Laborator 8

~casts și templete~

Type-casting = conversia variabileleor dintr-un tip de date în altul.

Unul dintre scopurile principale ale *typecasting-ului* este să evităm pierderea de date atunci când se evaluează o expresie. Atunci când o expresie folosește mai multe tipuri de date, dacă transformăm un tip de date cu prioritate mai mare într-unulcu prioritate mai mic, putem pierde informații. De aceea, compilatorul face automat conversia dintr-un tip de date „mai mic" într-unul „mai mare", pentru a păstra toate datele corect.

Tipuri de date de bază

C++ oferă mai multe tipuri de date predefinite:

- *Tipuri întregi:* int, short, long, long long
- *Tipuri în virgulă mobilă:* float, double, long double
- *Tipuri de caractere:* char, wchar_t
- *Tip boolean:* bool

C++ urmează o ierarhie a tipurilor de date, care stabilește cum interacționează tipurile între ele într-o expresie. În general, ierarhia este:

bool → char → short int → int → unsigned int → long → unsigned long → long long → float → double → long double

Conversie de tip implicită

Compilatorul convertește automat variabile conform ierarhiei de mai sus. Conversia implicită are loc atunci când:

- atribuim o valoare de un tip unei variabile de alt tip,
- în expresii care combină tipuri diferite (ex: int + float),
- când se **transmit argumente către funcții**, iar tipurile nu se potrivesc exact.

În C++, există câteva reguli de bază pentru **promovarea tipurilor** (type promotion):

- tipurile întregi mai mici (bool, char, short) sunt promovate automat la int când apar în expresii,

- în expresii aritmetice cu tipuri mixte, toți operanzii sunt convertiți la tipul cu cea mai mare prioritate,
- când se atribuie o valoare unei variabile de alt tip, valoarea este convertită (dacă se poate) la tipul variabilei.

```
#include<iostream>

using namespace std;
int main()
{
   int var1;
   //long var1;
   long double var2=3.2;
   int x=7;
   int y=8;

   //implicit converion of bool->int
   var1=(x>y); //in this the bool is converted into integer value
   cout<<"The value of the var1 is :"<<var1<<endl;

   //implicit converion of float->int
   float var3=3.14;
   int var4=var3;   // example of data loss
   cout<<"The value of the var3 is "<<var4<<endl;//while
converting the data is loss

   //implicit converion char to int
   char ch='A';
   int var5=10+ch;
   cout<<"The value of the var5 is: "<<var5<<endl; //the ascii
value of A is 65, it converted
                                                   // into int
value by implicit conversion


   return 0;
}
```

Conversie de tip explicită
Aici, conversia este făcută explicit de programator utilizând cuvântul-cheie **cast**.

C-style cast:

```
(new_type) expression
```

Function-style Cast:

```
new_type(expression)
```

Operatori de cast:

- static_cast

```
static_cast<new_type>(expression)
```

static_cast este folosit pentru toate conversiile care sunt bine definite. Acestea includ conversii „sigure”, pe care compilatorul le-ar permite și fără cast, precum și conversii mai puțin sigure, dar care sunt totuși permise și definite corect de limbaj.

Tipurile de conversii acoperite de static_cast includ:

- Conversii obișnuite, care nu au nevoie de cast
- Conversii de tip narrowing (care pot duce la pierderi de informație)
- Forțarea unei conversii din void* într-un alt tip de pointer
- Conversii implicite între tipuri
- Navigarea ierarhiei de clase în mod static (fără verificări la runtime)

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Shape { public: virtual ~Shape() {}; };
class Circle : public Shape {};
class Square : public Shape {};
class Other {};
int main() {
```

```
   Circle c;
   Shape* s = &c; // Upcast: normal and OK
   // More explicit but unnecessary:
   s = static_cast<Shape*>(&c);
   // (Since upcasting is such a safe and common
   // operation, the cast becomes cluttering)
   Circle* cp = 0;
   Square* sp = 0;
   // Static Navigation of class hierarchies
   // requires extra type information:
   if(typeid(s) == typeid(cp)) // C++ RTTI
       cp = static_cast<Circle*>(s);
   if(typeid(s) == typeid(sp))
       sp = static_cast<Square*>(s);
   if(cp != 0)
       cout << "It's a circle!" << endl;
   if(sp != 0)
       cout << "It's a square!" << endl;
   // Static navigation is ONLY an efficiency hack;
   // dynamic_cast is always safer. However:
   // Other* op = static_cast<Other*>(s);
   // Conveniently gives an error message, while
   Other* op2 = (Other*)s;
   // does not
} ///:~
```

- dynamic_cast

Această conversie are loc la runtime. Verifică dacă tipul în care vrem să facem conversia este valid. Dacă conversia eșuează, va apărea o eroare care indică faptul că nu s-a putut realiza conversia. Este folosită în mod special pentru clase polimorfe, adică pentru clase care au cel puțin o funcție virtuală. Operatorul dynamic_cast ne permite să verificăm dacă un pointer sau o referință către o clasă de bază (base) poate fi convertit în siguranță într-un pointer sau referință către o clasă derivată (derived).

```
#include <iostream>
using namespace std;
class Pet { public: virtual ~Pet(){}};
```

```
class Dog : public Pet {};
class Cat : public Pet {};
int main() {
   Pet* b = new Cat; // Upcast
   // Try to cast it to Dog*:
   Dog* d1 = dynamic_cast<Dog*>(b);
   // Try to cast it to Cat*:
   Cat* d2 = dynamic_cast<Cat*>(b);
   cout << "d1 = " << d1 << endl;
   cout << "d2 = " << d2 << endl;
} ///:
```

- const_cast

  const_cast este un tip de cast care ne permite sa transform un pointer constant (adică pointer care nu are voie să modifice datele către care pointează) în pointer care nu este constant.

```
int main() {
   const int i = 0;
   int* j = (int*)&i; // Deprecated form
   j = const_cast<int*>(&i); // Preferred
   // Can't do simultaneous additional casting:
   //! long* l = const_cast<long*>(&i); // Error
}
```

  Atunci când folosim const_cast putem modifica valorile către care se pointează, dar trebuie să ne asigurăm că nu modificăm obiecte care au fost declarate de la bun început cu const. Dacă încercăm să modificăm obiecte care de la bun început au fost declarate ca fiind const,atunci vom avea comportament nedefinit (*undefined behaviour*), adică nu știm cum se va comporta programul.

- reinterpret_cast

reinterpret_cast este cel mai puțin sigur dintre toate tipurile de cast-uri și cel mai predispus la erori. reinterpret_cast presupune că un obiect nu este

altceva decât o succesiune de biți care poate fi tratată — dintr-un motiv "obscur" — ca și cum ar fi un obiect complet diferit ca tip.

Este un exemplu clasic low-level bit twiddling, specifică limbajului C. În aproape toate cazurile, vom folosi un reinterpret_cast înapoi la tipul original (sau pentru a trata variabila ca fiind de tipul ei original), înainte să fie folosită în alt mod.

```
#include <iostream>
using namespace std;
const int sz = 10;
struct X { int a[sz]; };
void print(X* x) {
   for(int i = 0; i < sz; i++)
       cout << x->a[i] << ' ';
   cout << endl << "--------------------" << endl;
}
int main() {
   X x;
   print(&x);
   int* xp = reinterpret_cast<int*>(&x);
   for(int* i = xp; i < xp + sz; i++)
       *i = 0;
   // Can't use xp as an X* at this point
   // unless you cast it back:
   print(reinterpret_cast<X*>(xp));
   // In this example, you can also just use
   // the original identifier:
   print(&x);
} ///:~
```

Cast	Description	Safe?
static_cast	Performs compile-time type conversions between related types.	Yes
dynamic_cast	Performs runtime type conversions on pointers or references in an polymorphic (inheritance) hierarchy	Yes
const_cast	Adds or removes const.	Only for adding const
reinterpret_cast	Reinterprets the bit-level representation of one type as if it were another type	No
C-style casts	Performs some combination of `static_cast`, `const_cast`, or `reinterpret_cast`.	No

Template-uri

Function Templates

Un **function template** este un model pentru crearea de funcții generice.

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

template <typename T>
const T& template_max(const T& x, const T& y)
{
   return (x > y) ? x : y;
}

int main()
{
   int a = 3;
   int b = 7;
   cout<<template_max(a, b); // Se generează max(int, int)
   return 0;
}
```

```
#include <iostream>
template <class T>
T average(T *array, int length)
{
   T sum(0);
   for (int count{ 0 }; count < length; ++count)
       sum += array[count];
   sum /= length;
   return sum;
}
int main()
{
   int array1[]={ 5, 3, 2, 1, 4 };
   std::cout << average(array1, 5) << '\n'; // Afiseaza 3
   double array2[]={ 3.12, 3.45, 9.23, 6.34 };
   std::cout << average(array2, 4) << '\n'; // Afiseaza 5.535
   return 0;
}
```

Template classes

Clasele template permit crearea de clase generice, utile pentru containere (spre exemplu, vrem să avem vectori de int-uri, vectori de un tip A creat de noi etc). Putem face o singură clasă template care să ne permită să cream vectori pentru fiecare dintre aceste tipuri.

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

template <class T>          // Asa ii spunem compilatorului
                            // ca asta e o clasa template
class Array
{
private:
   int m_length{};
   T *m_data{};

public:
   Array(int length)
   {
       assert(length > 0);     // assert da eroare daca nu se
indeplineste conditia asta
       m_data = new T[length]{};
       m_length = length;
   }

   Array(const Array&) = delete;             // Nu avem copy
constructor.
   Array& operator=(const Array&) = delete;  // Nu avem opeartorul
=.

   ~Array()
   {
       delete[] m_data;
   }

   void erase()
   {
       delete[] m_data;                 // Dezalocam memoria
```

```
        m_data = nullptr;
        m_length = 0;
    }

    T& operator[](int index)            // Operator de indexare
    {
        assert(index >= 0 && index < m_length);
        return m_data[index];
    }

    int getLength() const;
};

// Functiile membre definite in afara clasei au nevoie de
declaratia cu template.
template <class T>
int Array<T>::getLength() const         // Numele clasei este
Array<T> nu Array.
{
    return m_length;
}

class B
{
public:
    int _x;
    B(int x = 0) : _x(x) {}
};

int main()
{
    Array<int> a(10);               // Aici se defineste (efectiv
se scrie codul)
                                          // pentru o clasa Array
in care inlocuim T cu int.
    for (int i = 0; i < 10; i++)
    {
        a[i] = i;
        cout << a[i] << " ";
    }
    cout << "\n";
```

```
    Array<B> b(20);                 // La fel aici cu B.
    for (int i = 0; i < 20; i++)
    {
        b[i] = B(i);
        cout << b[i]._x << " ";
    }

    return 0;
}
```

Template cu Parametri Non-Tip

Pana acum am vazut cum sa definim o functie sau o clasa care primeste paramtri de un tip T, template. Insa avem situatii in care ne trebuie si niste parametri ficsi, nu de tip template.

```
#include <iostream>

template <class T, int size>                // size e non-type
parameter
class StaticArray
{
private:
    T m_array[size];                        // Acel size
controleaza dimensiunea array-ului.

public:
    T* getArray();
    T& operator[](int index)
    {
        return m_array[index];
    }
};

// Cum se defineste o functie pentru o clasa cu non-type
parameter.
template <class T, int size>
T* StaticArray<T, size>::getArray()
{
    return m_array;
}
```

```
int main()
{
   StaticArray<int, 12> intArray;
   for (int i = 0; i < 12; ++i)
       intArray[i] = i;

   for (int i = 11; i >= 0; --i)
       std::cout << intArray[i] << " ";
   std::cout << '\n';

   StaticArray<double, 4> doubleArray;
   for (int i = 0; i < 4; ++i)
       doubleArray[i] = 4.4 + 0.1 * i;

   for (int i = 0; i < 4; ++i)
       std::cout << doubleArray[i] << ' ';

   return 0;
}
```

Specializarea Template-urilor

Atunci când lucrăm cu template-uri, de regulă scriem același comportament pentru fiecare tip de date cu care vom lucra. Totuși, pentru unele tipuri de date este posibil să fie nevoie să avem un comportament diferit.

De aceea, putem crea specializări ale clasei/funcției template care să implementeze funcționalități diferite pentru anumite tipuri de date. Exemplu: presupunem că avem o funcție template de sortare care folosește QuickSort pentru orice tip de date pe care îl primește. Totuși, din motive de eficiență, am vrea ca dacă primește un array de caractere să folosească Count Sort. Putem face acest lucru cu o specializare a funcției pe tipul char. Exemplu:

```
#include <iostream>
using namespace std;
template<typename T>
void sort(T array[], int n)
{
```

```
   cout << "Folosim QuickSort" << endl;
}
template<>
void sort<char>(char array[], int n) // avem o specializare a
functiei
// pentru tipul char. Deci daca
// functia se va apela cu un array
// de char-uri, se va intra pe
// aceasta specializare
{
   cout << "Folosim Count Sort" << endl;
}
int main()
{
   int a[100];
   char b[100];
   sort(a, 100); // va intra pe functia template de
   // baza
   sort(b, 100); // va intra pe specializarea cu char.
   return 0;
}
```

Clase specializate:

```
#include <iostream>
using namespace std;
template<typename T>
class test
{
public:
   test()
   {
       cout << "Constructor general" << endl;
   }
};
template<>
class test<int>
{
public:
   test()
{
```

```
        cout << "Constructor specializat pentru int" << endl;
    }
};
int main()
{
    test<char> a;       // Constructor general
    test<int> b;        // Constructor specializat pentru int
    test<test<int>> c;  // Constructor general
    return 0;
}
```

Atunci când noi creăm o specializare pentru o funcție sau pentru o clasă, compilatorul ne creează o cu totul altă funcție/clasă pentru tipul respectiv. Atunci când se decide pe care specializare să intre atunci când instanțiem un obiect sau apelăm o funcție, compilatorul va căuta prima oară o specializare cât mai apropiată de tipul de date pe care îl dăm, iar dacă nu găsește, va intra pe clasa/funcția generală.

Specializări Parțiale

Fie următoarea clasă template care primește un tip de date T și un număr întreg n și ține un array cu n obiecte de tipul T. Vom considera o funcție template print care afișează elementele unui astfel de array. Vom dori ca în cazul array-urilor de caractere să afișeze fiecare caracter fără să pună spații între ele, iar în cazul altor tipuri de date, să le afișeze cu spații între ele. Putem avea o specializare astfel:

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

template<typename T, int size>
class basic_array
{
public:
    T* get_array() { return _arr; }
    T& operator[](int index) { return _arr[index]; }

private:
    T _arr[size];
```

```
};

// Funcție generală
template<typename T, int size>
void print(basic_array<T, size>& arr)
{
   for (int i = 0; i < size; i++)
       cout << arr[i] << " ";
   cout << endl;
}

// Specializare parțială: doar pentru char
template<int size>
void print(basic_array<char, size>& arr)
{
   for (int i = 0; i < size; i++)
       cout << arr[i];
   cout << endl;
}


// specializare a funcției pentru array-uri de char de 14 caractere
template<>
void print(basic_array<char, 14>& arr)
{
   for (int i = 0; i < 14; i++)
       cout << arr[i]<<"*";
   cout << endl;
}

int main()
{
   basic_array<int, 3> test;
   for (int i = 0; i < 3; i++)
       test[i] = i;
   print(test);

   basic_array<char, 14> str14;
   strcpy(str14.get_array(), "Hello, world!");
   print(str14);

   basic_array<char, 12> str12;
```

```
    strcpy(str12.get_array(), "Hello!");
    print(str12);
    return 0;
}
```