12. Standard C++

Obiective

- Înţelegerea modului în care se foosesc operatorii `static_cast`, `const_cast` şi `reinterpret_cast`
- Înţelegerea şi folosirea *Run-time type information* (RTTI) şi a operatorilor `typeid` şi `dynamic_cast`
- Înţelegerea noţiunii de constructor `explicit`
- Folosirea membrilor `mutable` în obiecte `const`.

12.1 Introducere

Conversia unui tip de dată în alt tip de dată se numeşte *type-cast*. Am studiat două tipuri de conversie: conversia implicită şi conversia explicită.

Pentru *conversia implicită* nu este nevoie de un operator de cast şi se face automat la copierea unei valori într-o locaţie de memorie care are un tip compatibil.

Exemplu

```
short a=2000;
int b;
b=a;
```

În acest exemplu, valoarea `short` este convertită la `int` fără a folosi vreun operator special. Este vorba, aici, despre o conversie standard care se aplică tipurilor fundamentale de dată. Uneori, conversiile implicite conduc la pierderi de precizie a valorilor, iar compilatorul semnalează acest lucru. Mesajele compilatorului pot fi evitate prin conversia explicită.

Conversia implicită se face şi prin apelul operatorilor de conversie ai claselor care sunt constructorii cu un singur parametru.

Exemplu

```
class A {};
class B { public: B (A a) {} };

A a;
B b=a;
```

Limbajul C++ este unul puternic tipizat, iar conversiile care necesită o interpretare diferită a unei valori trebuie realizate prin *conversie explicită* care se poate specifica în două modalităţi.

Exemplu

```
short a=2000;
int b;
b = (int) a;
b = int (a);
```

Aceşti doi operatori pot fi folosiţi pentru orice tip de conversie, putând fi aplicaţi şi pentru clase şi pointeri la clase. Uneori, însă, deşi codul este corect din punct de vedere sintactic, pot apărea erori la rulare sau rezultate aleatoare.

Exemplu

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

class A {
    float i,j;
};

class B {
     int x,y;
  public:
     B (int a, int b) { x=a; y=b; }
     int result() { return x+y;}
};

int main () {
  A d;
  B * padd;
  padd = (B*) &d;
  cout << padd->result();
  return 0;
}
```

Acest program declară un pointer la clasa B, însă îi asignează o referinţă la un obiect incompatibil cu el folosind un cast explicit:

```
padd = (B*) &d;
```

Cast-ul din C permite conversia unui pointer într-un pointer de alt tip, iar rularea acelei secvenţe de cod poate produce o eroare sau rezultate ale unor operaţii cu valori aleatoare din memorie.

În acest capitol vom prezenta câteva elemente introduse de limbajul C++

Standard ANSI/ISO.

12.2 Operatorul `static_cast`

Limbajul C++ standard conţine patru operatori de cast care sunt de preferat celor folosiţi de versiunile mai vechi ale limbajelor C şi C++: `static_cast`, `const_cast`, `reinterpret_cast` şi `dynamic_cast`. Noii operatori de conversie explicită sunt mai puţin puternici, neavând gradul de generalitate al celor vechi. Acesta este un avantaj pentru că dau programatorului un control mai precis asupra operaţiilor de cast care se ştie că sunt adeseori o sursă de erori în timpul rulării unui program. Un alt avantaj al celor patru operatori de cast este că fiecare se foloseşte în contexte bine definite, spre deosebire de vechii operatori de cast a căror filosofie era aceea de a fi universali.

Limbajul C++ introduce operatorul `static_cast` pentru conversia între tipuri de date, iar verificarea tipurilor este făcută la compilare. Operatorul `static_cast` permite conversiile standard, de exemplu `void*` la `char*` sau `int` la `double`, şi inversele lor.

Programul de mai jos demonstrează folosirea operatorului `static_cast`.

Exemplu

test_static_cast.cpp

```
#include <iostream>

using std::cout;
using std::endl;

class BaseClass
{
  public:
  void f() const { cout << "BASE\n"; }
};

class DerivedClass : public BaseClass
{
  public:
    void f() const { cout << "DERIVED\n"; }
};

void test(BaseClass*);

int main()
```

```
{
  //se foloseste static_cast pentru conversie
  double d = 8.22;
  int x = static_cast<int>(d);

  cout << "d este " << d << "\nx este " << x << endl;

  BaseClass *basePtr = new DerivedClass;
  test(basePtr); //apelul functiei test
  delete basePtr;

  return 0;
}

void test(BaseClass *basePtr)
{
  DerivedClass *derivedPtr;
  //cast de la pointer la clasa de baza
  //la pointer la clasa derivata
  derivedPtr = static_cast<DerivedClass*>(basePtr);
  derivedPtr->f(); //functia f din DerivedClass
}
```

Rulând acest program, obţinem următorul rezultat:

```
d este 8.22
x este 8
DERIVED
```

Acest program foloseşte clasele `BaseClass` şi `DerivedClass`, fiecare dintre ele definind funcţia `f`.

Folosim operatorul `static_cast` pentru a converti variabila `d` de tip `double` la `int`:

```
double d = 8.22;
int x = static_cast<int>(d);
```

Pointerului `basePtr` către clasa `BaseClass` îi este asignat obiectului creat prin instrucţiunea `new DerivedClass`. Acest pointer este apoi transmis ca parametru funcţiei test. În această funcţie, el este convertit la un pointer la `DerivedClass` prin instrucţiunea

```
derivedPtr = static_cast<DerivedClass*>(basePtr);
```

Operaţia de mai sus se numeşte *downcast* de la `BaseClass*` la `DerivedClass*`. *Downcasting*-ul de la un pointer la clasa de bază la un pointer la o clasă derivată este considerată nesigură, programatorul trebuind să se asigure că obiectele sunt compatibile între ele iar dereferenţierea are sens.

12.3 Operatorul `const_cast`

C++ oferă posilibilitatea de a folosi operatorul `const_cast` pentru a permite modificarea unor date `const` sau `volatile`. Programul următor arată cum se poate folosi `const_cast` pentru a schimba valoarea unei date membre a unei clase printr-o funcţie membră care este `const` şi care, în mod obişnuit, nu ar putea modifica obiectul.

Exemplu

test_const_cast.cpp

```
#include<iostream>
using std::cout;
using std::endl;

class ConstCastTest
{
  public:
    void setNumber(int);
    int getNumber() const;
    void printNumber() const;
  private:
    int number;
};

void ConstCastTest::setNumber(int num)
  { number =  num; }

int ConstCastTest::getNumber() const
  { return number; }

void ConstCastTest::printNumber() const
{
  cout << "\nNumarul dupa modificare: ";

  //expresia number-- ar genera eroare la compilare
  //const_cast face modificarea posibila
```

```
  const_cast<ConstCastTest*>(this)->number--;

  cout << number << endl;
}

int main()
{
  ConstCastTest x;
  x.setNumber(8);

  cout << "Valoarea initiala a numarului: " << x.getNumber();

  x.printNumber();

  return 0;
}
```

Rulând acest program, obţinem următorul rezultat:

```
Valoarea initiala a numarului: 8
Numarul dupa modificare: 7
```

Clasa `ConstCastTest` conţine trei funcţii membre şi variabila `number` privată. Două dintre funcţiile membre sunt declarate `const`. Funcţia `setNumber` setează valoarea datei membre `number`, iar funcţia `getNumber` returnează valoarea lui `number`.

Funcţia membră `printNumber` este constantă, dar modifică valoarea lui number prin aplicarea operatorului `const_cast`:

```
const_cast<ConstCastTest*>(this)->number--;
```

În funcţia membră `printNumber`, tipul de dată al lui `this` este `const ConstCastTest*`. Instrucţiunea de mai sus anulează temporar caracterul `const` al lui `this` prin operatorul `const_cast`. Tipul pointerului este acum `ConstCastTest*`. Se permite, astfel, modificarea valorii lui `number`. Operatorul `const_cast` nu poate fi folosit direct pentru a afecta caracterul `const` al unei variabile constante.

Operatorul `const_cast` poate, de asemenea, să fie folosit la transmiterea unui parametru `const` unei funcţii al cărei parametru este non-`const`.

Exemplu

test_const_cast2.cpp

```
#include <iostream>
```

```
using namespace std;

void print (char * str)
{
  cout << str << endl;
}

int main () {
  const char * c = "Sir constant";
  print ( const_cast<char *> (c) );
  return 0;
}
```

Rulând acest program, obţinem următorul rezultat:

```
Sir constant
```

12.4 Operatorul `reinterpret_cast`

Conversiile nestandard pot fi implementate în C++ prin operatorul `reinterpret_cast`. El poate fi folosit, de exemplu, pentru conversia de la un pointer de un tip la un pointer de alt tip, însă nu poate fi folosit pentru conversiile standard, de exemplu de la `double` la `int`.

Exemplu

test_const_cast2.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include <queue>

int main()
{
  int x = 85, *ptr = &x;
  cout << *reinterpret_cast<char*>(ptr) << endl;

  return 0;
}
```

Rulând acest program, obţinem următorul rezultat:

```
U
```

Programul declară un întreg şi un pointer. Pointerul `ptr` este iniţializat cu

adresa lui x. Instrucţiunea

```
cout << *reinterpret_cast<char*>(ptr) << endl;
```

foloseşte operatorul `reinterpret_cast` pentru a îl converti pe `ptr` de la tipul `int*` la tipul `char*`. Adresa returnată este dereferenţiată. Valoarea 85 este, astfel, interpretată ca o valoare de tip `char`, iar rezultatul este caracterul `U` al cărui cod ASCII este 85.

Comportamentul operatorului `reinterpret_cast` este dependent de platforma pe care rulează aplicaţia, putând duce la comportamente diferite.

12.5 *Run-Time Type Information* (RTTI)

Run-time tipe information (*RTTI*) oferă o modalitate de determinare a tipului unui obiect în timpul rulării programului. Vom prezenta doi operatori importanţi: `typeid` şi `dynamic_cast`.

Programul de mai jos demonstrează modul în care se poate folosi operatorul `typeid`.

Exemplu

test_typeid.cpp

```
#include <iostream>
using std::cout;
using std::endl;

#include <typeinfo>

template <class T>
T maximum(T value1, T value2, T value3)
{
  T max = value1;
  if(value2 > value1)
    max = value2;

  if(value3 > max)
    max = value3;

  const char* dataType = typeid(T).name();

  cout << "Au fost comparate date de tip " << dataType
       << "\nCel mai mare " << dataType << " este ";

  return max;
```

```
}

int main()
{
  int a = 8, b = 88, c = 22;
  double d = 95.96, e = 78.59, f = 83.99;

  cout << maximum(a, b, c) << "\n";
  cout << maximum(d, e, f) << endl;

  return 0;
}
```

Rulând acest program, obţinem următorul rezultat:

```
Au fost comparate date de tip int
Cel mai mare int este 88
Au fost comparate date de tip double
Cel mai mare double este 95.96
```

Atunci când într-un program se foloseşte rezultatul lui `typeid`, trebuie inclus fişierul header `<typeinfo>`. Programul defineşte funcţia template maximum cu trei parametri care au tipul generic de dată `T`. Funcţia determină care dintre cele trei valori este cea mai mare.

Instrucţiunea

```
const char* dataType = typeid(T).name();
```

foloseşte funcţia `name` pentru a returna numele tipului de dată reprezentat de `T` la un moment dat. Operatorul `typeid` returnează o referinţă la un obiect de tip `type_info`, el reprezentând un tip de dată. Nu este recomandat să se folosească rezultatul lui `typeid` în instrucţiuni `switch`. Corect este ca această logică să fie implementată prin funcţii virtuale.

Operatorul `dynamic_cast` se foloseşte pentru implementarea conversiilor care au loc în timpul rulării programului şi pe care compilatorul nu le poate verifica. Acest operator este folosit, de regulă, pentru *downcasting* de la un pointer la clasa de bază către un pointer la clasa derivată. El poate fi folosit doar pentru pointeri sau referinţe la obiecte. RTTI este proiectat să fie folosit în ierarhii de moştenire care implementează comportamente polimorfice. Programul următor ilustrează folosirea operatorului `dynamic_cast`.

Exemplu

test_dynamic_cast.cpp

```
#include <iostream>
using std::cout;
using std::endl;

const double PI = 3.14159;

class Shape
{
  public:
    virtual double area() const
      { return 0.0; }
};

class Circle : public Shape
{
  public:
    Circle(int r = 1)
      { radius = r; }
    virtual double area() const
      { return PI * radius * radius; }
  protected:
    int radius;
};

class Cylinder : public Circle
{
  public:
    Cylinder(int h = 1)
      { height = h; }
    virtual double area() const
    {
      return 2 * PI * radius * height +
             2 * Circle::area();
    }
  private:
    int height;
};
```

```
void outputShapeArea(const Shape*);

int main()
{
  Circle circle;
  Cylinder cylinder;
  Shape *ptr = 0;

  outputShapeArea(&circle);  //aria cercului
  outputShapeArea(&cylinder);//aria cilindrului
  outputShapeArea(ptr);      //aria unui Shape
  return 0;
}

void outputShapeArea(const Shape* shapePtr)
{
  const Circle *circlePtr;
  const Cylinder *cylinderPtr;

  //cast Shape* la Cylinder*
  cylinderPtr = dynamic_cast<const Cylinder*>(shapePtr);
  if(cylinderPtr != 0)
    cout << "Aria cilindrului: " << shapePtr->area();
  else
  {
    circlePtr = dynamic_cast<const Circle*>(shapePtr);
    if(circlePtr != 0)
      cout << "Aria cercului: " << circlePtr->area();
    else
      cout << "Nu este nici Circle, nici Cylinder.";
  }
  cout << endl;
}
```

Rulând acest program, obţinem următorul rezultat:

```
Aria cercului: 3.14159
Aria cilindrului: 12.5664
Nu este nici Circle, nici Cylinder.
```

Programul defineşte o clasă de bază numită `Shape` care conţine funcţia virtuală area, o clasă derivată numită `Circle` care moşteneşte `public` clasa `Shape` şi o clasă derivată numită `Cylinder` care moşteneşte `public` clasa `Circle`. Atât `Circle` cât şi `Cylinder` suprascriu funcţia virtuală `area`.

În funcţia `main` sunt instanţiate obiectele `circle` şi `cylinder`. Tot în funcţia main este declarat un pointer la clasa `Shape` numit ptr care este iniţializat cu valoarea `0`.

Funcţia `outputShapeArea` calculează aria obiectelor de tip `Circle` şi `Cylinder` şi are un parametru de tip pointer constant la clasa `Shape`. Cele trei obiecte declarate în funcţia `main` sunt folosite pentru trei apeluri ale acestei funcţii.

Instrucţiunea

```
cylinderPtr = dynamic_cast<const Cylinder*>(shapePtr);
```

face un *downcast* dinamic al lui `shapePtr` de tip `const Shape*` către `const Cylinder*` prin operatorul `dynamic_cast`. Ca rezultat, `cylinderPtr` primeşte adresa lui `shapePtr` dacă acesta este de tip `const Cylinder*` sau `0` în caz contrar. Când adresa este diferită de `0`, funcţia `outputShapeArea` afişează aria cilindrului.

În mod asemănător, instrucţiunea

```
circlePtr = dynamic_cast<const Circle*>(shapePtr);
```

asignează lui `circlePtr` adresa lui `shapePtr` dacă este de tip `const Circle*` sau `0` în caz contrar. Când adresa este diferită de `0`, funcţia `outputShapeArea` afişează aria cercului.

12.6 Constructori `explicit`

În capitolul „Supraîncărcarea operatorilor” am arătat că orice constructor cu un singur argument poate fi folosit ca operator implicit de conversie de la tipul de dată al argumentului la tipul de dată al clasei. Conversia este automată şi programatorul nu trebuie să folosească operaţiile de cast. În unele situaţii, conversia implicită conduce, însă, la erori. Programul de mai jos conţine o implementare a clasei `Array` în care constructorul trebuie să creeze obiecte de tip tablou unidimensional al căror număr de elemente este dat de valoarea argumentului.

Exemplu

array2.h

```
#ifndef ARRAY2_H
#define ARRAY2_H

#include <iostream>
using std::ostream;
```

```
class Array
{
  friend ostream &operator<<(ostream&, const Array&);
  public:
    Array(int = 10);
    ~Array();
  private:
    int size;
    int *ptr;
};
#endif
```

array2.cpp

```
#include <iostream>

using std::cout;
using std::ostream;

#include <cassert>
#include "array2.h"

Array::Array(int arraySize)
{
  size = (arraySize > 0 ? arraySize : 0);
  cout << "Constructorul clasei Array apelat pentru "
       << size << " elemente\n";
  ptr = new int[size];
  assert(ptr != 0);
  for(int i = 0; i < size; i++)
    ptr[i] = 0;
}

Array::~Array()
  { delete [] ptr; }

ostream &operator<<(ostream &output, const Array&a)
{
  int i;
  for(i = 0; i < a.size; i++)
```

```
    output << a.ptr[i] << ' ';
  return output;
}
```

test_array2.cpp

```
#include <iostream>
using std::cout;

#include "array2.h"

void outputArray(const Array&);

int main()
{
  Array integers1(7);
  outputArray(integers1);
  outputArray(15);

  return 0;
}

void outputArray(const Array &arrayToOutput)
{
  cout << "Tabloul primit contine:\n"
       << arrayToOutput << "\n\n";
}
```

Rulând acest program, obţinem următorul rezultat:

```
Constructorul clasei Array apelat pentru 7 elemente
Tabloul primit contine:
0 0 0 0 0 0 0

Constructorul clasei Array apelat pentru 15 elemente
Tabloul primit contine:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Clasa `Array` defineşte un constructor cu un parametru de tip `int` şi poate fi folosit ca şi operator de conversie de la tipul `int` la tipul `Array`.

Funcţia `outputArray` primeşte un parametru de tip `const Array &arrayToOutput` şi poate fi apelat cu parametri de tip `Array`, dar şi cu parametri

de tip întreg care sunt convertiţi la `Array` de constructorul de conversie.

Prin instrucţiunile

```
Array integers1(7);
outputArray(integers1);
```

se declară un obiect de tip `Array` numit `integers1` care va reprezenta un tablou unidimensional de 7 elemente de tip intreg. Funcţia `outputArray` primeşte ca parametru obiectul `integers1` şi afişează conţinutul acestuia.

Prin apelul

```
outputArray(15);
```

este invocată funcţia `outputArray` cu parametru `15` de tip `int` care va fi convertit automat la `Array`, deoarece nu există o funcţie `outputArray` care să accepte parametru întreg. Funcţia va afişa conţinutul unui tablou unidimensional cu 15 elemente de tip `int`, cu toate că parametru funcţiei nu este un tablou. Acesta este o situaţie în care comportamentul constructorului cu un parametru ca şi operator de conversie conduce la erori de logică a programului.

Limbajul C++ dispune de cuvântul cheie `explicit` care, plasat înaintea prototipului constructorului din definiţia clasei suprimă comportamentul acestuia ca operator implicit de conversie. Pentru programul de mai sus, clasa `Array` ar trebui modificată astfel:

Exemplu

array3.h

```
#ifndef ARRAY3_H
#define ARRAY3_H

#include <iostream>
using std::ostream;

class Array
{
  friend ostream &operator<<(ostream&, const Array&);
  public:
    explicit Array(int = 10);
    ~Array();
  private:
    int size;
    int *ptr;
};
#endif
```

Odată făcută această modificare, instrucţiunea

```
outputArray(15);
```

nu va mai fi compilată, generând o eroare. Ea ar trebui modificată astfel:

```
outputArray(Array(15));
```

12.7 Membri `mutable` ai unei clase

Specificatorul `mutable` este, în C++, o alternativă la operatorul `const_cast` prin care se poate modifica valoarea unei date membre printr-o funcţie `const`. O dată membră `mutable` poate fi întotdeauna modificată chiar şi prin funcţii membre `const` sau obiecte `const`.

Atât `mutable` cât şi `const_cast` permit modificarea unei date membre, însă sunt folosite în contexte diferite. Pentru un obiect `const` cu nicio dată membră `mutable`, operatorul `const_cast` trebuie folosit de fiecare dată când trebuie modificată o dată membră. Acest mecanism reduce semnificativ posibilitatea de a modifica accidental o dată membră pentru că membrul nu este permanent modificabil. Operaţiile care implică operatorul `const_cast` sunt, de regulă, ascunse în implementările funcţiilor membre.

Programul de mai jos demonstrează modul în care poate fi folosit specificatorul `mutable`.

Exemplu

test_mutable.cpp

```
#include <iostream>
using std::cout;
using std::endl;

class TestMutable
{
  public:
    TestMutable(int v = 0) {value = v;}
    void modifyValue() const {value++;}
    int getValue() const {return value;}
  private:
    mutable int value;
};

int main()
{
  const TestMutable t(99);
```

```
    cout << "Valoarea initiala: " << t.getValue();
    t.modifyValue(); //modifica data membra mutable
    cout << "\nValoarea modificata: " << t.getValue() << endl;

return 0;
}
```

Rulând acest program, obţinem următorul rezultat:

```
Valoarea initiala: 99
Valoarea modificata: 100
```

Programul defineşte clasa `TestMutable` care conţine un constructor, două funcţii membre `const` şi o dată membră `mutable`. Prin instrucţiunile

```
void modifyValue() const {value++;}
```

se defineşte `modifyValue` ca funcţie `const` care incrementează data membră `mutable int value`. În mod normal, o funcţie membră const nu poate modifica valoare unei date membre decât dacă obiectul asupra căruia operează funcţia este afectat de o operaţie de tip `const_cast`. Pentru că `value` este `mutable`, funcţia `const` poate modifica data fără a mai aplica un `const_cast`. Valoarea iniţială a datei membre value este `99`, iar după incrementare ea devine `100`.