V. Clase

Tipul *class*

- extinde noţiunea de structură
- grupează
  - date - variabile (membri)
  - cod - funcţii (metode)
- reprezentare internă
  - cum putem pune împreună părţi de date şi cod?
  - răspuns: nu putem

Exemplu

```
class Tab {
    int n;
    int tab[20];
public:
    Tab() {n=0;}
    void add(int);
    void afisare();
};
```

Exemplu (cont.)

printf("Dimensiune: %d\n",sizeof(Tab));

- rezultat afişat

```
Dimensiune: 84
```

- deci o variabilă de tip Tab conţine doar membrii de date
- care este legătura cu metodele clasei?
- asigurată de compilator

Pointerul *this*

- reţine adresa obiectului curent
  - pentru ca membrii de date să poată fi accesaţi din cadrul metodelor clasei
- NU este membru al clasei
  - am văzut deja că dimensiunea unui obiect este suma dimensiunilor membrilor de date
  - când se apelează o metodă a clasei, primeşte adresa obiectului curent (*this*)

Accesul la membrii de date

- din interiorul unei metode - prin pointerul *this*
- cum se ajunge la el?
- Visual Studio - este pus în registrul ecx
  - alternativ - la adresa [ebp-8]
  - dar adresa poate varia în funcţie de versiunea compilatorului

Exemplu

```
void Tab::add(int e) {
    _asm {
        mov eax,[ecx]
        inc eax
        mov edx,e
        mov [ecx+eax*4],edx
        mov [ecx],eax
    }
}
```

Membri publici şi privaţi

- de ce avem membri privaţi?
- restricţii impuse de programator
- scop - evitarea erorilor de programare
- teoretic putem folosi doar membri publici
  - dar nu este recomandabil
- codul scris în limbaj de asamblare nu are restricţiile impuse de compilator
  - putem accesa membrii privaţi

Exemplu

```
Tab t;
t.add(5);
t.add(2);
t.afisare();
_asm {
    mov dword ptr t[4],16
}
t.afisare();
```

Rezultate

- prima afişare

```
5  2
```

- a doua afişare

```
16  2
```

- s-a realizat o modificare la deplasamentul 4
  - adică tab[0]
  - a primit valoarea 16

Dar din C++?

```
Tab t;
int *p;
t.add(5);
t.add(2);
t.afisare();
p=(int*)&t;
p[2]=12;
t.afisare();
```

Rezultate

- prima afişare

```
5  2
```

- a doua afişare

```
5  12
```

- a fost modificat tab[1]
  - de ce nu tab[2]?

Cum se explică?

- compilatorul poate bloca accesul la membrii privaţi
  - dacă folosim numele acestora
- în exemplul anterior am accesat adresa corespunzătoare unui membru privat
  - adresele se calculează la momentul execuţiei
  - când compilatorul nu mai are nici o influenţă
- nu e recomandabil să procedăm astfel

Constructori

- constructor - metodă specială, apelată automat imediat după ce obiectul este creat
  - nu este corect spus că obiectul este creat de constructor
- rol - iniţializarea membrilor de date
- nu trebuie apelat în mod explicit
- nu returnează nimic

Destructori

- destructor - similar constructorului
- apelat automat la distrugerea obiectului
  - nu şi în alte situaţii
- când are sens să definim un destructor?
  - atunci când trebuie realizate unele acţiuni suplimentare înainte de distrugerea obiectului
  - cel mai des - eliberare de memorie alocată dinamic

Moştenire

- în ce ordine se apelează constructorii în cazul claselor derivate?

1. constructorii claselor de bază
   - în C++ este permisă moştenirea multiplă
2. constructorii membrilor de date
3. constructorul clasei curente

- apelarea destructorilor - în ordine inversă

Exemplu

```
int n=0;
class A {
public:
   A() {n++;printf("A\t%d\n",n);}
};
class AA: public A {
public:
   AA() {n++;printf("AA\t%d\n",n);}
};
```

Exemplu (cont.)

```
class B {
public:
    B() {n++;printf("B\t%d\n",n);}
};
class BB: public B {
public:
    BB() {n++;printf("BB\t%d\n",n);}
};
```

Exemplu (cont.)

```
class C {
public:
   C() {n++;printf("C\t%d\n",n);}
};
class D: public AA, public BB {
   C c;
public:
   D() {n++;printf("D\t%d\n",n);}
};
```

Exemplu (cont.)

D x;

- rezultat afişat

```
A     1
AA    2
B     3
BB    4
C     5
D     6
```

Ordinea membrilor de date

- similar cu ordinea apelării constructorilor

1. membrii definiţi în clasele de bază
2. membrii definiţi în clasa curentă

Exemplu

```
class A {
public:
    int a;
};
class AA: public A {
public:
    int aa;
};
```

Exemplu (cont.)

```
class B {
public:
    int b;
};
class BB: public B {
public:
    int bb;
};
```

Exemplu (cont.)

```
class C {
public:
    int c;
};
class D: public AA, public BB {
public:
    C c;
};
```

Exemplu (cont.)

```
D x;
printf("x:\t%p\n",&x);
printf("a:\t%p\n",&x.a);
printf("aa:\t%p\n",&x.aa);
printf("b:\t%p\n",&x.b);
printf("bb:\t%p\n",&x.bb);
printf("c:\t%p\n",&x.c);
```

Exemplu (cont.)

- rezultat afişat

```
x:  0012FF14
a:  0012FF14
aa: 0012FF18
b:  0012FF1C
bb: 0012FF20
c:  0012FF24
```

Apelul metodelor

- similar cu apelul funcţiilor
- deosebiri
  - numele clasei trebuie să însoţească numele metodei
  - apelantul trebuie să pună în registrul ecx adresa obiectului curent (*this*)
  - la final, parametrii sunt eliminaţi de pe stivă de către metoda respectivă (nu de apelant)

Exemplu

```
Tab t;
// t.add(5) - apel din limbaj de asamblare
_asm {
   lea ecx,t
   push dword ptr 5
   call Tab::add
}
```

Apelul metodelor (cont.)

- excepţie - metodele cu număr variabil de parametri
  - pointerul *this* este pus pe stivă (nu mai este în ecx) ca un parametru ascuns
  - urmează pe stivă adresa obiectului de returnat
    - dacă nu este un tip elementar
  - apoi urmează parametrii "reali"
  - la final, parametrii de pe stivă sunt eliminaţi de către apelant

Accesul la membrii statici (1)

- membri de date statici
  - sunt comuni pentru toate obiectele clasei
  - deci nu pot fi accesaţi prin pointerul *this* al unui obiect
  - de fapt, sunt variabile globale
  - singurul mod de acces - folosindu-le numele
    - prefixat de numele clasei

Accesul la membrii statici (2)

- exemplu

```
class A {
   static int s;
public:
   void inc_s()
   { _ asm { inc A::s } }
};
```

Accesul la membrii statici (3)

- metode statice
  - apelul - tot prin nume
    - prefixat de numele clasei
    - la fel ca la metodele obişnuite
  - la apel nu se primeşte nici o adresă a vreunui obiect (pointer *this*)
    - deci nu pot fi accesaţi decât membrii statici de date
  - la final, parametrii de pe stivă sunt eliminaţi de către apelant

Accesul la membrii statici (4)

- exemplu

```
class A {
public:
    static void f() {
        //...
        }
};
```

Accesul la membrii statici (5)

```
A a;
A::f();
a.f();
```

- apelul se poate face în ambele moduri
- oricum, nici în al doilea caz nu se transmite adresa obiectului a (pointer *this*)

Metode - parametri

- de multe ori, parametrii sunt obiecte ai aceleiaşi clase
- ce putem transmite pe stivă
  - întregul obiect - ineficient
  - adresa sa (pointer sau referinţă)
- de preferat - referinţă
  - nu consumă memorie ca un obiect întreg
  - mai uşor de lucrat decât cu pointeri

Metode - valori returnate

- dacă sunt tipuri elementare - în `eax`
- dacă sunt clase - aceleaşi variante ca la parametri
  - preferăm tot referinţele
  - dar uneori nu este indicat
- în cazul claselor, se returnează adresa obiectului în care a fost depus rezultatul
  - chiar dacă obiectul nu are mai mult de 8 octeţi

Exemplu

```
class A {...};
A A::f() {
   A tmp;
   // calcul valoare de returnat
   return tmp;
}
A a1,a2,a3;
a3=a1.f()+a2.f();
```

Exemplu (cont.)

- la execuţia instrucţiunii return, variabila locală tmp este copiată în obiectul rezultat
  - acesta va fi menţinut de compilator cât timp este necesar
- dacă tipul returnat ar fi referinţă, s-ar returna direct adresa variabilei tmp
  - variabila tmp din primul apel ar putea fi suprascrisă în al doilea apel

V.1. Clase şi conversii de tip

Clase

- se pot converti între ele obiecte din clase aflate într-o relaţie de moştenire?

```
class A {
   int x;
};
class B: public A {
   int y;
};
```

Conversii implicite

A a;

B b;

a=b;

- corect - obiectul a primeşte valoarea obiectului de tip A conţinut în b

b=a;

- eroare de compilare - nu putem da valori tuturor membrilor obiectului b

Conversii explicite

- cum putem face ca instrucţiunea b=a; să fie corectă?
- constructor de copiere

```
B::B(A&) {...}
```

- operator de atribuire

```
void B::operator=(A&) {...}
```

Conversii explicite (cont.)

- care din cele două metode anterioare va fi utilizată?
- oricare este definită
- dacă sunt definite ambele, este apelat operatorul de atribuire

Pointeri la obiecte

A a,*pa;

B b,*pb;

pa=&b;

- corect

pb=&a;

- eroare la compilare

pb=(B*)&a;

- corect - conversie explicită