare la niveluri intermediare

- mostenire multipla
- diferente dintre typeid si dynamic_cast
- typeid-> referinta catre type_info
 - nu da informatia de clase intermediare
 - typeid returneaza tipul pointerului in finalul programului urmator

```
#include <cassert>
#include <typeinfo>
using namespace std;

class B1 {
public:
    virtual ~B1() {}
};

class B2 {
public:
    virtual ~B2() {}
};

class MI : public B1, public B2 {};
class Mi2 : public MI {};

int main() {
    B2* b2 = new Mi2;
    Mi2* mi2 = dynamic_cast<Mi2*>(b2);
    MI* mi = dynamic_cast<MI*>(b2);
    B1* b1 = dynamic_cast<B1*>(b2);
    assert(typeid(b2) != typeid(Mi2*));
    assert(typeid(b2) == typeid(B2*));
    delete b2;
}
```

```

//: C08:VoidRTTI.cpp
// RTTI & void pointers.
//!#include <iostream>
#include <typeinfo>
using namespace std;

class Stimpy {
public:
    virtual void happy() {}
    virtual void joy() {}
    virtual ~Stimpy() {}
};

int main() {
    void* v = new Stimpy;
    // Error:
    //! Stimpy* s = dynamic_cast<Stimpy*>(v);
    // Error:
    //! cout << typeid(*v).name() << endl;
} ///:~

```

- RTTI nu functioneaza cu pointeri void
- nu au informatie de tip

RTTI si templateuri

- templatele pe clase genereaza clase, deci se pot folosi cu RTTI

```
//: C08:ConstructorOrder.cpp
```

```
// Order of constructor calls.
```

```
#include <iostream>
```

```
#include <typeinfo>
```

```
using namespace std;
```

```
template<int id> class Announce {
```

```
public:
```

```
    Announce() {
```

```
        cout << typeid(*this).name() << " constructor"
              << endl;
```

```
    }
```

```
    ~Announce() {
```

```
        cout << typeid(*this).name() << " destructor"
              << endl;
```

```
    }
```

```
};
```

```
class X : public Announce<0> {
```

```
    Announce<1> m1;
```

```
    Announce<2> m2;
```

```
public:
```

```
    X() { cout << "X::X()" << endl; }
```

```
    ~X() { cout << "X::~X()" << endl; }
```

```
};
```

```
int main() { X x; } ///:~
```

Announce<0> constructor

Announce<1> constructor

Announce<2> constructor

X::X()

X::~X()

Announce<2> destructor

Announce<1> destructor

Announce<0> destructor

mostenire multipla

```

//: C08:RTTIandMultipleInheritance.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class BB {
public:
    virtual void f() {}
    virtual ~BB() {}
};

class B1 : virtual public BB {};
class B2 : virtual public BB {};
class MI : public B1, public B2 {};

int main() {
    BB* bbp = new MI; // Upcast
    // Proper name detection:
    cout << typeid(*bbp).name() << endl;
    // Dynamic_cast works properly:
    MI* mip = dynamic_cast<MI*>(bbp);
    // Can't force old-style cast:
    //! MI* mip2 = (MI*)bbp; // Compile error
} ///:~

```

folosirea RTTI

- programatorii din lb. imperative sunt tentati sa foloseasca RTTI si cu switch pentru procesarea ierarhiei
- a nu se folosi in felul acesta
- sa se foloseasca functiile virtuale in primul rand, se organizeaza codul cu aceasta idee in minte
- se foloseste RTTI doar cand e necesar

- daca clasa de baza nu este sub controlul programatorului (altcineva o creeaza, e intr-o librerie, etc.)
 - si e nevoie de o functie virtuala definita
 - atunci o solutie este RTTI
 - se deriveaza intr-o alta clasa
 - se defineste o functie noua
 - se detecteaza in cod daca obiectul e de tipul respectiv sau nu, daca da se executa functia noua

- adaugarea unei functii in clasa de baza (folosita doar in una din derivate) irita ceilalti programatori, celelalte clase au o functie nefolosita (si fara sens)
- interfata complicata
- eficienta: codul scris este corect dar pentru un anumit subtip este foarte lent, se ia subtipul respectiv (RTTI) si se implementeaza cod eficient pentru acel caz

mecanismul pentru RTTI

- un pointer extra in tabela de functii virtuale a clasei
- acest pointer catre type_info struct
- cand se cheama typeid():
 - se obtine pointerul catre type_info
 - se returneaza structura type_info
- cost: mergem prin doi pointeri (f rapid)

- **dynamic_cast<dest*>(pointer_sursa)**
- se obtine RTTI pentru pointer_sursa si *dest
- se verifica daca tipul sursei este la fel ca dest sau baza pentru dest
- pointerul poate fi diferit (mostenire multipla) daca sursa nu e prima baza in mostenire
- daca mostenire multipla, clase virtuale si dest apare de mai multe ori in mostenire
- se pot compara clase din ierarhii diferite

Exemplu RTTI

```
#include <iostream.h>
#include <typeinfo.h>
class Base {
public:
    Base():hide(0){ }
    void mutate(int i){ hide = i;}
    virtual void print()const
    { cout <<
        "hide in Base = " << hide << endl;}
private:
    int hide;
};
```

```
class One: public Base {
public:
    One():Base(),data(0) { }
    void mutate(int i){ data = i;}
    void print()const { cout <<
        "data in One = " << data << endl;}
    void newfunc() { cout << "Hi\n";}
private:
    int data;
};
```

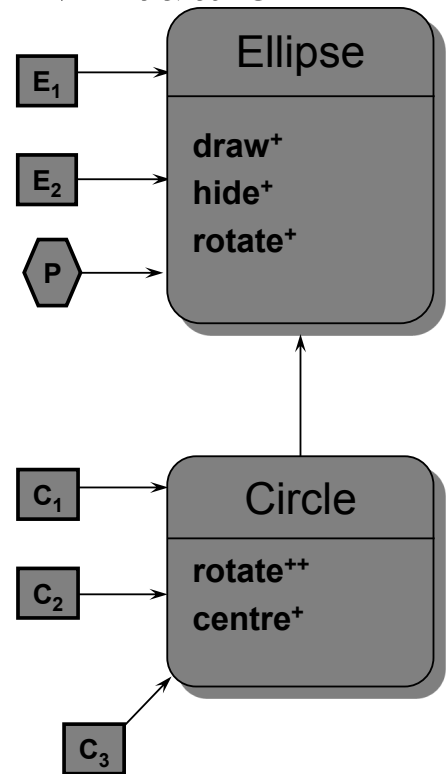
Exemplu RTTI

```
int main ()
{
    Base* bptr; One* derived; One o1; bptr = &o1;
    derived = dynamic_cast<One*>(bptr);
    derived->print();derived->newfunc();
    cout << typeid(bptr).name() << endl << typeid(*bptr).name() << endl;
    if (typeid(*bptr) == typeid(One))
        cout << "type bprt & derived same" << endl;
    cout << typeid(bptr).name();
    cout << " before " << typeid(o1).name() << ": " <<
        (typeid(bptr).before(typeid(o1))) ? true : false << endl;
}
```

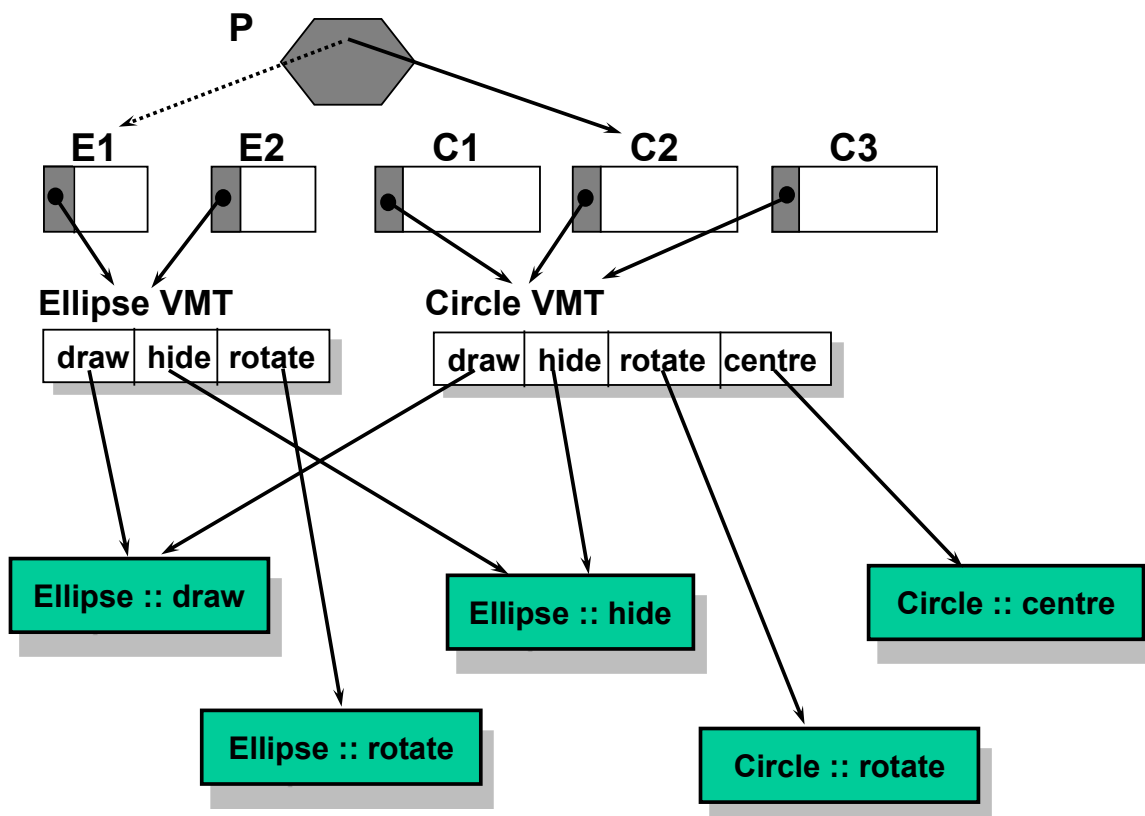
Implementarea Functiilor Virtuale

```
class Ellipse {  
    // ...  
public:  
    virtual void draw() const;  
    virtual void hide() const;  
    virtual void rotate(int);  
} E1, E2, *P;
```

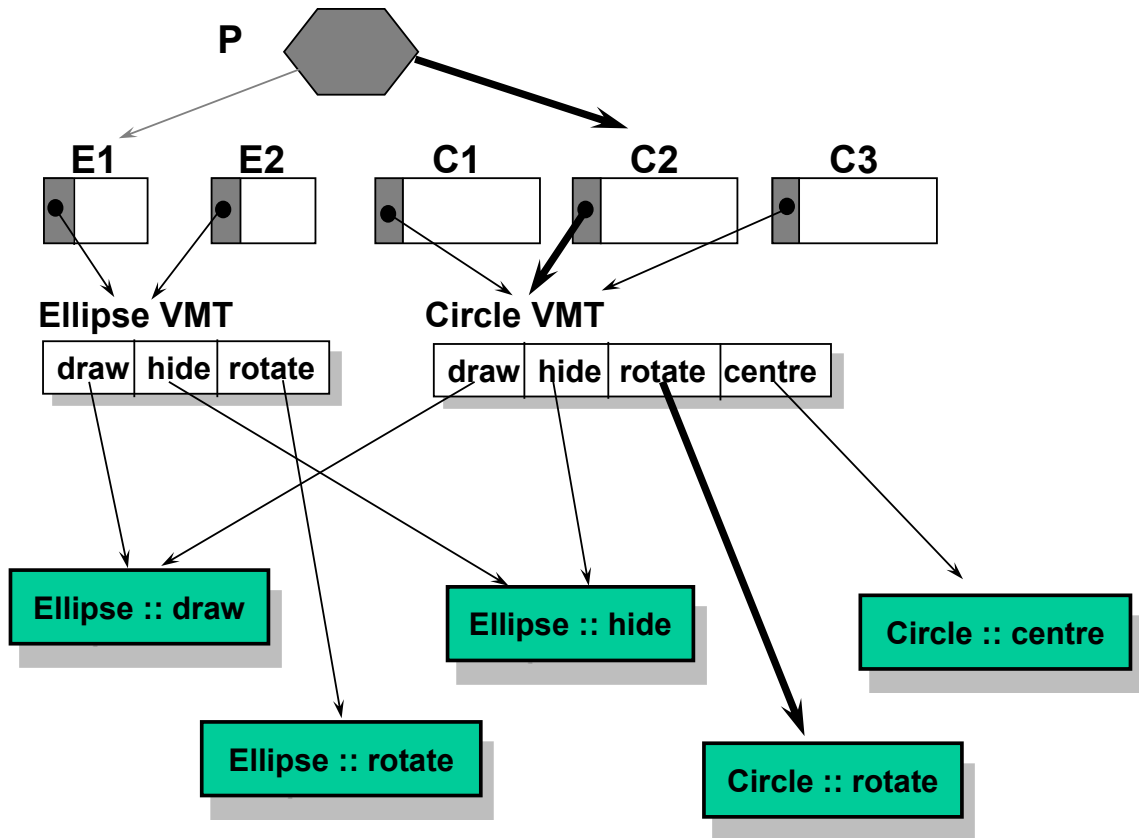
```
class Circle: public Ellipse {  
    //...  
public:  
    virtual void rotate(int);  
    virtual Point centre();  
} C1, C2, C3;
```



Tabelul de metode Virtuale (vtbl)



P->rotate()



Cursul de programare orientata pe obiecte

Seria 13

Saptamana 11, 8 mai 2014
(recuperare)

Andrei Paun

Birocratică

- Examen 20 iunie ora 14:00, **semnături**
- Examen LFA 13 iunie ora 9:00
- Bomberman specificatii
- Laboratoare, etc.
- Cursul de pe 29 aprilie 2014 se recupereaza pe 8 mai 2014, ora 8:00 in amfiteatrul II.

Bomberman

Din regulile mentionate la cursul 1, 18 feb 2014

- Laborator: implementare a unui proiect bonus de dimensiuni medii, pe fiecare subgrupa se va face un clasament si primul si al doilea proiect de acest fel primesc 1 punct bonus la examen
- Cel mai bun proiect pentru serie primeste 2 puncte bonus la examen

- Extra: primele 6 implementari (locurile 1, 2, si 3 de la ambele serii) vor intra in competitie cu implementarea “profilor”, daca bateti strategia noastra luati 10 la examen

Reguli

- Implementarea se va verifica in cursul saptamanii 26-30 Mai 2014 in timpul laboratorului de catre tutorele de laborator. In data de 25 mai 2014 inainte de miezul noptii se va trimite codul sursa atasat la e-mail catre tutorele de laborator si cu CC catre apaun@fmi.unibuc.ro cu urmatoarele informatii: Nume, prenume, subgrupa, ora si ziua laboratorului. O arhiva cu fisierele necesare compilarii se va trimite si pe moodle. Studentii care nu trimit acest e-mail pana la ora 23:59 in ziua de 25 mai 2014 NU vor intra in competitie.
- Pentru subgrupele cu 1 sau 2 studenti aflati in competitie: strategia trebuie sa miste omuletul, sa evite peretii (adica sa nu mearga tot timpul, spre exemplu, la dreapta), sa lase bombe etc. Daca se transmite o implementare prea simpla (se alege random directia de miscare a omuletului, lasa bombe random, merge in cerc continuu, etc. ne rezervam dreptul de a nu acorda niciun punct
- Pentru toate grupele: daca programul are greseli (nu compileaza, nu misca omuletul pe ecran, nu lasa bombe, etc.) studentul nu va primi niciun punct

Alte reguli

- Nu folositi variabile globale
- Nu folositi goto
- Nu folositi variabile de instanta publice
- Numarul variabilelor locale sa fie rezonabil (puteti avea maxim 200 de intregi)
- Intrebari? La pauza

Cuprinsul cursului: 8 mai 2014

- Recapitulare: RTTI
- const si volatile

Controlul tipului in timpul rularii programului in C++

- se numeste: “runtime type identification”
- sau RTTI
- este prezenta nu numai in C++ (Java, etc.)
- poate fi privita ca o facilitate a limbajului (facilitate secundara)
- se foloseste in situatii dificile

RTTI

- in mod normal ignoram tipul obiectului si lasam functiile virtuale sa proceseze datele
- uneori dorim sa aflam tipul cel mai derivat al unui obiect catre care nu avem decat pointer de tip baza
 - se pot face operatii mai eficient cu aceasta informatie
 - tipul de baza nu trebuie extins prea mult

RTTI

- apare suficient de frecvent incat majoritatea librariilor cu clase contin functii virtuale pt a produce informatii RTTI
- cand s-a adaugat la C++ procesarea exceptiilor compilatorul a trebuit sa introduca informatii despre tipul obiectului la rulare, asadar RTTI este o extensie mica

casturi la executie

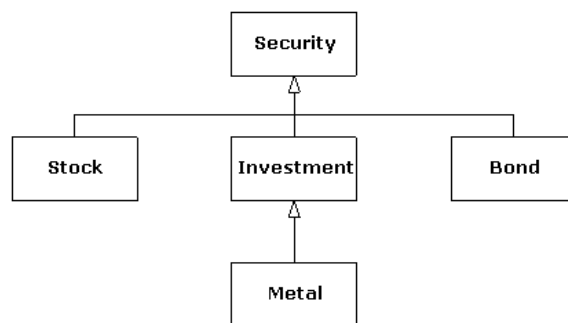
- daca vrem sa schimbam tipul unui obiect la executie
- se verifica daca un downcast este posibil (si deci valid)
- pointer catre baza este castat catre una din derivate

```
// Checks casts at runtime.
#include <iostream>
#include <vector>
#include "../purge.h"
using namespace std;
class Security {
protected: enum { BASEID = 0 };
public: virtual ~Security() {}
    virtual bool isA(int id) { return (id == BASEID); }
};
class Stock : public Security {
    typedef Security Super;
protected: enum { OFFSET = 1, TYPEID =
    BASEID + OFFSET };
public: bool isA(int id) {
    return id == TYPEID || Super::isA(id); }
    static Stock* dynacast(Security* s) {
        return (s->isA(TYPEID)) ?
            static_cast<Stock*>(s) : 0; }
};
class Bond : public Security {
protected: enum { OFFSET = 2, ... };
public: static Bond* dynacast(Security* s) {
    return (s->isA(TYPEID)) ?
        static_cast<Bond*>(s) : 0; }
};
```

```
class Investment : public Security {
    typedef Security Super;
protected:
    enum { OFFSET = 3, TYPEID = BASEID +
        OFFSET };
public:
    bool isA(int id) {
        return id == TYPEID || Super::isA(id); }
    static Investment* dynacast(Security* s) {
        return (s->isA(TYPEID)) ?
            static_cast<Investment*>(s) : 0; }
    void special() {
        cout << "special Investment function" << endl; }
};

class Metal : public Investment {
    typedef Investment Super;
protected:
    enum { OFFSET = 4, TYPEID = BASEID +
        OFFSET };
public: bool isA(int id) {
    return id == TYPEID || Super::isA(id); }
    static Metal* dynacast(Security* s) {
        return (s->isA(TYPEID)) ?
            static_cast<Metal*>(s) : 0;
    }
};
```

```
int main() {
    vector<Security*> portfolio;
    portfolio.push_back(new Metal);
    portfolio.push_back(new Investment);
    portfolio.push_back(new Bond);
    portfolio.push_back(new Stock);
    for(vector<Security*>::iterator it =
        portfolio.begin();
        it != portfolio.end(); ++it) {
        Investment* cm = Investment::dynacast(*it);
        if(cm)
            cm->special();
        else
            cout << "not an Investment" << endl;
    }
    cout << "cast from intermediate pointer:" <<
        endl;
    Security* sp = new Metal;
    Investment* cp = Investment::dynacast(sp);
    if(cp) cout << " it's an Investment" << endl;
    Metal* mp = Metal::dynacast(sp);
    if(mp) cout << " it's a Metal too!" << endl;
    purge(portfolio);
}
```



downcasting

- in mod normal este descurajat si nu este necesar in mod uzual
- se foloseste in baze de date, browsere de clase si debuggere
- C++ are un cast verificat “dynamic_cast”

```
#include <iostream>
class Security {public: virtual ~Security() {}};
class Stock : public Security {};
class Bond : public Security {};

class Investment : public Security {
public:
    void special() {
        std::cout << "special Investment function"
        <<std::endl; }
};

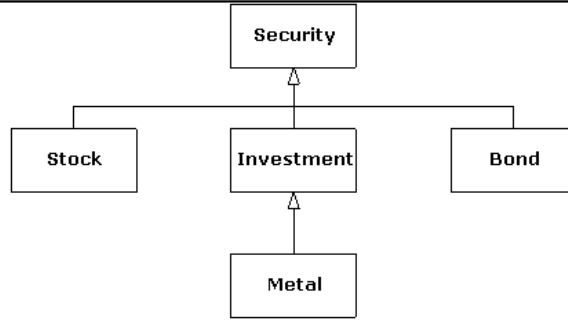
class Metal : public Investment {};

#include <vector>
#include "../purge.h"
#include "Security.h"
using namespace std;

int main() {
    vector<Security*> portfolio;
    portfolio.push_back(new Metal);
    portfolio.push_back(new Investment);
    portfolio.push_back(new Bond);
    portfolio.push_back(new Stock);
    for(vector<Security*>::iterator it =
        portfolio.begin(); it != portfolio.end(); ++it) {
        Investment* cm =dynamic_cast<Investment*>(*it);
        if(cm)    cm->special();
        else      cout << "not a Investment" << endl;
    }
    cout << "cast from intermediate pointer:" << endl;
    Security* sp = new Metal;
    Investment* cp = dynamic_cast<Investment*>(sp);
    if(cp) cout << " it's an Investment" << endl;
    Metal* mp = dynamic_cast<Metal*>(sp);
    if(mp) cout << " it's a Metal too!" << endl;
    purge(portfolio);
}
```

```
//: C08: CatchBadCast.cpp
#include <typeinfo>
#include "Security.h"
using namespace std;
```

```
int main() {
    Metal m;
    Security& s = m;
    try {
        Investment& c =
            dynamic_cast<Investment&>(s);
        cout << "It's an Investment" << endl;
    } catch(bad_cast&) {
        cout << "s is not an Investment type" << endl;
    }
    try {
        Bond& b = dynamic_cast<Bond&>(s);
        cout << "It's a Bond" << endl;
    } catch(bad_cast&) {
        cout << "It's not a Bond type" << endl;
    }
}
```



- casting prin referinte
- nu exista referinta null
- se face catch

operatorul typeid

- alt fel de a obtine informatie despre tipul obiectului la rulare
- typeid intoarce un obiect type_info
- daca tipul este polimorfic da informatie despre cea mai derivata clasa (tipul dinamic)
- altfel da informatii statice

```

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

struct PolyBase { virtual ~PolyBase() {} };
struct PolyDer : PolyBase { PolyDer() {} };
struct NonPolyBase {};
struct NonPolyDer : NonPolyBase { NonPolyDer(int) {} };

int main() { // Test polymorphic Types
    const PolyDer pd;
    const PolyBase* ppb = &pd;
    cout << typeid(ppb).name() << endl;
    cout << typeid(*ppb).name() << endl;
    cout << boolalpha << (typeid(*ppb) == typeid(pd)) << endl;
    cout << (typeid(PolyDer) == typeid(const PolyDer)) << endl;
    // Test non-polymorphic Types
    const NonPolyDer npd(1);
    const NonPolyBase* nppb = &npd;
    cout << typeid(nppb).name() << endl;
    cout << typeid(*nppb).name() << endl;
    cout << (typeid(*nppb) == typeid(npd)) << endl;
    // Test a built-in type
    int i;
    cout << typeid(i).name() << endl;
} ///:~

```

```

struct PolyBase const *
struct PolyDer
true
true
struct NonPolyBase const *
struct NonPolyBase
false
int

```

- prima folosire intoarce tipul pointerului
- pentru RTTI trebuie sa ne uitam la un obiect
– a doua linie
- RTTI ignora const pentru nivelul superior
- pentru chestii statice se obtine tipul pointerului
- tipurile de baza sunt si ele procesate

- rezultatul unei operatii typeid nu se poate tine intr-un obiect type_info (nu exista constructori accesibili, atribuirea nu e valida)
- se foloseste in comparatii directe
- sirul exact intors depinde de compilator
- typeid(null) genereaza exceptie: bad_typeid

```
//: C08:RTTIandNesting.cpp
```

```
#include <iostream>
```

```
#include <typeinfo>
```

```
using namespace std;
```

One::Nested

```
class One {
```

```
    class Nested {};
```

```
    Nested* n;
```

```
public:
```

```
    One() : n(new Nested) {}
```

```
    ~One() { delete n; }
```

```
    Nested* nested() { return n; }
```

```
};
```

```
int main() {
```

```
    One o;
```

```
    cout << typeid(*o.nested()).name() << endl;
```

```
}
```

- putem verifica daca un `type_info` precede un alt `type_info`

`if(typeid(me).before(typeid(you)))`

- verifica daca `me` apare inainte de `you`
- se foloseste in implementarea `type_info` ca si chei pentru o structura

castare la niveluri intermediare

- mostenire multipla
- diferente dintre `typeid` si `dynamic_cast`
- `typeid`-> referinta catre `type_info`
 - nu da informatia de clase intermediare
 - `typeid` returneaza tipul pointerului in finalul programului urmator


```

#include <cassert>
#include <typeinfo>
using namespace std;

class B1 {
public:
    virtual ~B1() {}
};

class B2 {
public:
    virtual ~B2() {}
};

class MI : public B1, public B2 {};
class Mi2 : public MI {};

int main() {
    B2* b2 = new Mi2;
    Mi2* mi2 = dynamic_cast<Mi2*>(b2);
    MI* mi = dynamic_cast<MI*>(b2);
    B1* b1 = dynamic_cast<B1*>(b2);
    assert(typeid(b2) != typeid(Mi2*));
    assert(typeid(b2) == typeid(B2*));
    delete b2;
}

```

```

//: C08:VoidRTTI.cpp
// RTTI & void pointers.
//!#include <iostream>
#include <typeinfo>
using namespace std;

class Stimpy {
public:
    virtual void happy() {}
    virtual void joy() {}
    virtual ~Stimpy() {}
};

int main() {
    void* v = new Stimpy;
    // Error:
    //! Stimpy* s = dynamic_cast<Stimpy*>(v);
    // Error:
    //! cout << typeid(*v).name() << endl;
} ///:~

```

- RTTI nu functioneaza cu pointeri void
- nu au informatie de tip

RTTI si templateuri

- templatele pe clase genereaza clase, deci se pot folosi cu RTTI

```
//: C08:ConstructorOrder.cpp
```

```
// Order of constructor calls.
```

```
#include <iostream>
```

```
#include <typeinfo>
```

```
using namespace std;
```

```
template<int id> class Announce {
```

```
public:
```

```
    Announce() {
```

```
        cout << typeid(*this).name() << " constructor"
             << endl;
```

```
    }
```

```
    ~Announce() {
```

```
        cout << typeid(*this).name() << " destructor"
             << endl;
```

```
    }
```

```
};
```

```
class X : public Announce<0> {
```

```
    Announce<1> m1;
```

```
    Announce<2> m2;
```

```
public:
```

```
    X() { cout << "X::X()" << endl; }
```

```
    ~X() { cout << "X::~X()" << endl; }
```

```
};
```

```
int main() { X x; } ///:~
```

```
Announce<0> constructor
```

```
Announce<1> constructor
```

```
Announce<2> constructor
```

```
    X::X()
```

```
    X::~X()
```

```
Announce<2> destructor
```

```
Announce<1> destructor
```

```
Announce<0> destructor
```

mostenire multipla

```
//: C08:RTTIandMultipleInheritance.cpp
#include <iostream>
#include <typeinfo>
using namespace std;

class BB {
public:
    virtual void f() {}
    virtual ~BB() {}
};

class B1 : virtual public BB {};
class B2 : virtual public BB {};
class MI : public B1, public B2 {};

int main() {
    BB* bbp = new MI; // Upcast
    // Proper name detection:
    cout << typeid(*bbp).name() << endl;
    // Dynamic_cast works properly:
    MI* mip = dynamic_cast<MI*>(bbp);
    // Can't force old-style cast:
    //! MI* mip2 = (MI*)bbp; // Compile error
} ///:~
```

folosirea RTTI

- programatorii din lb. imperative sunt tentati sa foloseasca RTTI si cu switch pentru procesarea ierarhiei
- a nu se folosi in felul acesta
- sa se foloseasca functiile virtuale in primul rand, se organizeaza codul cu aceasta idee in minte
- se foloseste RTTI doar cand e necesar

- daca clasa de baza nu este sub controlul programatorului (altcineva o creeaza, e intr-o librarie, etc.)
 - si e nevoie de o functie virtuala definita
 - atunci o solutie este RTTI
 - se deriveaza intr-o alta clasa
 - se defineste o functie noua
 - se detecteaza in cod daca obiectul e de tipul respectiv sau nu, daca da se executa functia noua

- adaugarea unei functii in clasa de baza (folosita doar in una din derivate) irita ceilalti programatori, celelalte clase au o functie nefolosita (si fara sens)
- interfata complicata
- eficienta: codul scris este corect dar pentru un anumit subtip este foarte lent, se ia subtipul respectiv (RTTI) si se implementeaza cod eficient pentru acel caz

mecanismul pentru RTTI

- un pointer extra in tabela de functii virtuale a clasei
- acest pointer catre `type_info` struct
- cand se cheama `typeid()`:
 - se obtine pointerul catre `type_info`
 - se returneaza structura `type_info`
- cost: mergem prin doi pointeri (f rapid)

- **dynamic_cast<dest*>(pointer_sursa)**
- se obtine RTTI pentru pointer_sursa si *dest
- se verifica daca tipul sursei este la fel ca dest sau baza pentru dest
- pointerul poate fi diferit (mostenire multipla) daca sursa nu e prima baza in mostenire
- daca mostenire multipla, clase virtuale si dest apare de mai multe ori in mostenire
- se pot compara clase din ierarhii diferite

Exemplu RTTI

```
#include <iostream.h>
#include <typeinfo.h>
class Base {
public:
    Base():hide(0){ }
    void mutate(int i){ hide = i;}
    virtual void print()const
    {cout <<
        "hide in Base = " << hide << endl;}
private:
    int hide;
};
```

```
class One: public Base {
public:
    One():Base(),data(0) { }
    void mutate(int i){ data = i;}
    void print()const {cout <<
        "data in One = " << data << endl;}
    void newfunc() {cout << "Hi\n";}
private:
    int data;
};
```

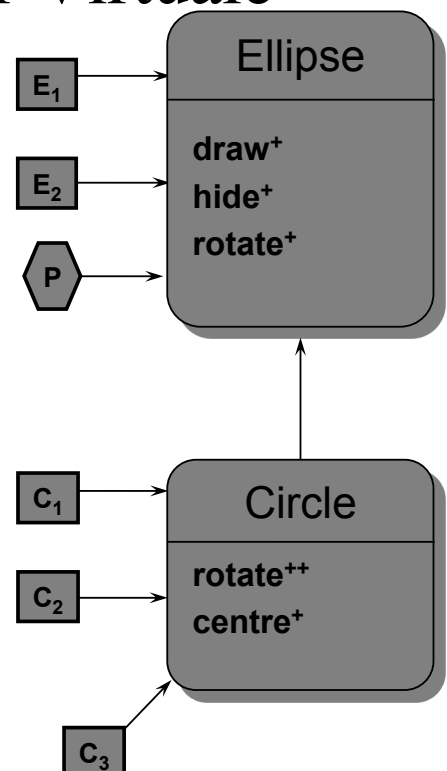
Exemplu RTTI

```
int main ()
{
    Base* bptr; One* derived; One o1; bptr = &o1;
    derived = dynamic_cast<One*>(bptr);
    derived->print();derived->newfunc();
    cout << typeid(bptr).name() << endl << typeid(*bptr).name() << endl;
    if (typeid(*bptr) == typeid(One))
        cout << "type bprt & derived same" << endl;
    cout << typeid(bptr).name();
    cout << " before " << typeid(o1).name() << ": " <<
        (typeid(bptr).before(typeid(o1)) ? true : false) << endl;
}
```

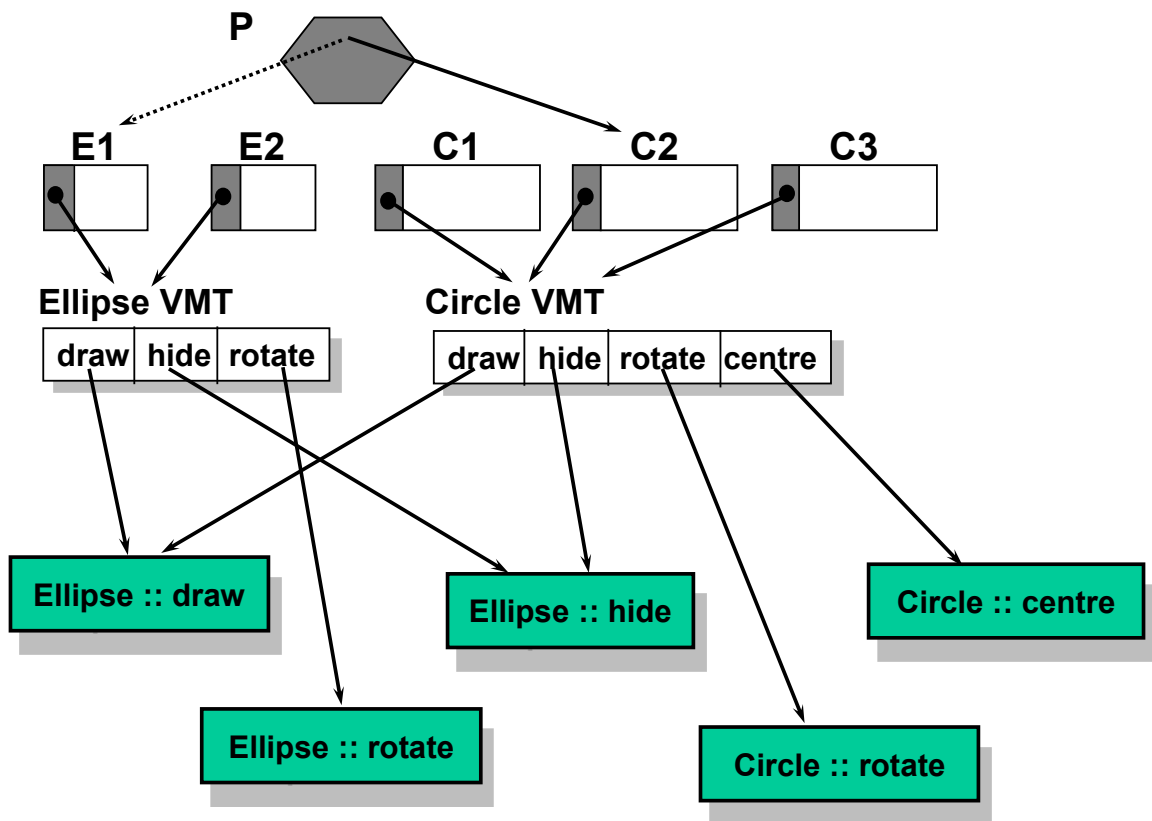
Implementarea Functiilor Virtuale

```
class Ellipse {
// ...
public:
    virtual void draw() const;
    virtual void hide() const;
    virtual void rotate(int);
} E1, E2, *P;
```

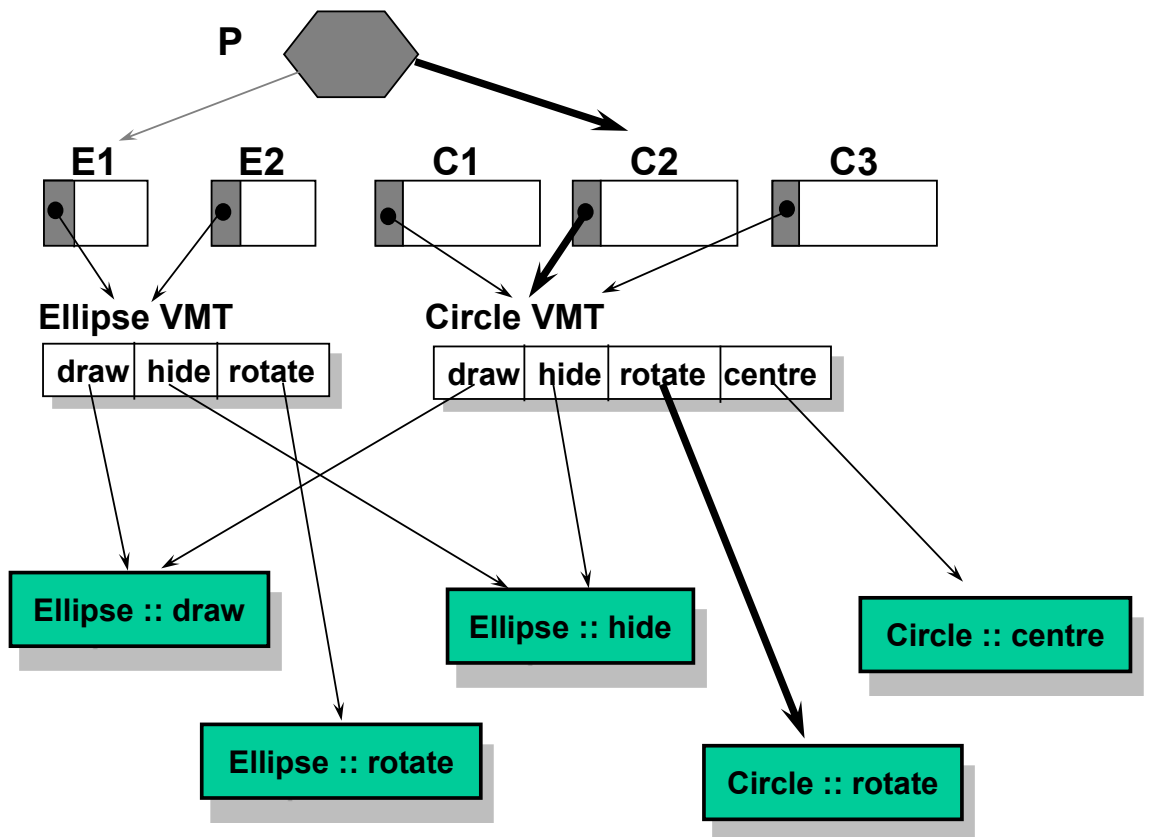
```
class Circle: public Ellipse {
//...
public:
    virtual void rotate(int);
    virtual Point centre();
} C1, C2, C3;
```



Tabelul de metode Virtuale (vtbl)



P->rotate()



const si volatile

- idee: sa se elimine comenzile de preprocesor `#define`
- `#define` faceau substitutie de valoare
- se poate aplica la pointeri, argumente de functii, param de intoarcere din functii, obiecte, functii membru
- fiecare dintre aceste elemente are o aplicare diferita pentru `const`, dar sunt in aceeasi idee/filosofie

- `#define BUFSIZE 100` (tipic in C)
- erori subtile datorita substituirii de text
- `BUFSIZE` e mult mai bun decat “valori magice”
- nu are tip, se comporta ca o variabila
- mai bine: `const int bufsize = 100;`

- acum compilatorul poate face calculele la inceput: “constant folding”: important pt. array: o expresie complicata e calculata la compilare
- `char buf[bufsize];`
- se poate face `const` pe: **char, int, float, double** si variantele lor
- se poate face `const` si pe obiecte

- `const` implica “internal linkage” adica e vizibila numai in fisierul respectiv (la linkare)
- trebuie data o valoare pentru elementul constant la declarare, singura exceptie:
- `extern const int bufsize;`
- in mod normal compilatorul nu alocă spațiu pentru constante, dacă e declarat ca extern alocă spațiu (sa poata fi accesat si din alte parti ale programului)

- pentru structuri complicate folosite cu const se alocă spațiu: nu se știe dacă se alocă sau nu spațiu și atunci const impune localizare (să nu existe coliziuni de nume)
- de aceea avem “internal linkage”

```
// Using const for safety
#include <iostream>
using namespace std;

const int i = 100; // Typical constant
const int j = i + 10; // Value from const expr
long address = (long)&j; // Forces storage
char buff[j + 10]; // Still a const expression

int main() {
    cout << "type a character & CR:";
    const char c = cin.get(); // Can't change
    const char c2 = c + 'a';
    cout << c2;
    // ...
}
```

- dacă știu că variabila nu se schimbă să o declarăm cu const
- dacă încercăm să o schimbăm primim eroare de compilare

- `const` poate elimina memorie si acces la memorie
- `const` pentru agregate: aproape sigur compilatorul alocă memorie
- nu se pot folosi valorile la compilare

```
// Constants and aggregates  
const int i[] = { 1, 2, 3, 4 };  
//! float f[i[3]]; // Illegal  
struct S { int i, j; };  
const S s[] = { { 1, 2 }, { 3, 4 } };  
//! double d[s[1].j]; // Illegal  
int main() {}
```

diferente cu C

- `const` in C: o variabila globala care nu se schimba
- deci nu se poate considera ca valoare la compilare

```
const int bufsize = 100;
```

```
char buf[bufsize];
```

eroare in C

- in C se poate declara cu
`const int bufsize;`
- in C++ nu se poate asa, trebuie extern:
`extern const int bufsize;`

- diferenta:
 - C external linkage
 - C++ internal linkage

- in C++ compilatorul incearca sa nu creeze spatiu pentru const-uri, daca totusi se transmite catre o functie prin referinta, extern etc atunci se creeaza spatiu
- C++: const in afara tuturor functiilor: scopul ei este doar in fisierul respectiv: internal linkage,
- alti identificatori declarati in acelasi loc (fara const) EXTERNAL LINKAGE

poinetri const

- const poate fi aplicat valorii pointerului sau elementului pointat
- const se alatura elementului cel mai apropiat

`const int* u;`

- u este pointer catre un int care este const
- `int const* v;` la fel ca mai sus

pointeri constanti

- pentru pointeri care nu isi schimba adresa din memorie

```
int d = 1;
```

```
int* const w = &d;
```

- w e un pointer constant care arata catre intregi+initializare

const pointer catre const element

```
int d = 1;
```

```
const int* const x = &d; // (1)
```

```
int const* const x2 = &d; // (2)
```

```
//: C08:ConstPointers.cpp
const int* u;
int const* v;
int d = 1;
int* const w = &d;
const int* const x = &d; // (1)
int const* const x2 = &d; // (2)
int main() { } ///:~
```

- se poate face atribuire de adresa pentru obiect non-const catre un pointer const
- nu se poate face atribuire pe adresa de obiect const catre pointer non-const


```
int d = 1;
const int e = 2;
int* u = &d; // OK -- d not const
//! int* v = &e; // Illegal -- e const
int* w = (int*)&e; // Legal but bad practice
int main() { } ///:~
```

constante caractere

```
char* cp = "howdy";
```

daca se incerca schimbarea caracterelor din
“howdy” compilatorul ar trebui sa genereze
eroare; nu se intampla in mod uzual
(compatibilitate cu C)

mai bine:

```
char cp[] = "howdy";
```


si atunci nu ar mai trebui sa fie probleme

argumente de functii, param de intoarcere

- apel prin valoare cu const: param formal nu se schimba in functie
- const la intoarcere: valoarea returnata nu se poate schimba
- daca se transmite o adresa: promisiune ca nu se schimba valoarea la adresa

```
void f1(const int i) {  
i++; // Illegal -- compile-time error  
}
```

cod mai clar echivalent mai jos:

```
void f2(int ic) {  
const int& i = ic;  
i++; // Illegal -- compile-time error  
}
```

```

// Returning consts by value
// has no meaning for built-in types

int f3() { return 1; }
const int f4() { return 1; }

int main() {
    const int j = f3(); // Works fine
    int k = f4(); // But this works fine too!
}

```

```

// Constant return by value
// Result cannot be used as an lvalue

class X { int i;
public: X(int ii = 0); void modify();
};

X::X(int ii) { i = ii; }
void X::modify() { i++; }

X f5() { return X(); }
const X f6() { return X(); }

void f7(X& x) { // Pass by non-const reference
    x.modify();
}

int main() {
    f5() = X(1); // OK -- non-const return value
    f5().modify(); // OK
// Causes compile-time errors:
//! f7(f5());
//! f6() = X(1);
//! f6().modify();
//! f7(f6());
} ///:~

```

DESIGN PATTERNS

Curs OOP 20 mai 2014

Contents:

- ❖ What does the **term** Design Pattern stand for?
- ❖ **Why** should we **study** design patterns?
- ❖ **List** of design patterns and **categories**
- ❖ Presented patterns:
 - Singleton
 - Builder
 - Factory and Abstract Factory
 - ❑ Adapter
 - ❖ Iterator

Design Patterns

What are those?

First time the term of Design pattern was introduced by **Christopher Alexander**.

According to him, a pattern:

- Describes a **recurring problem**
- Describes the **core of a solution**
- Is capable of generating many distinct designs

“Gang of Four” Pattern Structure

- Gang of Four (GoF): Gamma, Johnson, Helm, Vlissides
 - Authors of the popular “Design Patterns” **book**
- A pattern has a **name**
 - e.g., the Singleton pattern
- A pattern documents a **recurring problem**
 - e.g., Issuing requests to objects without knowing in advance what's to be requested or of what object

“Gang of Four” Pattern Structure

- A pattern describes the **core of a solution**
 - e.g., class roles, relationships, and interactions
 - Important: this is different than describing a design
- A pattern considers **consequences** of its use
 - Trade-offs, unresolved forces, other patterns to use

Why study Design Patterns?

- ✓ **reuse solutions** that have worked in the past; why waste time reinventing the wheel?
- ✓ **shared vocabulary** around software design
- ✓ tell a software engineer “I used a Strategy pattern”
- ✓ don’t have to waste time explaining what you mean since you both know the Strategy pattern

Why study Design Patterns?

- ✓ Design patterns provide you not with code reuse but with experience reuse
- ✓ Knowing concepts such as abstraction, inheritance and polymorphism will NOT make you a **good designer**, unless you use those concepts to create **flexible designs** that are maintainable and that can **cope with change**
- ✓ Design patterns can show you how to apply those concepts to achieve those goals

Design Pattern Categories

- Creational patterns
- Structural patterns
- Behavioral patterns
- Concurrency patterns

Creational patterns

Deal with **object creation mechanisms**, trying to create objects in a manner suitable to the situation by somehow controlling this object creation.

Two dominant ideas:

- ❖ **encapsulating knowledge** about which classes the system uses.
- ❖ **hiding** how instances of these classes are created and combined.

Further categorized into **Object-creational** patterns and **Class-creational** patterns, where Object-creational patterns deal with Object creation and Class-creational patterns deal with Class-instantiation.

Structural patterns

Ease the design by identifying a simple way to realize relationships between entities

Behavioral patterns

Identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication

Singleton

Problem : Want to ensure a single instance of a class, shared throughout a program

Context : Need to address initialization versus usage ordering

Solution :

- Provide a **global access method** (static in C++)
- First use of the access method **instantiates** the class
- Constructors for instance can be made private (actually must C++)

Consequences :

- Object is never created if it's never used
- Object is shared efficiently among all uses

Singleton

```
template <class T> class Singleton {
public:
    static T *instance();
private:
    Singleton();
    static T *instance_;
};

// Initialize the static instance pointer
template <class T>
T *Singleton::instance_ = 0;

// Global access point
template <class T>
T *Singleton::instance() {
    // check for existing instance
    if (Singleton<T>::instance_ == 0) {
        // may want a try/catch here
        Singleton<T>::instance_ =
            new Singleton<T>;
    }
    return Singleton<T>::instance_;
};
```

Singleton

- Parameterized by a **concrete type**
- Notice constructor and variable are **private**
- Initialization of static `s_instance` variable
 - Outside class declaration
 - Outside method definition
 - Done before any method in compilation unit is called
- Instance accessor method can then check
 - For a 0 instance pointer
 - And create a new instance if so
- **Same object** is always returned by accessor

Singleton

```
Foo *f1 = Singleton<Foo>::instance();  
  
Foo *f2 = Singleton<Foo>::instance();  
//SAME OBJECT for both f1 and f2
```

- Need a single instance
 - E.g., a common buffer of text tokens from a file
- Shared across multiple points in the code
- Need to share buffer
 - Copying is wasteful
 - Need to refer to same object instance
- What about **deleting** these instances?

Builder

Intent:

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Parse a complex representation, create one of several targets.

Builder

Intent:

- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
- Parse a complex representation, create one of several targets.

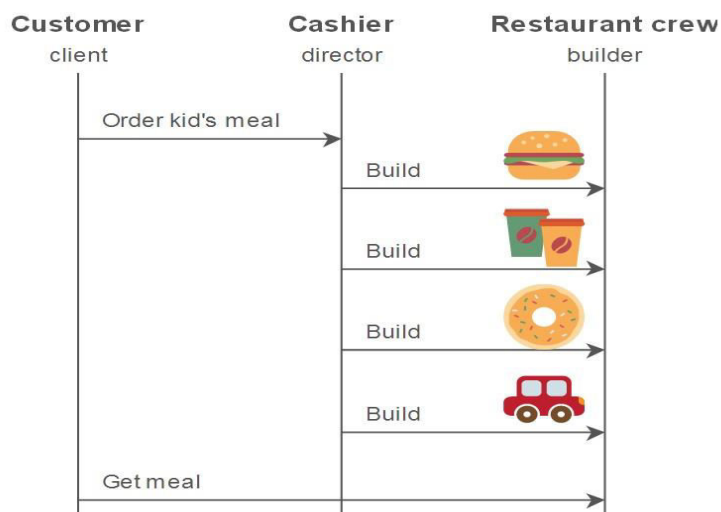
Builder

Example:

This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, Coke, and toy dinosaur). Note that there can be variation in the content of the children's meal, but the construction process is the same.

Builder

Example:



Builder

Example:

Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same. The employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a bag. The drink is placed in a cup and remains outside of the bag. This same process is used at competing restaurants.

Builder

How to use a builder?

1. Decide if a common input and many possible representations (or outputs) is the problem at hand.
2. Encapsulate the parsing of the common input in a Reader class.
3. Design a standard protocol for creating all possible output representations. Capture the steps of this protocol in a Builder interface.

Builder

How to use a builder?

4. Define a Builder derived class for each target representation.
5. The client creates a Reader object and a Builder object, and registers the latter with the former.
6. The client asks the Reader to "construct".
7. The client asks the Builder to return the result.

Builder

Code:

http://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns/Creational_Patterns

(Abstract) Factory

Factory:

A utility class that creates an instance of a class from a family of derived classes

Abstract Factory:

A utility class that creates an instance of several families of classes. It can also return a factory for a certain group.

(Abstract) Factory

Intent:

Provide an interface for creating families of related or dependent objects without specifying their concrete classes

(Abstract) Factory

Example: pasta maker



The code is the pasta maker .Different disks create **different pasta** shapes .These are the factories.
All disks have certain **properties in common** so that they will work with the pasta maker.
All pastas have certain **characteristics in common** that are inherited from the generic “Pasta” object.

(Abstract) Factory

Code:

http://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns/Creational_Patterns#Factory

Adapter

Problem :

Have an object with an interface that's close to but not exactly what we need.
What do we do?

Context:

- Want to **re-use an existing** class
- Can't change its interface
- Impractical to extend class hierarchy more generally

Adapter

Solution:

Wrap a particular class or object with the interface needed (2 forms: class form and object forms)

Consequences:

Implementation you're given gets the interface you want

Iterator

Intent:

Want to access **aggregated elements sequentially**, like traverse a list of names and print them out.

Don't want to know details of how they're stored, in a linked list, or an array, or a balanced binary tree

Solution core:

Provide a **separate interface** for iteration over each container

Iterator

Consequences:

- Frees user from knowing details of how elements are stored
- Decouples **containers from algorithms** (crucial in C++ STL)

Examples:

C++ pointers, C++ STL `list<int>::iterator`

Iterator

Specifications:

- Each container may have a **different iterator type**
- Iterator **knows the internals** of the container
- Object-oriented form shown below (for user-defined types)
- Slightly different with built-in types, templates
 - E.g., no inheritance relationship, may use traits, etc.

Iterator

Implementation:

Object-oriented version of iterator is natural to implement as a class in C++.

Constructor stores passed pointer to C-style string `s`, positions `current_` at `s` (string),

first (re)positions iterator at the start of the string

next moves iterator to the next position

is_done iterating whether iterator is at the end of the string

current_item returns a pointer to the character at the current iterator position

Iterator

Code:

```
class StringIterator {
public:

    StringIterator (char * s)
        : s_ (s), current_ (s) {}

    void first () {
        current_ = s_;
    }

    void next () {
        ++current_;
    }

    bool is_done () {
        return *current_ == 0;
    }

    char * current_item () {
        return current_;
    }

private:
    char *s_;
    char *current_;
};
```

Iterator

- ▶ Iterators naturally support use in looping constructs

- ▶ **first** is used to initialize
- ▶ **is_done** used for loop test
- ▶ **next** used to increment
- ▶ **current_item** is used in loop comparison

- ▶ loop completes => function shown has

- ▶ Iterated through entire string
- ▶ Counted occurrences of the character value in the string
- ▶ Returned the occurrence count

```
unsigned int letter_count
(StringIterator& si, char c)
{
    unsigned int count = 0;
    for (si.first ();
        ! si.is_done ();
        si.next ())
        if (*si.current_item () == c)
            ++count;
    return count;
}
```

Cursul de programare orientata pe obiecte

Seria 13

Saptamana 13, 13 mai 2014

Andrei Paun

Birocratie

- Examen 20 iunie ora 14:00, **semnături**
- Examen LFA 13 iunie ora 9:00
- Bomberman specificatii
- Laboratoare, etc.

Bombberman

Din regulile mentionate la cursul 1, 18 feb 2014

- Laborator: implementare a unui proiect bonus de dimensiuni medii, pe fiecare subgrupa se va face un clasament si primul si al doilea proiect de acest fel primesc 1 punct bonus la examen
- Cel mai bun proiect pentru serie primeste 2 puncte bonus la examen

- Extra: primele 6 implementari (locurile 1, 2, si 3 de la ambele serii) vor intra in competitie cu implementarea “profilor”, daca bateti strategia noastra luati 10 la examen

Reguli

- Implementarea se va verifica in cursul saptamanii 26-30 Mai 2014 in timpul laboratorului de catre tutorele de laborator. In data de 25 mai 2014 inainte de miezul noptii se va trimite codul sursa atasat la e-mail catre tutorele de laborator si cu CC catre apaun@fmi.unibuc.ro cu urmatoarele informatii: Nume, prenume, subgrupa, ora si ziua laboratorului. O arhiva cu fisierele necesare compilarii se va trimite si pe moodle. Studentii care nu trimit acest e-mail pana la ora 23:59 in ziua de 25 mai 2014 NU vor intra in competitie.
- Pentru subgrupele cu 1 sau 2 studenti aflati in competitie: strategia trebuie sa miste omuletul, sa evite peretii (adica sa nu mearga tot timpul, spre exemplu, la dreapta), sa lase bombe etc. Daca se transmite o implementare prea simpla (se alege random directia de miscare a omuletului, lasa bombe random, merge in cerc continuu, etc. ne rezervam dreptul de a nu acorda niciun punct
- Pentru toate grupele: daca programul are greseli (nu compileaza, nu misca omuletul pe ecran, nu lasa bombe, etc.) studentul nu va primi niciun punct

Alte reguli

- Nu folositi variabile globale
- Nu folositi goto
- Nu folositi variabile de instanta publice
- Numarul variabilelor locale sa fie rezonabil (puteti avea maxim 200 de intregi)
- Intrebari? La pauza

Cuprinsul cursului: 13 mai 2014

- const si volatile
- Static

const si volatile

- idee: sa se elimine comenzile de preprocesor
#define
- #define faceau substitutie de valoare
- se poate aplica la pointeri, argumente de functii,
param de intoarcere din functii, obiecte, functii
membru
- fiecare dintre aceste elemente are o aplicare
diferita pentru const, dar sunt in aceeasi
idee/filosofie

- `#define BUFSIZE 100` (tipic in C)
- erori subtile datorita substituirii de text
- BUFSIZE e mult mai bun decat “valori magice”
- nu are tip, se comporta ca o variabila
- mai bine: `const int bufsize = 100;`

- acum compilatorul poate face calculele la inceput: “constant folding”: important pt. array: o expresie complicata e calculata la compilare
- `char buf[bufsize];`
- se poate face const pe: **char, int, float, double** si variantele lor
- se poate face const si pe obiecte

- const implica “internal linkage” adica e vizibila numai in fisierul respectiv (la linkare)
- trebuie data o valoare pentru elementul constant la declarare, singura exceptie:
`extern const int bufsiz;`
- in mod normal compilatorul nu alocă spațiu pentru constante, dacă e declarat ca extern alocă spațiu (sa poata fi accesat si din alte parti ale programului)

- pentru structuri complicate folosite cu const se alocă spațiu: nu se știe dacă se alocă sau nu spațiu și atunci const impune localizare (sa nu existe coliziuni de nume)
- de aceea avem “internal linkage”

```
// Using const for safety
#include <iostream>
using namespace std;

const int i = 100; // Typical constant
const int j = i + 10; // Value from const expr
long address = (long)&j; // Forces storage
char buff[j + 10]; // Still a const expression

int main() {
    cout << "type a character & CR:";
    const char c = cin.get(); // Can't change
    const char c2 = c + 'a';
    cout << c2;
    // ...
}
```

- daca stim ca variabila nu se schimba sa o declaram cu const
- daca incercam sa o schimbam primim eroare de compilare

- const poate elimina memorie si acces la memorie
- const pentru agregate: aproape sigur compilatorul alocă memorie
- nu se pot folosi valorile la compilare

```
// Constants and aggregates
const int i[] = { 1, 2, 3, 4 };
//! float f[i[3]]; // Illegal
struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };
//! double d[s[1].j]; // Illegal
int main() {}
```

diferente cu C

- const in C: o variabila globala care nu se schimba
- deci nu se poate considera ca valoare la compilare

```
const int bufsize = 100;
char buf[bufsize];
```

eroare in C

- in C se poate declara cu
const int bufsize;
- in C++ nu se poate asa, trebuie extern:
extern const int bufsize;
- diferenta:
 - C external linkage
 - C++ internal linkage

- in C++ compilatorul incearca sa nu creeze spatiu pentru const-uri, daca totusi se transmite catre o functie prin referinta, extern etc atunci se creeaza spatiu
- C++: const in afara tuturor functiilor: scopul ei este doar in fisierul respectiv: internal linkage,
- alti identificatori declarati in acelasi loc (fara const) EXTERNAL LINKAGE

pointeri const

- const poate fi aplicat valorii pointerului sau elementului pointat
- const se alatura elementului cel mai apropiat

`const int* u;`

- u este pointer catre un int care este const
- `int const* v;` la fel ca mai sus

pointeri constanti

- pentru pointeri care nu isi schimba adresa din memorie

`int d = 1;`

`int* const w = &d;`

- w e un pointer constant care arata catre intregi+initializare

const pointer catre const element

```
int d = 1;  
const int* const x = &d; // (1)  
int const* const x2 = &d; // (2)
```

```
//: C08:ConstPointers.cpp  
const int* u;  
int const* v;  
int d = 1;  
int* const w = &d;  
const int* const x = &d; // (1)  
int const* const x2 = &d; // (2)  
int main() { } ///:~
```

- se poate face atribuire de adresa pentru obiect non-const catre un pointer const
- nu se poate face atribuire pe adresa de obiect const catre pointer non-const

```
int d = 1;
const int e = 2;
int* u = &d; // OK -- d not const
//! int* v = &e; // Illegal -- e const
int* w = (int*)&e; // Legal but bad practice
int main() { } ///:~
```


constante caractere

```
char* cp = "howdy";
```

daca se incerca schimbarea caracterelor din
“howdy” compilatorul ar trebui sa genereze
eroare; nu se intampla in mod uzual
(compatibilitate cu C)

```
mai bine: char cp[] = "howdy";
```

si atunci nu ar mai trebui sa fie probleme

argumente de functii, param de intoarcere

- apel prin valoare cu const: param formal nu se schimba in functie
- const la intoarcere: valoarea returnata nu se poate schimba
- daca se transmite o adresa: promisiune ca nu se schimba valoarea la adresa

```
void f1(const int i) {  
i++; // Illegal -- compile-time error  
}
```

cod mai clar echivalent mai jos:

```
void f2(int ic) {  
const int& i = ic;  
i++; // Illegal -- compile-time error  
}
```

```
// Returning consts by value  
// has no meaning for built-in types
```

```
int f3() { return 1; }  
const int f4() { return 1; }
```

```
int main() {  
    const int j = f3(); // Works fine  
    int k = f4(); // But this works fine too!  
}
```

```

// Constant return by value
// Result cannot be used as an lvalue

class X { int i;
public: X(int ii = 0); void modify();
};

X::X(int ii) { i = ii; }
void X::modify() { i++; }

X f5() { return X(); }
const X f6() { return X(); }

void f7(X& x) { // Pass by non-const reference
    x.modify();
}

int main() {
    f5() = X(1); // OK -- non-const return value
    f5().modify(); // OK
// Causes compile-time errors:
//! f7(f5());
//! f6() = X(1);
//! f6().modify();
//! f7(f6());
} ///:~

```

- f7() creeaza obiecte temporare, de aceea nu compileaza
- aceste obiecte au constructor si destructor dar pentru ca nu putem sa le “atingem” sunt definite ca si obiecte constante
- f7(f5()); se creeaza ob. temporar pentru rezultatul lui f5(); si apoi apel prin referinta la f7
- ca sa compileze (dar cu erori mai tarziu) trebuie apel prin referinta const

- `f5() = X(1);`
- `f5().modify();`
- compileaza fara probleme, dar procesarea se face pe obiectul temporar (modificarile se pierd imediat, deci aproape sigur este bug)

parametrii de intrare si iesire: adrese

- e preferabil sa fie definiti ca const
- in felul asta pointerii si referintele nu pot fi modificate

// Constant pointer arg/return

```
void t(int*) {}
```

```
void u(const int* cip) {  
    //! *cip = 2; // Illegal -- modifies value  
    int i = *cip; // OK -- copies value  
    //! int* ip2 = cip; // Illegal: non-const  
}
```

```
const char* v() {  
    // Returns address of static character array:  
    return "result of function v()";  
}
```

```
const int* const w() {  
    static int i;  
    return &i;  
}
```

```
int main() {  
    int x = 0;  
    int* ip = &x;  
    const int* cip = &x;  
    t(ip); // OK  
    //! t(cip); // Not OK  
    u(ip); // OK  
    u(cip); // Also OK  
    //! char* cp = v(); // Not OK  
    const char* ccp = v(); // OK  
    //! int* ip2 = w(); // Not OK  
    const int* const ccip = w(); // OK  
    const int* cip2 = w(); // OK  
    //! *w() = 1; // Not OK  
} ///:~
```

- cip2 nu schimba adresa intoarsa din w (pointerul contant care arata spre constanta) deci e ok; urmatoarea linie schimba valoarea deci compilatorul intervine

comparatii cu C

- in C daca vrem param. o adresa: se face pointer la pointer
- in C++ nu se incurajeaza acest lucru: const referinta
- pentru apelant e la fel ca apel prin valoare
 - nici nu trebuie sa se gandeasca la pointeri
 - trimiterea unei adrese e mult mai eficienta decat transmiterea obiectului prin stiva, se face const deci nici nu se modifica

Ob. temporare sunt const

```
//: C08:ConstTemporary.cpp
```

```
// Temporaries are const
```

```
class X {};
```

```
X f() { return X(); } // Return by value
```

```
void g1(X&) {} // Pass by non-const reference
```

```
void g2(const X&) {} // Pass by const reference
```

```
int main() {
```

```
    // Error: const temporary created by f():
```

```
    //! g1(f());
```

```
    // OK: g2 takes a const reference:
```

```
    g2(f());
```

```
} ///:~
```

- In C avem pointeri deci e OK

Const in clase

- const pentru variabile de instanta si
- functii de instanta de tip const
- sa construim un vector pentru clasa respectiva, in C folosim #define
- problema in C: coliziune pe nume

- in C++: punem o variabila de instanta const
- problema: toate obiectele au aceasta variabila, si putem avea chiar valori diferite (depinde de initializare)
- cand se creeaza un const intr-o clasa nu se poate initializa (constructorul initializeaza)
- in constructor trebuie sa fie deja initializat (altfel am putea sa il schimbam in constructor)

- initializare de variabile const in obiecte:
lista de initializre a constructorilor

```
// Initializing const in classes
#include <iostream>
using namespace std;

class Fred {
    const int size;
public:
    Fred(int sz);
    void print();
};

Fred::Fred(int sz) : size(sz) {}
void Fred::print() { cout << size << endl; }

int main() {
    Fred a(1), b(2), c(3);
    a.print(), b.print(), c.print();
} ///:~
```

rezolvarea problemei initiale

- cu static
 - insemna ca nu e decat un singur asemenea element in clasa
 - il facem static const si devine similar ca un const la compilare
 - static const trebuie initializat la declarare (nu in constructor)

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

```
class StringStack {
    static const int size = 100;
    const string* stack[size];
    int index;
public:
    StringStack();
    void push(const string* s);
    const string* pop();
};
```

```
StringStack::StringStack() : index(0) {
    memset(stack, 0, size * sizeof(string*));}
```

```
void StringStack::push(const string* s) {
    if(index < size)
        stack[index++] = s;}
```

```
const string* StringStack::pop() {
    if(index > 0) {
        const string* rv = stack[--index];
        stack[index] = 0;
        return rv; }
    return 0;}
```

```
string iceCream[] = {
    "pralines & cream",
    "fudge ripple",
    "jamocho almond fudge",
    "wild mountain blackberry",
    "raspberry sorbet",
    "lemon swirl",
    "rocky road",
    "deep chocolate fudge"
};
```

```
const int iCsz =
    sizeof iceCream / sizeof *iceCream;
```

```
int main() {
    StringStack ss;
    for(int i = 0; i < iCsz; i++)
        ss.push(&iceCream[i]);
    const string* cp;
    while((cp = ss.pop()) != 0)
        cout << *cp << endl;
}
```


enum hack

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

```
class Bunch {
    enum { size = 1000 };
    int i[size];
};
```

```
int main() {
    cout << "sizeof(Bunch) = " << sizeof(Bunch)
        << ", sizeof(i[1000]) = "
        << sizeof(int[1000]) << endl;
}
```

- in cod vechi
- a nu se folosi cu C++ modern
- static const int size=1000;

obiecte const si functii membru const

- obiecte const: nu se schimba
- pentru a se asigura ca starea obiectului nu se schimba functiile de instanta apelabile trebuie definite cu const
- declararea unei functii cu const nu garanteaza ca nu modifica starea obiectului!

functii membru const

- compilatorul si linkerul cunosc faptul ca functia este const
- se verifica acest lucru la compilare
- nu se pot modifica parti ale obiectului in aceste functii
- nu se pot apela functii non-const

```
//: C08:ConstMember.cpp
```

```
class X {  
    int i;  
public:  
    X(int ii);  
    int f() const;  
};
```

```
X::X(int ii) : i(ii) {}
```

```
int X::f() const { return i; }
```

```
int main() {  
    X x1(10);  
    const X x2(20);  
    x1.f();  
    x2.f();  
} ///:~
```

- toate functiile care nu modifica date sa fie declarate cu const
- ar trebui ca “default” pentru functiile membru sa fie de tip const

```
#include <iostream>
#include <cstdlib> // Random number generator
#include <ctime> // To seed random generator
using namespace std;

class Quoter { int lastquote;
public:      Quoter();
            int lastQuote() const;
            const char* quote();
};

Quoter::Quoter(){ lastquote = -1;
    srand(time(0)); // Seed random number generator
}

int Quoter::lastQuote() const { return lastquote;}

const char* Quoter::quote() {
    static const char* quotes[] = {
        "Are we having fun yet?",
        "Doctors always know best",
        "Is it ... Atomic?",
        "Fear is obscene",
        "There is no scientific evidence "
        "to support the idea "
        "that life is serious",
        "Things that make us happy, make us wise",
    };
    const int qsize = sizeof quotes/sizeof *quotes;
    int qnum = rand() % qsize;
    while(lastquote >= 0 && qnum == lastquote)
        qnum = rand() % qsize;
    return quotes[lastquote = qnum];
}

int main() {
    Quoter q;
    const Quoter cq;
    cq.lastQuote(); // OK
    //! cq.quote(); // Not OK; non const function
    for(int i = 0; i < 20; i++)
        cout << q.quote() << endl;
} ///:~
```

schimbari in obiect din functii const

- “casting away constness”
- se face castare a pointerului this la pointer catre tipul de obiect
- pentru ca in functii const este de tip clasa const*
- dupa aceasta schimbare de tip se modifica prin pointerul this

```
// "Casting away" constness

class Y {
    int i;
public:
    Y();
    void f() const;
};

Y::Y() { i = 0; }

void Y::f() const {
    //! i++; // Error -- const member function
    ((Y*)this)->i++; // OK: cast away const-ness
    // Better: use C++ explicit cast syntax:
    (const_cast<Y*>(this))->i++;
}

int main() {
    const Y yy;
    yy.f(); // Actually changes it!
} ///:~
```

- apare in cod vechi
- nu e ok pentru ca functia modifica si noi credem ca nu modifica
- o metoda mai buna: in continuare

// The "mutable" keyword

```
class Z {  
    int i;  
    mutable int j;  
public:  
    Z();  
    void f() const;  
};  
  
Z::Z() : i(0), j(0) {}  
  
void Z::f() const {  
    //! i++; // Error -- const member function  
    j++; // OK: mutable  
}  
  
int main() {  
    const Z zz;  
    zz.f(); // Actually changes it!  
} ///:~
```

volatile

- e similar cu const
- obiectul se poate schimba din afara programului
- multitasking, multithreading, intreruperi
- nu se fac optimizari de cod
- avem obiecte volatile, functii volatile, etc.

static

- ceva care isi tine pozitia neschimbata
- alocare statica pentru variabile
- vizibilitate locala a unui nume

- variabile locale statice
- isi mentin valorile intre apelari
- initializare la primul apel

```
#include "../require.h"
#include <iostream>
using namespace std;

char oneChar(const char* charArray = 0) {
    static const char* s;
    if(charArray) {
        s = charArray;
        return *s;
    }
    else
        require(s, "un-initialized s");
    if(*s == '\0')
        return 0;
    return *s++;
}

char* a = "abcdefghijklmnopqrstuvwxyz";

int main() {
    // oneChar(); // require() fails
    oneChar(a); // Initializes s to a
    char c;
    while((c = oneChar()) != 0)
        cout << c << endl;
} ///:~
```

obiecte statice

- la fel ca la tipurile predefinite
- avem nevoie de constructorul predefinit

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

class X {
    int i;
public:
    X(int ii = 0) : i(ii) {} // Default
    ~X() { cout << "X::~~X()" << endl; }
};

void f() {
    static X x1(47);
    static X x2; // Default constructor required
}

int main() {
    f();
} ///:~
```


destructori statici

- cand se termina main se distrug obiectele
- de obicei se cheama `exit()` la iesirea din main
- daca se cheama `exit()` din destructor e posibil sa avem ciclu infinit de apeluri la `exit()`
- destructorii statici nu sunt executati daca se iese prin `abort()`

- daca avem o functie cu obiect local static
- si functia nu a fost apelata, nu vrem sa apelam destructorul pentru obiect neconstruit
- C++ tine minte ce obiecte au fost construite si care nu

<pre> #include <fstream> using namespace std; ofstream out("statdest.out"); // Trace file class Obj { char c; // Identifier public: Obj(char cc) : c(cc) { out << "Obj::Obj() for " << c << endl; } ~Obj() { out << "Obj::~Obj() for " << c << endl; } }; Obj a('a'); // Global (static storage) // Constructor & destructor always called void f() { static Obj b('b'); } void g() { static Obj c('c'); } </pre>	<pre> int main() { out << "inside main()" << endl; f(); // Calls static constructor for b // g() not called out << "leaving main()" << endl; } ///:~ </pre>
	<pre> Obj::Obj() for a inside main() Obj::Obj() for b leaving main() Obj::~Obj() for b Obj::~Obj() for a </pre>

static pentru nume (la linkare)

- orice nume care nu este intr-o clasa sau functie este vizibil in celalalte parti ale programului (external linkage)
- daca e definit ca static are internal linkage: vizibil doar in fisierul respectiv
- linkarea e valabila pentru elemente care au adresa (clase, var. locale nu au)

- `int a=0;`
- in afara claselor, functiilor: este var globala, vizibila pretutindeni
- similar cu: `extern int a=0;`
- `static int a=0; // internal linkage`
- nu mai e vizibila pretutindeni, doar local in fisierul respectiv

```
//{L} LocalExtern2
```

```
#include <iostream>
```

```
int main() {
```

```
    extern int i;
```

```
    std::cout << i;
```

```
} ///:~
```

```
//: C10:LocalExtern2.cpp {O}
```

```
int i = 5;
```

```
///:~
```

functii extern si static

- schimba doar vizibilitatea
 - `void f();` similar cu `extern void f();`
 - restrictiv:
 - `static void f();`
-
- alti specificatori:
 - `auto`: aproape nefolosit; spune ca e var. locala
 - `register`: sa se puna intr-un registru

variabile de instanta statice

- cand vrem sa avem valori comune pentru toate obiectele
- static

```
class A {  
    static int i;  
    public:  
    //...  
};
```

```
int A::i = 1;
```

- `int A::i = 1;`
- se face o singura data
- e obligatoriu sa fie facut de creatorul clasei, deci e ok
-

```

#include <iostream>
using namespace std;

int x = 100;

class WithStatic {
    static int x;
    static int y;
public:
    void print() const {
        cout << "WithStatic::x = " << x << endl;
        cout << "WithStatic::y = " << y << endl;
    }
};

int WithStatic::x = 1;
int WithStatic::y = x + 1;
// WithStatic::x NOT ::x

int main() {
    WithStatic ws;
    ws.print();
}

```

// Static members & local classes

```

#include <iostream>
using namespace std;

```

// Nested class CAN have static data members:

```

class Outer {
    class Inner {
        static int i; // OK
    };
};

```

```

int Outer::Inner::i = 47;

```

// Local class cannot have static data members:

```

void f() {
    class Local {
    public:
        //! static int i; // Error
        // (How would you define i?)
    } x;
}

```

```

int main() { Outer x; f(); } ///:~

```

functii membru statice

- nu sunt asociate cu un obiect, nu au this

```
class X {  
    public:  
        static void f(){};  
};
```

```
int main() {  
    X::f();  
} ///:~
```

Despre examen

- Descrieți pe scurt funcțiile șablon (template).

```
#include <iostream.h>  
class problema  
{    int i;  
    public: problema(int j=5){i=j;}  
        void schimba(){i++;}  
        void afiseaza(){cout<<"starea curenta "<<i<<"\n";}  
};  
problema mister1() { return problema(6);}  
void mister2(problema &o)  
{    o.afiseaza();  
    o.schimba();  
    o.afiseaza();  
}  
int main()  
{    mister2(mister1());  
    return 0;  
}
```

```

#include<iostream.h>
class B
{ int i;
  public: B() { i=1; }
          virtual int get_i() { return i; } };
class D: virtual public B
{ int j;
  public: D() { j=2; }
          int get_i() {return B::get_i()+j; } };
class D2: virtual public B
{ int j2;
  public: D2() { j2=3; }
          int get_i() {return B::get_i()+j2; } };
class MM: public D, public D2
{ int x;
  public: MM() { x=D::get_i()+D2::get_i(); }
          int get_i() {return x; } };
int main()
{ B *o= new MM();
  cout<<o->get_i()<<"\n";
  MM *p= dynamic_cast<MM*>(o);
  if (p) cout<<p->get_i()<<"\n";
  D *p2= dynamic_cast<D*>(o);
  if (p2) cout<<p2->get_i()<<"\n";
  return 0;
}

```

```

#include <iostream.h>
#include <typeinfo>
class B
{ int i;
  public: B() { i=1; }
          int get_i() { return i; }
};
class D: B
{ int j;
  public: D() { j=2; }
          int get_j() {return j; }
};
int main()
{ B *p=new D;
  cout<<p->get_i();
  if (typeid((B*)p).name()=="D*")
  cout<<((D*)p)->get_j();
  return 0;
}

```



```

#include<iostream.h>
template<class T, class U>
T f(T x, U y)
{ return x+y;
}
int f(int x, int y)
{ return x-y;
}
int main()
{ int *a=new int(3), b(23);
  cout<<*f(a,b);
  return 0;
}

```

```

#include<iostream.h>
class A
{ int x;
  public: A(int i=0) { x=i; }
         A operator+(const A& a) { return x+a.x; }
         template <class T> ostream& operator<<(ostream&);
};
template <class T>
ostream& A::operator<<(ostream& o) { o<<x; return o; }
int main()
{ A a1(33), a2(-21);
  cout<<a1+a2;
  return 0;
}

```

