PROGRAMARE ORIENTATĂ OBIECTE

Curs 5

Programarea orientată pe obiecte in C++

Prieteni
Pointeri la membri
Supraîncarcarea operatorilor

Prieteni

- Ascunderea membrilor unei clase este anulată pentru prietenii acelei clase, care pot fi funcţii sau alte clase.
- O funcţie prietenă are acces la toti membrii clasei, inclusiv la cei privaţi ori protejaţi.
- Regula este valabilă pentru toti membri unei clase prietene
- Pentru fiecare prieten al unei clase, există o declaraţie în interiorul clasei, ***`friend`***.
  - *Funcţii prietene care apartin domeniului global*
  - *Funcţii prietene care sunt membre ale unei clase*
  - *Clase prietene*

Funcții prietene globale

```
class Time
{
        int min, sec, hour;
public:
        Time(void) : min(0), sec(0), hour(0){};
        friend int min(Time& t);
        friend int sec(Time& t);
        friend int hour(Time& t);
}

inline int min(Time& t) { return t.min; }
inline int sec(Time& t) { return t.sec; }
inline int hour(Time& t) { return t.hour; }

Time t;
cout << "min: "<<min(t)<<", sec: "<<sec(t)<<", hour:
"<<hour(t)<<endl;
```

- Funcţiile prietene sunt utile pentru supraîncărcarea
- operatorilor

Funcții membru prietene

- O funcţie membru a unei clase poate fi funcţie prietenă pentru altă clasă

```
class XXX {
public:
        void f(void);
};

class YYY {
        int nr;
public:
        YYY(int n): nr(n){}
        friend void XXX::f(void);
};

void XXX::f(void) {
        YYY ob(10);
        cout <<ob.nr<<endl;
}
```

Clase prietene

- Declarând o clasă prietenă în cadrul unei alte clase, se extinde calitatea de prieten la toate funcţiile membru

```
class YYY
{
        friend class XXX;
}
```

Pointeri la membri

- Se presupune tipul de date:
  - `typedef  void (*PtrFuncţie)(int);`
- şi declaraţiile pentru funcţiile *fct()* şi *push()*
  - `void fct(int);`
  - `void Stiva::Push(int);`
- Deşi aparent cele două funcţii au aceeaşi structură ele nu sunt de acelaşi tip, prin urmare:
  - `PtrFunctie pf1 = fct;  //corect`
  - `PtrFunctie pf2 = &Stiva::Push;  //eroare`
- A doua funcţie, pe lângă argumentul `int` mai primeşte ca argument implicit pointerul this
- Pentru a defini un pointer la funcţia `Stiva::Push()` se va utiliza un concept nou: ***pointer la membru***

Pointeri la membri

- Un pointer la membru al unei clase X se defineşte folosind construcţia **X::***
  - `typedef void (Stiva::*PFmembru)(int);`
  - `PFmembru pf3 = &Stiva::Push;`
- Utilizarea pointerilor la membri (dereferenţierea lor) se face cu ajutorul unor operatori speciali:
  - `.*` sau `->*`
- Cu operatori de mai sus se selectează un membru pe baza unui pointer la membru
- Se foloseşte operatorul **`.*`** când obiectul este specificat prin nume
- Se foloseşte operatorul **`->*`** când obiectul este specificat prin pointer.
- Obiectul apare ca operand stâng iar membrul ca operand drept

Pointeri la membri

- Conceptul de pointer la membru este valabil şi pentru membri de tip dată (adrese relative faţă de începutul zonei de memorie alocate obiectului)

```
class X
{
public:
        double d;
        X(void):d(0){}
        void fct(void)
        {
                cout <<"\nS-a apelat fct\n";
        }
};
```

```
int main(void)
{
        X obj, obj4; obj4.d = 1;
        double *pd1;
        double X::*pd2;
        void (X::*pf3)(void);

        pd1 = &obj.d;
        pd2 = &X::d;
        pf3 = &X::fct;

        cout <<*pd1<<obj.*pd2<<obj4.*pd2;
        (obj.*pf3)();

        X *px = new X();
        pd1 = &px->d;

        cout <<*pd1<<px->*pd2;
        (px->*pf3)();

        return 0;
}
```

Supraîncărcarea operatorilor

- În fiecare domeniu au fost dezvoltate notaţii, convenţii pentru a discuta, prezenta diverse concepte într-un mod cat mai convenabil.
- Ex.

```
x + y * z;
```

este mult mai clar decât

multiplică y cu z si adună rezultatul la x

- C++ are un set de operatori pentru tipurile de date predefinite
- Pentru tipurile de date definite de utilizatori operatorii trebuiesc redefiniti
- Ex. Dacă este nevoie de numere complexe, matrici algebrice sau siruri de caractere se vor defini clase pentru reprezentare

Supraîncărcarea operatorilor

- Definirea operatorilor pentru astfel de clase permite programatorului utilizarea unor notaţii convenţionale si convenabile pentru manipularea obiectelor comparativ cu utilizarea funcţiilor clasice.

```
class complex
{
        double re,im;
public:
        complex(double r, double i) : re(r), im(i) {}
        complex operator+(complex);
        complex operator*(complex);
};
```

- Au fost definiti operatorii `complex::operator+()` şi `complex::operator*()` pentru a furniza semnificatie `+, *`.
- Ex. Fie `complex b, c;`

  `b+c` inseamna de fapt `b.operator+(c)`

Introducere

- Regulile de prioritate a operatorilor se respectă

```
void f(void)
{
    complex a = complex(1, 3.1);
    complex b = complex(1.2, 2);
    complex c = b;
    a = b + c;
    b = b + c * a;
    c = a * b + complex(1, 2);
}
```

- Astfel `b=b+c*a` înseamnă `b=b+(c*a)` nu `b=(b+c)*a`
- Utilizarea operatorilor definiti de utilizatori nu este restrictionată doar la tipuri concrete. Astfel proiectarea unor interfeţe generale si abstracte poate conduce la supraîncărcarea operatorilor `->, [ ], ( )`

Funcții operator

- Pot fi definite urmatoarele funcţii operator

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

- Următorii operatori nu pot fi definiţi de utilizator

  `::` (apartenenta),
  `.` (selectare membru)
  `.*` (selectare membru prin intermediul unui pointer la functie).

- Operatorii `=, ->, ->*, ( ), [ ]` pot fi supraîncărcaţi numai ca funcţii membru.
- Nu este posibil definirea de noi operatori, în schimb se recomandă folosirea funcţiilor. Ex. `pow()` în loc de `**`

Functii operator

- Denumirea funcţiei operator este cuvântul cheie *operator* urmat de operatorul insusi, ex. *operator<<.*
- O functie operator este declarata şi apelată ca orice altă funcţie. Folosirea operatorului este o prescurtare a unui apel explicit a functiei operator

```
void f(complex a, complex b)
{
        complex c = a + b;            //prescurtare
        complex d = a.operator+(b);   //apel explicit
}
```

- Pentru orice tip, fundamental sau abstract, compilatorul are definite variante implicite ale lui *operator=* (atribuire) si *operator&* (adresa lui …)

Operatori binari si unari

- Operatorii binari pot fi definiti atât de funcţii membre nestatice având un singur argument sau functii nemembre cu doua argumente.
- Pentru fiecare operator `@`, `aa@bb` poate fi interpretat ca `aa.operator@(bb)` sau `operator@(aa, bb)`
- Dacă sunt definite amândouă algoritmul de potrivire va determina care sa fie folosită

```
class X {
public:
        void operator+(int);
        X(int);
};
void operator+(X, X);
void operator+(X, double);
void f(X a) {
        a+1;    //a.operator+(1)
        1+a;    //::operator+(X(1),a)
        a+1.0; //::operator+(a,1.0)
}
```

Operatori binari si unari

- Operatorii unari atât cei prefixati cât şi cei postfixati, pot fi definiţi atât de funcţii membre nestatice fără argument cât si de funcţii nemembre având doar un argument.
- Pentru orice operator prefixat @, `@aa` poate fi interpretat ca `aa.operator@()` sau ca `operator@(aa)`.
- Daca ambele sunt definite, regulile de potrivire determina care va fi folosita.
- Pentru orice operator postfixat @, `aa@` poate fi interpretat ca `aa.operator@(int)` sau ca `operator@(aa,int)`.
- Daca ambele sunt definite regulile de potrivire determina care va fi folosita.
- Un operator poate fi declarat doar pentru sintaxa definita pentru el.

Operatori binari si unari

```
class X
{
        //functii membre (pointerul this este implicit):
        X*operator&(void);          //unar prefixat &(adresa ...)
        X operator&(X);             //binar&(si)
        X operator++(int);          //postfixat de incrementare
        X operator&(X, X);          //eroare: ternar
        X operator/(void);          //eroare: unar/
};
//functii nemembre:
X operator-(X);             //prefixat unar minus
X operator-(X, X);          //binar minus
X operator--(X&, int);      //postfixat decrementare
X operator-(void);          //eroare: nici un operand
X operator-(X, X, X);       //eroare:ternar
X operator%(X);             //eroare:unar %
```

Operatori unari - detaliere

- Operatorii unari au un singur argument: obiectul asupra caruia se aplică. Acest argument poate fi dat explicit, dacă se supraîncarcă o variantă nemembru, sau implicit prin pointerul ***this***, dacă se supraîncarcă o variantă membru
- Fie clasa *Time:*

```
class Time {
        int min, sec, hour;
public:
        Time& operator++(void);
}
Time& Time::operator++(void) {
        hour += (min += ++sec/60 ) /60;
        sec %= 60; min %= 60; hour %=24;
        return *this;
}
Time time;
++time;
time++;
time.operator++();
```

Operatori unari - detaliere

- *Operatorul++* implementat ca mai sus, are două forme de apel acceptate, ca operator prefix, şi ca operator sufix, dar, pentru tipul abstract implementat, ele sunt echivalente
- Dacă se doreşte ca operator++ forma sufix să fie diferit de forma prefix se recurge la un compromis între supraîncărcarea unui operator unar şi a unuia binar:
  - Se cunoaşte că un operator unar operează asupra unui singur operand, forma operatorului fiind

```
tip operator++();
tip operator--();
```

  - unde operandul este dat prin argumentul implicit *this*
  - Un operator binar are doi operanzi: cel implicit (curent) şi cel explicit dat ca argument în metodă:
    - `tip operator++(argument);`
  - Se implementează o operaţie binară în care al doilea operand nu este utilizat

Operatori unari - detaliere

- Forma sufix:

```
Time& Time::operator++(int i)
{
    hour++;
    hour %= 24;
    return *this;
}
int i = 1;
Time time;
time++;
time.operator++(i);
++time;
time.operator++();
```

- Concluzie:
  - Dacă ar fi lipsit definiţia variantei sufix atunci atât *time++* cât şi *++time* ar fi fost rezolvate prin varianta prefix
  - Dacă ar fi lipsit definitia variantei prefix, ar fi apărut eroare la *++time* căci varianta sufix cere un argument in lista de argumente

Operatori unari - detaliere

- Dacă *operator*++ este definit ca funcţie globală atunci are acces la partea privată a obiectului Time numai dacă este funcţie prietenă.

```
class Time
{
        int min, sec, hour;
public:
        friend Time& operator++(Time&);
}
Time& operator++(Time& t)
{
        t.hour += (t.min += ++t.sec/60 ) /60;
        t.sec %= 60;
        t.min %= 60;
        t.hour %=24;
        return t;
}
```

- Apel: `Time time; time++; ++time; operator++(t); //varianta prefix`

Operatori unari - detaliere

- Varianta pentru forma sufix

```
class Time
{
        int min, sec, hour;
public:
        friend Time& operator++(Time&);
        friend Time& operator++(time&, int);
}

Time& operator++(Time& t, int i)
{
        t.hour++;
        t.hour %= 24;
        return t;
}
```

- Pentru supraîncărcarea unui operator, in formă nemembru şi neprieten trebuie folosite funcţiile de acces la membri privaţi din clasa *Time*

Vă mulţumesc !