

Capitolul 6

Supraîncărcarea operatorilor.

Operatorii sunt tratați de către compilator ca niște funcții speciale, la care sintaxa de apel este diferită de cea a funcțiilor obișnuite, numele operatorilor și al parametrilor (operanzilor) respectând reguli asemănătoare celor din algebră.

Și limbajul C posedă operatori supraîncărcați care efectuează operații diferite. De exemplu, operatorul / reprezintă atât operația de împărțire reală în cazul în care cel puțin un operator este real, cât și operația de determinare a câtului împărțirii a două numere întregi în cazul în care ambii operanzi sunt întregi.

Limbajul C++ permite supraîncărcarea de către programatori a operatorilor limbajului. Cu alte cuvinte, se pot redefini operatorii acestui limbaj astfel încât să efectueze operații definite de programator.

Indiferent de modul în care se realizează supraîncărcarea operatorilor, aceștia *trebuie să fie asociați unor clase*, în sensul că cel puțin unul dintre parametrii trebuie să fie un obiect al unei clase (explicit sau implicit prin intermediul parametrului ascuns `this`).

Observație. Operația de supraîncărcare a operatorilor nu are nici o legătură cu derivarea claselor, deci nu generează obiecte polimorfice.

6.1 Generalități. Definirea și apelul operatorilor

Avantajul utilizării operatorilor supraîncărcați în cadrul claselor decurge din simplificarea scrierii și citirii unei aplicații, datorită formei sintactice de apel a operatorilor.

Exemplul 6.1. În cazul proiectării unei clase pentru operațiile cu polinoame, operația de adunare a două polinoame poate fi descrisă fie printr-o funcție obișnuită, fie prin supraîncărcarea operatorului de adunare:

A) Utilizarea unei funcții:

```
class polinom {
    double *a;
    int n;
public:
    polinom();
    // ...
    friend polinom suma(polinom& a, polinom& b);
    // ...
};
```

```

void prelucrare() {
    polinom p, q, r;
    // ...
    r = suma(p, q);
    // ...
}

```

B) Utilizarea unui operator prieten supraîncărcat:

```

class polinom {
    double *a;
    int n;
public:
    polinom();
    // ...
    friend polinom operator+(polinom& a, polinom& b);
    // ...
};
void prelucrare() {
    polinom p, q, r;
    // ...
    r = p + q;
    // ...
}

```

Se observă faptul că *definirea* unui operator supraîncărcat reprezintă practic definirea unei funcții, la care numele are forma predefinită:

```
operator <op>
```

unde <op> reprezintă operatorul ce se dorește supraîncărcat. *Apelul* unui operator supraîncărcat se face conform sintaxei predefinite a operatorului respectiv.

Deoarece un operator supraîncărcat este o funcție asociată cu o anumită clasă, funcția operator poate fi definită atât ca o funcție *friend* a clasei respective, cât și ca o funcție membru. În cazul funcțiilor membru, numărul de parametri este întotdeauna mai mic cu 1 decât aritatea operatorului corespunzător, deoarece un parametru este implicit în acest caz (obiectul curent al clasei, reprezentat de parametrul ascuns *this* al funcției).

Rezultă astfel o a treia variantă pentru clasa *polinom*:

C) Utilizarea unui operator supraîncărcat, membru al clasei:

```

class polinom {
    double *a;
    int n;
public:
    polinom();
    // ...
    polinom operator+(polinom& a);
    // ...
};

```

```

void prelucrare() {
    polinom p, q, r;
    // ...
    r = p + q;
    // ...
}

```

Ceea ce diferă în ultimele două cazuri la apelul operatorilor este sintaxa generată de compilator pentru apel. În cazul funcției `friend`, sintaxa efectivă de apel pentru `p+q` este:

```
operator+(p, q)
```

pe când în cazul funcției membru, sintaxa este:

```
p.operator+(q)
```

Majoritatea operatorilor limbajului C pot fi supraîncărcați, după cum se observă din tabelul următor:

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

Pentru determinarea numărului de argumente ale funcției operator, trebuie luate în considerare atât aritatea operatorului original din limbajul C, cât și dacă funcția operator este o funcție membru sau o funcție `friend`.

Tabelele următoare prezintă sintaxa de apel și forma de apel efectiv pentru operatorii unari și binari (s-au notat cu X și Y operandii și cu Op operatorul) în cazul funcțiilor membru, precum și al funcțiilor `friend`:

Sintaxa de apel	Apel efectiv
Op X	X.operatorOp()
X Op	X.operatorOp()
X Op Y	X.operatorOp(Y)

Sintaxa de apel	Apel efectiv
Op X	operatorOp(X)
X Op	operatorOp(X)
X Op Y	operatorOp(X, Y)

Utilizarea operatorilor supraîncărcați în cadrul unui program C++, trebuie să respecte însă anumite restricții impuse de tratarea uniformă de către compilator a operatorilor:

- 1) Nu se pot defini operatori noi, alții decât cei prezentați în primul tabel;
- 2) Nu se poate schimba aritatea sau precedența unui operator;
- 3) Operatorii nu pot fi combinați pentru a grupa operatori noi. De exemplu, dacă s-au definit operatorii `+` și `=`, apelul `+=` nu trebuie să însemne apelul lui `+`, urmat de apelul lui `=`.

4) Operatorii = , [] , -> și () trebuie să fie funcții membre nestatice.

Necesitatea ultimei restricții va fi prezentată într-un paragraf ulterior.

Dupa cum s-a specificat anterior, operatorii pot fi definiți atât ca funcții membru cât și ca funcții prietene unei anumite clase. Excepție fac operatorii = , () , [] , -> • i -> * , care trebuie să fie *întotdeauna* funcții membru. Pentru ceilalți operatori pot fi luate în considerare următoarele sugestii de definire a operatorilor:

- Este indicat ca toți operatorii unari să fie definiți ca funcții membru;
- Operatorii binari compusi de atribuire (+= , -= , /= , *= , ^= , &= , |= , %= , >>= , <<=) este indicat să fie definiți ca funcții membru;
- Ceilalți operatori binari este indicat să fie definiți ca funcții friend.

Operatorul () de apel de funcție prezintă un aspect mai deosebit datorită arității sale. Un asemenea operator permite ca un anumit obiect să fie utilizat ca și o funcție, căreia i se pot transmite un anumit număr de parametri. Deoarece un asemenea operator trebuie să fie o funcție membru a clasei, numărul de parametri ai operatorului este egal cu numărul de parametri ai funcției asociate. Rezultă astfel că se pot suprascrie mai mulți operatori () pentru a anumită clasă, fiecare având o semnificație distinctă.

6.2 Operatori unari

Dintre operatorii unari, singurii operatori care pot prezenta probleme sunt cei de incrementare și decrementare, datorită faptului că aceștia pot avea două forme de apel: prefixă și postfixă. Efectul lateral este același, dar valoarea de evaluare a unei expresii este diferită în cazul celor două forme de apel.

Exemple:

```
int n = 4;
if (n++ < 6) cout << "OK\n"; //Evaluare(n++)≡4
cout << n << endl; //n=5
if (++n < 6) cout << "OK\n"; //Evaluare(++n)≡6
cout << n << endl; //n=6
```

Rezultă astfel că există doi operatori diferiți pentru operatorii de incrementare și decrementare, compilatorul generând apeluri diferite pentru cele două forme ale acestora.

De exemplu, în cazul unui operator de incrementare ++ definit ca o funcție prietenă unei clase, pentru forma prefixă ++a, se va genera un apel de forma:

```
operator++(a)
```

pe când în cazul formei postfixe a++, se va genera un apel de forma:

```
operator++(a,int)
```

Parametrul auxiliar este utilizat doar pentru a face deosebire între cei doi operatori.

Diferența dintre cele două forme de utilizare ale acestor operatori constă în modul de utilizare a valorii returnate și a tipului acesteia. În mod uzual, valoarea returnată de un operator postfix este o R-valoare, pe când valoarea returnată de un operator prefix este o L-valoare, care poate fi modificată imediat în program. Deși acest lucru nu este verificat de

către compilator, este indicat ca funcțiile corespunzătoare celor două forme ale unui asemenea operator să implementeze aceste caracteristici.

Exemplul 6.2. Clasa *fractie* conține ambele forme ale operatorului ++.

```
class fractie {
    int p, q;
public:
    fractie(int a = 0, int b = 1) { p = a; q = b; }
    fractie(fractie&);
    int GetP() const { return p; }
    int GetQ() const { return q; }
    fractie& operator=(fractie&);
    fractie& operator++();
    fractie operator++(int);
    fractie operator--(int);
    // Returneaza valoarea reala a lui p /q
    double operator()() const { return (double)p/q; }
    // Returneaza catul impartirii p /q
    int operator()(int) const { return p/q; }
};

fractie::fractie(fractie& f) {
    p = f.p;
    q = f.q;
}

fractie& fractie::operator=(fractie& f) {
    if (&f == this)
        return *this;
    p = f.p;
    q = f.q;
    return *this;
}

fractie& fractie::operator++() {
    ++p;
    return *this;
}

fractie fractie::operator++(int) {
    fractie tmp(*this);
    p++;
    return tmp;
}

fractie fractie::operator--(int) {
    fractie tmp(*this);
    p--;
    return tmp;
}
```

```

    }

    void main() {
        fractie f1(5, 2);
        while ((f1++).GetP() < 10) {
            cout << f1.GetP() << endl;
            f1 = f1--;
        }
        fractie f2(5, 2);
        while ((++f2).GetP() < 10) {
            cout << f2.GetP() << endl;
            f2 = f2--;
        }
        fractie f3(5, 2);
        while ((++f3).GetP() < 10) {
            cout << f3() << " " << f3(0) << endl;
            f3 = f3--;
        }
    }
}

```

Programul precedent afișează 6, 7, 8, 9, 10 pentru primul ciclu și 6, 7, 8, 9 pentru ciclul al doilea. Se observă faptul că funcția accesoriu *GetP* se aplică în primul caz asupra unei copii a obiectului curent, înainte de incrementarea număratorului.

În plus, clasa *fractie* mai conține doi operatori supraîncărcați de apel de funcție, unul utilizat pentru determinarea valorii reale a unei fracții, iar al doilea pentru determinarea părții întregi a unei fracții. Deoarece ambele apeluri de funcții nu au în mod normal parametri, s-a introdus un parametru forțat de tip *int*. Ultimul ciclu al exemplului anterior afișează valorile:

```

3 3
3.5 3
4 4
4.5 4

```

6.3 Operatorul de atribuire

Operatorul de atribuire este unul dintre cei mai importanți și utilizați operatori supraîncărcați, fiind și singurul operator care poate fi generat implicit de către compilator în lipsa definiției explicite a acestuia într-o clasă.

Mai mult însă, operatorul de atribuire face parte din categoria operatorilor care trebuie să fie definiți ca *funcții membru*. Aceasta restricție este naturală, datorită operației de atribuire:

```
<variabila> = <expresie>
```

operatorul de atribuire fiind strâns legat de variabila ce reprezintă primul operand.

Există o anumită asemănare între operația de atribuire și cea de inițializare a unei variabile, ceea ce generează o asemănare între operatorul de atribuire și constructorul de copiere.

Exemplul 6.3. Definiția și utilizarea unei clase reprezentând numerele complexe.

```

class NumarComplex {
    double x,y;
public:
    NumarComplex (double a=0, double b=0) { x=a; y=b; }
    NumarComplex (NumarComplex& c) { x=c.x; y=c.y; }
    void operator=(NumarComplex& c) { x=c.x; y=c.y; }
    void Print() { cout << x << y << endl; }
};

void main() {
    NumarComplex z1(2, 7);
    NumarComplex z2 = z1; // constructor de copiere
    NumarComplex z3;
    z3 = z1; // operator de atribuire
    z1.Print();
    z2.Print();
    z3.Print();
}

```

Deosebirea între atribuire și inițializare este semnificativă, deoarece în cazul inițializării, în afara de inițializarea valorilor membrilor unui obiect se alocă în plus și memoria pentru obiectul respectiv.

Una dintre problemele care pot apărea se referă la tipul valorii returnate de un operator de atribuire. În cazul în care tipul rezultatului este `void`, ca în exemplul precedent, nu se poate realiza atribuire multiplă. De exemplu, pentru atribuirea:

```
z2 = z1 = z;
```

compilatorul ar trebui să genereze:

```
z2.operator=(z1.operator=(z));
```

Dar `z1.operator=(z)` are tipul `void`, pe când parametrul funcției `z2.operator=()` trebuie să fie tot o referință la un obiect de tipul *NumarComplex*.

În mod uzual operatorii de atribuire returnează un obiect instanță al clasei respective sau o referință la un asemenea obiect. De exemplu, operatorul de atribuire pentru clasa *NumarComplex* din exemplul 6.3, poate fi definit astfel:

```

NumarComplex operator=(NumarComplex& c) {
    x = c.x;
    y = c.y;
    return *this;
}

```

În cazul în care o clasă nu posedă un operator propriu de atribuire, compilatorul va genera un operator implicit, într-un mod asemănător constructorului de copiere: se va genera o asignare membru cu membru a datelor componente.

Exemplul 6.4.

```
class A {
```

```

public:
    A& operator=(const A&) {
        cout << "clasa A; operator=" << endl;
        return *this;
    }
};

class B {
    A a;
};

void main() {
    // se genereaza constructori impliciti pentru A și B
    B b1, b2;
    b1 = b2;    // operator= implicit pentru clasa B
}

```

Ieșirea programului anterior afișează mesajul "clasa A; operator=", ceea ce înseamnă că s-a generat un operator de atribuire pentru membrul *a*.

Observațiile de la constructorul de copiere în cazul claselor derivate rămân valabile și în acest caz: dacă o clasă este derivată din una sau mai multe clase de bază și conține în plus și date membru ce sunt instanțe ale altor clase, operatorul de atribuire generat implicit de compilator apelează întâi operatorii de atribuire ale claselor de bază, iar apoi pe cei ai claselor la care aparțin obiectele membre, în ordinea specificării lor în clasa derivată.

De exemplu, pentru clasa:

```

class D: public B2, B1 {
    M1 m1;
    M2 m2;
    // ...
};

```

o secvență de forma

```

class D d1, d2;
// ...
d2 = d1;

```

apelează operatorii de atribuire în următoarea ordine a claselor: B2, B1, M1, M2.

În cazul în care o instanță a unei clase componente sau de bază este la rândul ei o clasă derivată, regula de apel a operatorilor de atribuire se aplică în continuare recursiv.

Dacă într-o clasă se utilizează pointeri, este neindicat să se lase operatorul de atribuire să fie generat automat de compilator, deoarece în acest caz se va copia doar valoarea pointerului (adresa obiectului), nu și obiectul indicat de aceasta:

Exemplul 6.5. Un operator de atribuire pentru clasa *polinom* definită anterior:


```

class polinom {
    double *a;
    int n;
public:
    polinom(int k = 0) { a = new double [n=k]; }
    polinom& operator=(polinom& p);
    // ...
};

polinom& polinom::operator=(polinom& p) {
    a = new double [n=p.n];
    // Se copiaza toti coeficientii
    for (int i=0; i<=n; i++)
        a[i] = p.a[i];
    return *this;
}

```

În implementarea anterioară a operatorului de atribuire există o eroare destul de subtilă: în cazul unei atribuirii, zona de memorie ocupată de coeficienții polinomului inițial va rămâne blocată până la sfârșitul execuției programului. În mod corect va trebui testat dacă apare o asemenea situație și eventual eliberată zona de memorie aferentă coeficienților:

```

polinom& polinom::operator=(polinom& p) {
    delete[] a;
    a = new double[n=p.n];
    for (int i=0; i<=n; i++)
        a[i] = p.a[i];
    return *this;
}

```

Operatorul de atribuire anterior mai conține o eroare ascunsă. În cazul în care se efectuează o atribuire de forma:

```
p = p;
```

după operația de dealocare a memoriei pentru obiectul curent, informația acestuia nu mai este disponibilă și nu se mai poate efectua copierea datelor membre. Din acest motiv, cazul în care se încearcă atribuirea unei variabile la ea însăși, trebuie tratat separat.

Rezultă o nouă variantă pentru operatorul de atribuire al clasei *polinom*:

```

polinom& polinom::operator=(polinom& p) {
    if (&p == this)
        return *this;
    a=new double[n=p.n];
    for(int i=0; i<=n; i++)
        a[i] = p.a[i];
    return *this;
}

```

Există două restricții importante la supraîncărcarea operatorului de atribuire, care nu se aplică la majoritatea celorlalți operatori:

- funcția `operator=` trebuie să fie nestatică, pentru a se asigura că operandul stâng al atribuirii este întotdeauna un obiect;
- funcția `operator=` nu poate fi moștenită într-o ierarhie de clase (ca și constructorul de copiere), deoarece fiecare clasă derivată dintr-o ierarhie poate conține și membri specifici, iar pentru aceștia nu se poate utiliza un operator de atribuire dintr-o clasă de bază.

Exemplul 6.6.

```
class B {
    int n;
public:
    B(int k=0) { n=k; }
    B& operator=(B& b) { n=b.n; return *this; }
};

class D: public B {
    int k;
public:
    D(int a, int b):B(a), k(b) {}
    D& operator=(D& d) {
        k = d.k;           // copiaza membrul suplimentar
        B::operator=(d);   // se copiaza partea de bază
        return *this;
    }
};
```

Se observă că în exemplul precedent s-a utilizat operatorul de atribuire al clasei de bază, caruia i s-a transmis un obiect din clasa derivată, ceea ce este corect.

6.4 Operatori binari

Majoritatea operatorilor supraîncărcabili sunt binari, deoarece singurul operator ternar nu se poate supraîncărca, iar operatorii unari sunt puțini.

Posibilitatea de definire a operatorilor binari ca funcții membru sau funcții prieten diferă în funcție de tipul lor, după cum s-a precizat anterior. Operatorii `=`, `[]`, `()`, `->` și `->*` *este necesar* să fie definiți ca funcții membru, pe când restul operatorilor binari *este indicat* să fie definiți ca funcții prieten.

Avantajul utilizării operatorilor binari de tip `friend` constă în posibilitatea conversiei automate a operandilor, spre deosebire de operatorii de tip membru, când operandul din stanga operatorului trebuie să aibă tipul de date corespunzător (clasa în care s-a definit operatorul).

Exemplul 6.7.

```
class Numar
{
```

```

    int n;
public:
    Numar(int k=0): n(k) {}
    const Numar operator+(const Numar& k) const
        { return k + n.k; }
    friend const Numar operator-(const Numar&,
        const Numar&);
};

const Numar operator-(const Numar& n1, const Numar& n2)
    { return Numar (n1.n-n2.n); }

void main()
{
    Numar a(7), b(3);
    a+b; // OK
    a+1; // OK: al doilea argument este convertit la Numar
    1+a; // Eroare: primul argument trebuie să fie Numar
    a-b; // OK
    a-1; // OK: al doilea argument este convertit la Numar
    1-a; // OK: primul argument este convertit la Numar
}

```

Deoarece clasa *Numar* are un constructor de conversie de la *int* la *Numar*, la apelul unui operator de forma:

```
operator-(<arg1>, <arg2>)
```

se poate crea un obiect de tip *Numar* plecând de la o valoare întreagă, atât pentru primul argument cât și pentru al doilea. În cazul unui apel de forma:

```
<arg1>.operator-(<arg2>)
```

această conversie se poate realiza doar pentru argumentul al doilea.

Exemplul 6.8. În continuare se va prezenta un exemplu în care se definește clasa *polinom*, care conține operatori supraîncărcați:

```

class polinom {
    double *a;
    int n;
public:
    polinom(int k) { a = new double[n=k]; }
    polinom() { n=0; a=0; }
    polinom(polinom&);
    ~polinom() { delete[] a; a=0; n=0; }
    polinom& operator=(polinom &);
    friend polinom operator*(polinom&, polinom&);
    int operator<(polinom& p) { return n<p.n; }
    double& operator[](int i) { return a[i]; }
    // ...
};

polinom::polinom(polinom& p):n(p.n) {
    if (&p == this)

```

```

        return *this;
    delete[] a;
    a = new double[n=p.n];
    for (int i=0; i<=n; i++)
        a[i] = p.a[i];
    return *this;
}

polinom operator*(polinom &p1, polinom &p2) {
    polinom p(p1.n+p2.n);
    for(int k=0; k<=p.n; k++) {
        p.a[k] = 0;
        for(int i=0; i<=p1.n; i++)
            p.a[k] += p1.a[i]*p2.a[k-i];
    }
    return p;
}

```

6.5 Conversia tipurilor

Limbajul C permite două moduri de converie a tipurilor de date: *conversia explicită* a tipurilor (operatorul `cast`, de conversie explicită), precum și o *conversie implicită* a acestora.

Conversia explicită nu presupune supraîncărcarea operatorilor, ci un set de operatori predefiniți ai limbajului C++.

A. Conversia explicită a tipurilor

Limbajul C++ suportă conversia explicită a tipurilor din C, dar are în plus o serie de operatori specfici de converie explicită. Avantajul utilizării acestora constă în faptul că fiecare din ei tratează o anumită categorie de conversii de tip.

Principalii operatori de conversie explicită sunt: `static_cast`, `const_cast`, `dynamic_cast` și `reinterpret_cast`. Pentru apnelul acestora, în loc de sintaxa tradițională:

```
(<tip>) <expresie>
```

se utilizează sintaxa:

```
<operator-cast> '<' <tip> '>' <expresie>
```

unde `<operator-cast>` poate fi unul dintre operatorii menționați anterior.

Operatorul `static_cast` este principalul opetaor de conversie explicită și se utilizează în general pentru conversiile bine definite, pentru care se poate utiliza operatorul `cast` al

limbajului C. Ceilalți operatori de conversie sunt operatori specializați, care pot fi utilizați în anumite cazuri particulare.

Exemple:

```
//A. Conversii spre un tip apropiat
int a, b;
double x = static_cast<double>(a)/b;
//B. Conversii de pointeri de la tipul void*
double *p = static_cast<double*>malloc(sizeof(double));
//C. Conversii de tip implicite, care in mod normal
//sunt efectuate de compilator
void f(double z);
int k = 3;
f(static_cast<double>(k));
```

Operatorul `const_cast` este utilizat în cazul calificatorilor `const` și `volatile`. Notând cu T un tip de date, acest operator permite conversia de la un tip `const T` sau `volatile T` la tipul T sau un tip derivat din T (T^* de exemplu).

Exemple:

```
class A {
    // ...
};
class B: public A {
    // ...
};
void g(B *pb);
B b;
const B& cb = b;
// Eroare! Trebuie B*, nu const B*
g(&cb);
// OK.
g(const_cast<B*>(&cb));
// OK. Acelasi lucru, dar utilizand stilul C
g((B*)&cb);
A *pa = new A;
// Eroare! Trebuie B*, nu A*
g(pa);
// Eroare! const_cast nu poate fi utilizat pentru
// o conversie de tip downcasting
g(const_cast<B*>(pa));
```

Operatorul `dynamic_cast` este utilizat doar în cazul ierarhiilor de clase, pentru a realiza conversia de la o clasă aflată în partea de sus a unei ierarhii de clase la o clasă aflată în partea de sus a acesteia. Operația se numește în mod uzual **downcasting** și se va discuta într-un capitol destinat polimorfismului.

Observație. Operatorul se poate utiliza doar în cazul ierarhiilor de clase care utilizează polimorfismul (care posedă funcții virtuale), deoarece folosește informațiile din tabela VFTABLE.

Exemple:

```
class A {
    // Nu exista functii virtuale
    // ...
};
class B: public A {
    // ...
};
class A1 {
public:
    virtual ~A1() {}
    // ...
};
class B1: public A1 {
    // ...
};
// ...
A1* pa1 = new B1; // Upcasting
// OK. Downcasting
B1* pb1 = dynamic_cast<B1*>(pa1);
int a, b;
// Eroare! Nu exista mostenire
double x = dynamic_cast<double>(a)/b;
A* pa = new B; // Upcasting
// Eroare. Nu exista functii virtuale
B* pb = dynamic_cast<B*>(pa);
```

Operatorul `reinterpret_cast` este utilizat în cazurile în care un obiect este privit ca o structură de biți și se dorește să fie interpretat ca un obiect cu o structură complet diferită. În mod uzual rezultatul conversiei este dependent de implementare, ceea ce înseamnă că acest tip de conversie nu este portabil.

O folosire uzuală a operatorului `reinterpret_cast` constă în conversia între diferite tipuri de pointeri.

De exemplu, se consideră un tablou de pointeri la funcții:

```
typedef void (*FuncPtr)();
FuncPtr funcPtrArray[10];
```

În cazul în care se dorește ca dintr-un anumit motiv să se utilizeze un pointer la o funcție cu un alt prototip:

```
int Func();
```

se poate proceda astfel:

```
// Eroare! Tipuri diferite
funcPtrArray[0] = &Func;
// OK.
funcPtrArray[0] = reinterpret_cast<FuncPtr>(&Func);
```

Se impune o atenție sporită la utilizarea acestui tip de conversie, deoarece compilatorul este forțat să nu mai efectueze verificări de tip și acest lucru poate genera frecvent erori.

În plus, pentru o folosire corectă, se impune o reconversie spre tipul inițial a tipului convertit.

Exemplul 6.9.

```
#include <iostream>
using namespace std;
const int sz = 100;
struct X {
    int a[sz];
};
void print(X* x) {
    for(int i = 0; i < sz; i++)
        cout << x->a[i] << ' ';
    cout << endl;
}
void main() {
    X x;
    print(&x);
    int* xp = reinterpret_cast<int*>(&x);
    for(int* i = xp; i < xp + sz; i++)
        *i = 0;
    // Nu se poate folosi xp ca X*
    // daca nu se converteste inapoi
    print(reinterpret_cast<X*>(xp));
    // Identificatorul x se poate folosi fara conversie
    print(&x);
}
```

B. Conversia implicită a tipurilor

Un exemplu de *conversie automată* a tipurilor de date în cadrul limbajului C este la evaluarea expresiilor. De exemplu, în cazul definirii funcției următoare:

```
void f(double x) { cout << x << endl; }
```

dacă se apelează funcția f cu un parametru întreg, $f(5)$, compilatorul realizează o conversie automată de la tipul `int` la `double`.

În cazul limbajului C++ este posibil să se realizeze o conversie automată și pentru tipurile de date definite de utilizator. Există pentru aceasta două posibilități: folosind *constructori de conversie* sau folosind *operatori de conversie*.

B1) În cazul utilizării *constructorilor de conversie*, se poate realiza conversia automată de la tipul parametrului constructorului respectiv la tipul clasei în care este definit constructorul.

Exemplul 6.10.

```
class A {
    int n;
public:
    // constructor de conversie de la tipul int la tipul A
    A(int k=0) { n=k; }
    void Print() { cout<<n<<endl; }
};

void f(A a) {
    a.Print();
}

void main () {
    A a(10);
    f(a);    // nu se face conversie de tip
    f(5);    // se face conversie int -> A
}
```

La al doilea apel al funcției *f* se apelează constructorul de conversie al clasei *A*.

Există însă cazuri când nu se dorește să se realizeze o conversie automată a tipurilor. În aceste situații se poate împiedica conversia prin plasarea cuvântului cheie *explicit* în fața constructorului respectiv.

Exemplul 6.11.

```
class B1 {
public:
    double x;
    B1(double z = 0.0): x(z) { }
};

class B2 {
public:
    double y;
    B2(double z = 0.0): y(z) { }
    // se poate realiza conversia automata B1 -> B2
    B2(const B1& b): y(b.x) { }
    void Print() { cout << y << endl; }
};

class B3 {
```



```

public:
    double k;
    B3(double z = 0.0) { k = z; }
    // nu se permite conversia implicita B1 -> B3
    explicit B3(const B1& b): k(b.x) { }
    void Print() { cout << k << endl; }
};

void f(B3 b) {
    b.Print();
}

void g(B2 b) {
    b.Print();
}

void main () {
    B1 b1;
    B2 b2;
    B3 b3;
    // OK! Exista operator de conversie B1 -> B2
    g(b1);
    // Eroare! Nu exista converia implicita B2 -> B3
    f(b2);
    // Eroare! Nu exista permisa converia implicita B1->B3
    f(b1);
    // OK! Nu este nevoie de conversie
    f(b3);
    // OK! Conversia este apelata explicit de programator
    f(B3(b1));
}

```

B2) A doua metodă ce permite conversia automată a tipurilor utilizează un *operator de conversie*. Acesta este un operator cu o sintaxă specială ce permite conversia de la tipul clasei în care se definește operatorul la un tip de date dorit de programator. Prototipul unui asemenea operator este:

```
operator <nume_clasa> ()
```

Se observă faptul că numele operatorului este numele clasei destinație a conversiei, iar tipul de date al funcției operator lipsește, fiind înlocuit de cuvântul cheie operator.

Exemplul 6.12.

```

class C1 {
    int n;
public:
    //Constructor de conversie: int -> C1
    C1( int i = 0): n(i) { }
    void Print() const { cout << n << endl; }
};

```

```

class C2 {
    int m;
public:
    //Constructor de conversie: int -> C2
    C2(int i): m(i) { }
    //Operator de conversie: C2 -> C1
    operator C1() const { return C1(m); }
};

void h(C1 c) { c.Print(); }

void main( ) {
    C2 c(1);
    h(c);    // se aplica operatorul de conversie
    h(1);   // se aplica constructorul de conversie
}

```

Deosebirea între cele două metode constă în faptul că în cazul constructorului de conversie, operația de conversie între un tip sursă și un tip destinație se realizează de către obiectele tipului destinație, pe când în cazul operatorului de conversie obiectele tipului sursă sunt responsabile de această conversie.

În mod uzual este permis un singur tip de conversie automată între două tipuri de date. În cazul în care s-ar permite mai multe metode de conversie (și operator de conversie și constructor de conversie), poate apare confuzie la selectarea de către compilator a tipului de conversie.

Exemplul 6.13.

```

class B;

class A {
public:
    operator B() const ;    // conversie A -> B
};

class B {
public:
    B(A);    // conversie A -> B
}

void f(B) { }

void main {
    A a;
    f(a);    // Eroare!! Ambiguitate apel
}

```

O altă eroare ce poate apare se referă strict la operatorul de conversie. Într-o anumită clasă este permisă o singură conversie de tip spre o altă clasă cu ajutorul acestui operator, pentru că în caz contrar ar putea să apară confuzie la selectarea de către compilator a operatorului dorit de conversie.

Exemplul 6.14.

```
class A;
class B;

class C {
public:
    operator A() const ;    // conversie C -> A
    operator B() const ;    // conversie C -> B
};

void f(A);

void f(B);    // supraancarcarea funcției f

void main {
    C c;
    f(a);    // Eroare!!
}
```

În exemplul precedent, nu se poate determina care versiune a funcției *f* trebuie apelată.