Obiecte temporare în C++

Pentru a scrie cod eficient e necesar să se înțeleagă origiunea obiectelor temporare, costurile impuse de acestea şi felul în care ele pot fi eliminate.

Un obiect temporar este un obiect generat de către compilator şi care nu are un echivalent în codul sursă; să considerăm un prim exemplu:

```
class test{
      public:
      test(int i = 0, int j = 0) : x(i), y(y){}
      test& operator=(const test&);
      private:
      int x;
      int y;
};
//...
test t1(10);
test t2 = test(10);
test t3 = 10;
```

Doar la inițializarea lui t1 se garantează că un compilator C++ nu va genera obiecte temporare; la inițializarea lui t2 si t3, în funcție de implementarea compilatorului, acesta poate genera obiecte temporare. De exemplu:

```
//pseudo-cod C++
test t3;
test tmp;
tmp.test::test(10,1);    //ini•ializare temporar
t3.test::test(tmp);      //ini•ializare t3 utilizând c-tor de copie
tmp.test::~test();       //distrugere temporar
}
```

Se observă că, față de inițializarea lui t1, se apelează în plus un c-tor şi un d-tor; în practică, majoritatea compilatoarelor optimizează procesul de inițializare, astfel că cele trei forme prezentate sunt echivalente din punctul de vedere al eficienței.

Exemplul anterior este o particularizare a unui caz general: nepotrivirea tipurilor. Aceasta are loc la transmiterea argumentelor prin referință, ori de câte ori este aşteptat un obiect de un tip şi este furnizat un obiect de un alt tip; în astfel de cazuri, compilatorul caută o modalitate de conversie spre tipul aşteptat, ceea ce poate conduce la generarea de obiecte temporare. De exemplu:

```
test t;
t = 20;
```

Operatorul de atribuire al clasei test primeşte drept argument un obiect test şi nu un int; înainte de a semnala eroare, compilatorul caută o cale de a obține un obiect test dintr-un int. El găseşte un constructor care face acest lucru, ceea ce conduce la:

```
//pseudo-cod C++
test tmp;                      //creare temporar
tmp.test::test(20,1);          //initializare temporar
t.test::operator=(tmp);
tmp.test::~test();             //distrugere temporar
```

Această abilitate a compilatorului de a executa conversii *implicite* între tipuri este convenție de programare; atunci când nu sunt dorite, conversiile implicite sunt blocate prin utilizarea cuvântului cheie *explicit* în fața constructorilor:

```
explicit test(int i = 0, int j = 0) : x(i), y(j){}
```

Într-o astfel de situație, conversia nu poate fi decât *explicită*, adică apelată efectiv de programator în codul sursă:

```
test t;
t = test(20);
```

Evident, în acest caz, generarea unui obiect temporar poate fi împiedicată şi printr-o supraîncărcare a operatorului de atribuire, astfel încât acesta să accepte şi argumente de tip int:

```
class test{
      public:
      //...
      test& operator=(const test&);
      test& operator=(int);
};
```

Mai trebuie specificat faptul că obiectele temporare sunt const (nemodificabile); din acest motiv, temporarele pot fi generate doar dacă argumente sunt transmise prin referințe const!

O altă cale de generare a obiectelor temporare este returnarea rezultatului prin valoare; să considerăm următorul exemplu:

```
class test
{
      friend test operator+(const test&, const test&);
      public:
      test(int i=0, int j=0) : x(i), y(i){
            cout << "test(int, int)\n";
      }

      test(const test& t) : x(t.x), y(t.y){
            cout << "test(const test&)\n";
      }

      test& operator=(const test& t){
            x = t.x;
            y = t.y;
            cout << "operator=()\n";
      }

      ~test(){
            cout << "~test()\n";
      }
      private:
      int x;
      int y;
};

test operator+ (const test& t1, const test& t2){
      test ret_val;
      ret_val.x = t1.x + t2.x;
      ret_val.y = t1.y + t2.y;
      cout << "operator+()\n";
      return ret_val;
}
int main(){
      test t1, t2, t3;        //ini•ializare
      t3 = t1 + t2;           //atribuire
      return 0;
}
```

Standardul C++ permite unui compilator să omită crearea temporarelor care sunt utilizate doar pentru a iniţializa un alt obiect de acelaşi tip. Pentru g++, opţiunea *fno-elide-constructors* dezactivează această optimizare şi forţează compilatorul să apeleze constructorii de copie corespunzători. Compilând acest cod sursă cu g++ şi cu *fno-elide-constructors*, şi executând programul, obţinem următorul output:

```
test(int, int)                  //ini•ializare t1
test(int, int)                  //ini•ializare t2
test(int, int)                  //ini•ializare t3
test(int, int)                  //ini•ializare ret_val
operator+()
test(const test&)               //ini•ializare temporar
~test()                         //distrugere ret_val
operator=()                     //copiere temporar in t3
~test()                         //distrugere temporar
~test()                         //distrugere t3
~test()                         //distrugere t2
~test()                         //distrugere t1
```

Valoarea returnată de *operator+()* este de tip X; un obiect temporar este generat pentru a păstra acestă valoare. Temporarul este iniţializat de constructorul de copie, care are drept argument pe ret_val. Operatorul de atribuire va copia apoi valoarea temporarului în t3.

De ce este generat obiectul temporar? Pentru că nu avem libertatea de a şterge vechiul conţinut al lui t3 înainte de a-l scrie pe cel nou; singurul care poate face acest lucru este operatorul de atribuire. Deoarece compilatorul nu are permisiunea de a "sări" peste operatorul de atribuire, utilizarea unui temporar este o condiţie necesară.

Compilând fără *fno-elide-constructors*, compilatorul are libertatea de a efectua optimizări şi de a elimina constructorul de copie, ceea ce conduce la următorul output:

```
test(int, int)
test(int, int)
test(int, int)
test(int, int)
operator+()
operator=()
~test()
~test()
~test()
~test()
```

Efectuând modificarea:

```
int main(){
      test t1, t2;              //ini•ializare
      test t3 = t1 + t2;        //ini•ializare
      return 0;
}
```

obţinem, cu *fno-elide-constructors,* output-ul:

```
test(int, int)                  //ini•ializare t1
test(int, int)                  //ini•ializare t2
test(int, int)                  //ini•ializare ret_val
operator+()
test(const test&)               //ini•ializare temporar
~test()                         //distrugere ret_val
test(const test&)               //copiere temporar in t3
~test()                         //distrugere temporar
~test()
~test()
~test()
```

iar fără *fno-elide-constructors* output-ul:

```
test(int, int)                  // ini•ializare t1
test(int, int)                  // ini•ializare t2
test(int, int)                  // ini•ializare ret_val
operator+()
~test()
~test()
~test()
```

De ce sunt aceste output-uri diferite față de primele două? Pentru că, spre deosebire de prima formă a programului, când t3 este deja inițializat, în acestă a doua formă t3 este un obiect nou, care trebuie inițializat; inițializarea se realizează prin intermediul unui obiect de acelaşi tip, ceea ce conduce la utilizarea constructorului de copie. Libertatea de a optimiza permite compilatorului să nu mai genereze constructorii de copie, ceea ce reduce, în final, numărul apelurilor de funcție la 7!

Cum poate un compilator să nu mai genereze constructori de copie? În mod uzual se utilizează o tehnică numită *return value optimization*, care constă în eliminarea intermediarilor şi scrierea rezultatului direct în locul dorit. De exemplu, instrcțiunea

```
t3 = t1 + t2
```

poate fi transformată de compilator în următoarea secvență:

```
struct test tmp;                //alocarea unei zone de memorie
                                //f•r• apel constructor
test_aduna(tmp, t1, t2);        //transmiterea argumentelor prin referin••
t3 = tmp;                       //scrierea rezultatului
```

Fără optimizări, test_adună() arată astfel:

```
void test_aduna(test& tmp, const test& t1, const test& t2){
      struct test ret_val;          //alocare memorie
      ret_val.test::test();         //ini•ializare
      ret_val.x = t1.x + t2.x;
      ret_val.y = t1.y + t2.y;
      tmp.test::test(ret_val);      //scrierea rezultatului prin c-tor copie
      ret_val.test::~test();
return;
}
```

Compilatorul poate aduce însă pe test_adună() la următoarea formă:

```
void test_aduna(test& tmp, const test& t1, const test& t2){
      tmp.x = t1.x + t2.x;
      tmp.y = t1.y+ t2.y;
return;
}
```

Concluzii:

- Obiectele temporare pot fi generate la transmiterea argumentelor prin referințe const sau la returnarea rezultatului prin valoare.
- Costul impus de un obiect temporar înseamnă un apel de constructor şi un apel de destructor.
- Temporarele generate în urma nepotrivirilor de tip pot fi evitate prin supraîncărcarea corespunzătoare a funcțiilor sau blocate prin declararea constructorilor cu *explicit*.
- Return Value Optimization este o tehnică ce depinde de compilator şi elimină nevoia creării şi distrugerii unui obiect local.

Bibliografie:

- ISO/IEC 14882 – Programming Languages – C++
- Scott Meyers – Effective C++
- Scott Meyers – More Effective C++
- Dov Bulka, David Mayhew - Efficient C++ Performance Programming Techniques