Type conversions
-Conversii de tip-

Conversii implicite
Conversiile impilicite sunt realizate automat atunci când o valoarea este copiată într-un tip compatibil.
Exemplu:

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

Valoarea lui a trece de la *short* la *int* fara sa fie nevoie de vre-un operator explicit.
Se mai numește și *conversie standard.* Conversiile standard afectează tipurile de date fundamentale și permit conversia între tipuri numerice. (short to int, int to float, double to int...)

Unele conversii pot să nu-și păstreze valoarea exactă.
Exemplu: De la o valoarea întreagă la un tip unsigned. (-1 devine cea mai mare valoare reprezentabilă de acel tipe, -2 a doua cea mai mare, șamd).
Unele conversii pot să implice pierderea preciziei, compilatorul semnalând acest comportament cu un warning. Pentru a rezolva aceste probleme se pot folosi conversii explicite.

Conversii implicite cu clase
In privința claselor, conversiile implicite pot fi controlate prin
-Constructor cu un singur argument – permite conversii implicite de la un tip particular pentru initializarea unui obiect
-Operator de atribuire - permite conversii implicite de la un tip particular la atribuire
-Type-cast operator – permite conversii implicite la un tip particular

Exemplu:

```
// implicit conversion of classes:
#include <iostream>
using namespace std;
class A {};
class B {
public:
  // conversion from A (constructor):
  B (const A& x) {}
  // conversion from A (assignment):
  B& operator= (const A& x) {return
*this;}
  // conversion to A (type-cast operator)
  operator A() {return A();}
};
int main ()
{
  A foo;
  B bar = foo;     // calls constructor
  bar = foo;       // calls assignment
  foo = bar;       // calls type-cast
operator
  return 0;}
```

Operatorul de type-cast folosește o sintaxă particulară, folosește cuvântul cheie *operator* urmat de tipul destinație și un set de paranteze goale. Observați că tipul returnat este tipul destinației și prin urmare nu este specificat înainte de cuvântul operator.

Cast la tip (Type casting)
Multe conversii, în special cele care implică o interpretare diferită a valorii necesită o conversie explicită, cunoscută în C++ sub numele de type-casting.
Există două sintaxe: functional și c-like

```
double x = 10.3;
int y;
y = int (x);    // functional notation
y = (int) x;    // c-like cast notation
```

Această formă de type-casting este suficientă pentru multe cazuri dacă lucrăm cu tipurile de date fundamentale. Totuși acești operatori pot fi aplicați pe clase și pointeri la clase, care pot duce la un cod care deși este corect sintactic, poate provoca erori la runtime. Exemplu

```
// class type-casting
#include <iostream>
using namespace std;

class Dummy {
    double i,j;
};

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

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

Programul declară un pointer la Addition, dar îi asignează o referință a unui obiect de un tip fără legătură folosind type-cast explicit.

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

Type-castul explicit nerestrictionat permite conversia oricărui pointer la ori alt tip de pointer, independent de tipul la care ei pointează. Apelarea metodei result o să producă ori o eroare la runtime, ori o să aibe un rezultat neașteptat.
Pentru a controla aceste tipuri de conversii între clase avem diverse metode pe care o să le prezentăm mai jos.

Formatul lor este să urmărească noul tip din parantezele ascuțite și imediat după expresia din paranteze ce urmează să fie convertită.

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

Modul tradițional echivalent ar fi

```
(new_type) expression
new_type (expression)
```

dar fiecare cu caracteristicile sale proprii.

dynamic_cast

dynamic_cast poate fi folosit **doar** cu pointeri și referințe la clase. Scopul este să asigure că rezultatul conversiei pointează către un obiect complet valid la destinație.

Includem în mod natural upcastingul.

Dynamic cast poate fi folosit si pentru downcast dacă și numai dacă obiectul la care pointam este complet valid.

Exemplu:

```
1  // dynamic_cast
2  #include <iostream>
3  #include <exception>
4  using namespace std;
5
6  class Base { virtual void dummy() {} };
7  class Derived: public Base { int a; };
8
9  int main () {
10   try {
11     Base * pba = new Derived;
12     Base * pbb = new Base;
13     Derived * pd;
14
15     pd = dynamic_cast<Derived*>(pba);
16     if (pd==0) cout << "Null pointer on first
   type-cast.\n";
17
18     pd = dynamic_cast<Derived*>(pbb);
19     if (pd==0) cout << "Null pointer on second
   type-cast.\n";
20
21   } catch (exception& e) {cout << "Exception: "
   << e.what();}
22   return 0;
23 }
```

```
Null pointer on second type-
cast.
```

Compatibility note: Acest tip de dynamic_cast necesită RTTI (Run-Time Type Information) pentru a urmări tipurile dinamic. Unele compilatoare suportă acest lucru iar altele îl au dezactivat. Este necesar să fie activat pentru verificări la runtime.

Codul de mai sus încearcă să realizeze dynamic cast de la pointerul obiectului de tip baza Base* (pba și pbb) la pointerul obiectului de tip Derived*. Doar primul este realizat cu succes. Important este modul în care au fost inițializați pointerii de tip Base.

```
1 Base * pba = new Derived;
2 Base * pbb = new Base;
```

Chiar dacă ambii sunt de tip Base*, pba pointează către un obiect de tip Derived (este etichetat astfel), pe când pbb pointează către un obiect de tip Base. La type-castul realizat prin dynamic-cast, pba pointează către un obiect complet Derived, pe când pbb pointează catre un obiect din Base, care este un obiect incomplet Derived.

Când dynamic_cast nu face cast deoarece obiectul este incomplet ca in exemplul de mai sus atunci returnează un pointer *null* pentru a identifica problema. Dacă este folosit pentru a converti la o referință și conversia nu este posibilă atunci aruncă un *bad_cast.*

static_cast

`static_cast` poate realiza conversii între pointeri din clase înrudite, nu doar upcast cât și downcast. Nu sunt făcute veriricări la runtime pentru a garanta dacă obiectul la care se face conversia este într-adevăr complet. Aici programatorul trebuie să se asigure că totul funcționează corespunzător și sigur. Nu efectuează verificările făcute de dynamic_cast.

```
1 class Base {};
2 class Derived: public Base {};
3 Base * a = new Base;
4 Derived * b = static_cast<Derived*>(a);
```

Acesta ar fi un cod valid, deși b ar indica un obiect incomplet al clasei și ar putea duce la erori de execuție.
Deci static_cast este capabil să realizeze nu numai conversiile permise implicit dar și conversiile opuse.

typeid

`typeid` permite să verificăm tipul unei expresii.

```
typeid (expression)
```

Acest operator returnează o referință la un obiect constant de tipul *type_info* care este definit in header-ul <typeinfo>. O valoare returnată de *typeid* poate fi comparată cu altă valoare

```cpp
// typeid
#include <iostream>
#include <typeinfo>
using namespace std;

int main () {
  int * a,b;
  a=0; b=0;
  if (typeid(a) != typeid(b))
  {
    cout << "a and b are of different
types:\n";
    cout << "a is: " <<
typeid(a).name() << '\n';
    cout << "b is: " <<
typeid(b).name() << '\n';
  }
  return 0;
}
```

```
a and b are of different types:
a is: int *
b is: int
```

Când typeid este aplicat pe clase, folosește RTTI pentru a urmări tipul obiectelor dinamice.

```cpp
// typeid, polymorphic class
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;

class Base { virtual void f(){} };
class Derived : public Base {};

int main () {
  try {
    Base* a = new Base;
    Base* b = new Derived;
    cout << "a is: " << typeid(a).name() << '\n';
    cout << "b is: " << typeid(b).name() << '\n';
    cout << "*a is: " << typeid(*a).name() << '\n';
    cout << "*b is: " << typeid(*b).name() << '\n';
  } catch (exception& e) { cout << "Exception: " <<
e.what() << '\n'; }
  return 0;
}
```

```
a is: class Base *
b is: class Base *
*a is: class Base
*b is: class Derived
```

Note: Stringul returnat de membrul name al lui type_info depinde de specificațiile implementării de către compilator și librării. Nu este obligatoriu să aveți același output ca în exemplul de față.
Observați că pentru a vedea tipul dinamic este folosit *.

Bibliografie: http://www.cplusplus.com/doc/tutorial/typecasting/