Перетворення типів (casts)

Перетворення типів (implicit)

```
1) short a=2000;
   int b;
   b=a; // OK

2) int a1=2000;
  short b1;
  b1=a1; // Not OK
  b1=(short)a1; // OK

3) int x=1;
  unsigned y;

   y =x; // OK , but ….

4) int x1;
  unsigned y1=1;

  x1 =y1; // Not OK
```

```
5) const int a=2000;
int b;
b=a; // OK

6) int a1=2000;
const int b1;
b1=a1; // OK ? const int b1 =a1;

b1=(short)a1; // OK ?

7) int x=1;
float y; double z;

y =x; // Ok, but …
z=y; // Ok

8) double x1=1;
float y1; int z1;
y1 = x1; // OK?
z1= y1; //Ok
```

Правила перетворення типів

- Від’ємне integer в unsigned — перетворює знаковий біт на одиницю.
- При перетворенні у bool : false стає 0 (for numeric types) або *null pointer* (для вказівників); true еквівалентне іншим значенням та конвертується в 1.
- Для floating-point в integer видаляється частина за крапкою, якщо не підходить по значенню - *undefined behavior*.
- Інакше (integer-to-integer або floating-to-floating), перетворення можливе,але *implementation-specific* (може бути не портабельне).

Для не фундаментальних типів масиви та функції неявно перетворюються на вказівники, а вказівники взагалі дозволяють наступні переходи:

- Null покажчики можуть бути перетворені у вказівники будь-якого типу
- Вказівники на будь-який тип можуть бути перетворені в Null вказівники.
- *upcast* вказівника: вказівники на похідний клас можуть бути перетворені в вказівник доступного і однозначного базового класу, не змінюючи його константну (const) або волатільну (volatile) специфікацію.

Type casting (C-style cast)

- ***(new_type) expression***

- ***new_type (expression)***

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

X = double(y); // functional notation
X = (double) Y; // c-like cast notation
```

```
class Dummy {
    float i,j;
}
class Addition {
    int x,y;
  public:
    Addition (int a, int b) { x=a; y=b; }
    int result() { return x+y;}
};
// cast вказівника на клас
int main () {
  Dummy d;
  Addition * padd;
  // Це коректно - але небезпечно !!!!!
  padd = (Addition*) &d;
  cout << padd->result(); ///???
  return 0;
}
```

C++ style casts:

- **(new_type) expression** або **new_type (expression)**
  ```
  int x = (int) 2.0f;   float z= float(4.0);
  ```

- **static_cast <new_type> (expression)** — звичайне перетворення
  ```
  int x = static_cast<int>(2.456);
  float y = static_cast<int>(++x*x - 2.55);
  ```

- **dynamic_cast <new_type> (expression)** — динамічне перетворення
  ```
  int x = dynamic_cast<int> (y);
  ```
- **const_cast <new_type> (expression)** — константне перетворення
  ```
  const unsigned z; unsigned t = const_cast<unsigned>(z);
  ```

- **reinterpret_cast <new_type> (expression)** — інтерпретоване перетворення
  ```
  unsigned z [10];
  int * t = const_cast<int*>(z);
  ```

static_cast

static_cast виконує всі перетворення, які дозволені неявно (наприклад, з вказівниками на класи) та обернені від них:

- Перетворити з void * на будь-який тип вказівника.
- Перетворювати цілі, дійсні та перерахування на типи перерахування.

static_cast також може виконувати:

- Виклик конструктора з одним аргументом або оператор перетворення.
- Перетворити на посилання rvalue (аргумент що може бути у присвоєнні зправа).
- Перетворення значень класу перерахування в цілі або дійсні типи.
- Перетворити будь-який тип на void.

static_cast для класів та стандартних типів

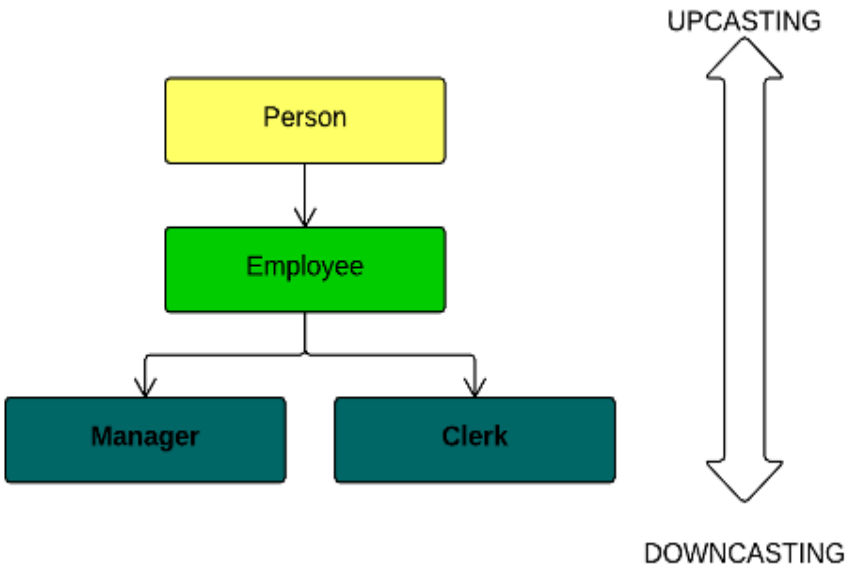
```
class Base {};
class Derived: public Base {};
****
```

```
Base * a = new Base; // вказівник на базовий клас
Derived * b = static_cast<Derived*>(a); // перетворили на клас-потомок

int a =static_cast<int>(4.578f); // для цілий в дійсний тип

float b= static_cast<float>(a*3); // для дійсний в цілий тип

long long * ptr = new long long;
int *ptr2 = static_cast<int*> (ptr); // між вказівниками
```

static_cast для класів та стандартних типів

```
Base * a = new Base(1);
Derived * b = static_cast<Derived*>(a); // Upcast — перетворення вверх іерез вказівник
int a1 =static_cast<int>(4.578f); float b1= static_cast<float>(a1*3);
void * ptr = new long long (12LL); // void * можна з будь-чого імплісіт
int *ptr2 = static_cast<int*> (ptr); // з void * до будь-якого вказівника
long long * ptr3 = new long long (44LL);
//int *ptr4 = static_cast<int*> (ptr3); // error: invalid static_cast from type ‘long long int*’ to type
‘int*’ // вказівники так просто не переводяться
int * ptr5 = new int (33);
//long long *ptr6 = static_cast<long long*> (ptr5); // і так теж
void *ptr6 = static_cast<void*> (ptr5); // у void * можна з будь-якого
long long *ptr7 = static_cast<long long*> (ptr6); // а звідси можна

Base a2(1);
//Derived b2 = static_cast<Derived>(a2); //Upcast тільки через вказівник
Derived a3(2,2);
Base b3 = static_cast<Base>(a3); // A downcast по любому
Derived * a4 = new Derived(1,1);
Base * b4 = static_cast<Base*>(a);
```

Перетворення вниз/вверх по ієрархії



Run-Time Type Information (RTTI)

dynamic_cast<new_type> (object)

dynamic_cast може використовуватися тільки з вказівниками та посиланнями на класи.Включає в себе:

- **upcast** вказівника (перетворення з вказівника на нащадка до вказівника на батька), *так само, як в неявному перетворенні*;
- може робити **downcast** (перетворення з вказівника на ***базу*** до вказівника на ***нащадок***) *поліморфні* класи (з віртуальними членами), якщо і тільки якщо - ***вказаний об'єкт є дійсним повним об'єктом цільового типу*** (безпечне перетворення)

Приклад використання typeid

```
#include <iostream>
#include <typeinfo>
using namespace std;
class BaseClass { // поліморфний
   int a, b;
  virtual void f() {};
};
class Derived1: public BaseClass {
    int i, j;};
class Derived2: public BaseClass {
    int k;};
int main() {
   int i;
   BaseClass *p, baseob;
   Derived1 ob1;
   Derived2 ob2;
   // Імя типу
   cout << "Typeid of i is ";
   cout <<typeid(i).name()<< endl;
// демонстрація typeid з поліморфами
p = &baseob;
cout << "p is pointing to an object of type ";
cout << typeid(*p).name()<<" and "<<boolalpha<<
typeid(*p).before(typeid(baseob))<<endl;
p = &ob1;
cout << "Чи є Derived1 та наслідником ";
cout << boolalpha << (typeid(*p)==
typeid(Derived1)) <<" та "<<
typeid(*p).before(typeid(baseob))<< endl;
p = &ob2;
cout << "p is pointing to an object of type ";
cout << typeid(p).name() << endl;
return 0;
}
```

```
Typeid of i is i
p is  9BaseClass and false
p is  true and true
p is  P9BaseClass
```

Приклад dynamic_cast

```
// dynamic_cast <new_type> object
#include <iostream>
#include <exception>
using namespace std;
class Base { virtual void dummy() {} };
class Derived: public Base { int a; };
int main () {
  try {   // dynamic_cast може коректно кидати виключення !
    Base * pba = new Derived;
    Base * pbb = new Base;
    Derived * pd;
    pd = dynamic_cast<Derived*>(pba); // Апкаст
    if (pd==0) cout << "Null pointer on first type-cast.\n";
    pd = dynamic_cast<Derived*>(pbb); // Даункаст
    if (pd==0) cout << "Null pointer on second type-cast.\n";

  } catch (exception& e) {cout << "Exception: " << e.what();}
  return 0;
}
```

Приклад dynamic_cast

```
#include <iostream>
using namespace std;
#define NUM_EMPLOYEES 4
class employee {
  public:     employee () { cout << "Constructing employee\n"; }
    virtual void print() = 0;
};
class programmer : public employee {
  public:     programmer() { cout << "Constructing programmer\n"; }
    void print () { cout << "Printing programmer object\n"; }
};
class salesperson : public employee {
  public:    salesperson () { cout << "Constructing salesperson\n"; }
   void print() { cout << "Printing salesperson object\n"; }
};
class executive : public employee {
  public:
   executive () { cout << "Constructing executive\n"; }
   void print () { cout << "Printing executive object\n"; }
};
```

Приклад dynamic_cast

```
int main() {
  programmer prog1, prog2;
  executive ex;   salesperson sp;
  // масив робітників
  employee *e[NUM_EMPLOYEES];
  e[0] = &prog1;
  e[1] = &sp;
  e[2] = &ex;
  e[3] = &prog2;
  // виведемо програмістів
  for (int i = 0; i < NUM_EMPLOYEES; i++) {
   programmer *pp = dynamic_cast<programmer*>(e[i]);
   if(pp) {
     cout << "Is a programmer\n";
     pp->print();
   }
   else {
    cout << "Not a programmer\n";
   }
 }
}
```

```
Constructing employee
Constructing programmer
Constructing employee
Constructing programmer
Constructing employee
Constructing executive
Constructing employee
Constructing salesperson

Is a programmer
Printing programmer object
Not a programmer
Not a programmer
Is a programmer
Printing programmer object
```

const_cast

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

void print (char * str)
{
  cout << str << '\n';
}

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

```
void func(const int * arr);

 int main(){
     arr = new int[1000];
     func(const_cast<const int*>)(arr);
    cout<<"Все гаразд"<<endl;
      delete[] arr;
 } catch(bad_alloc ex) {
     cout<<ex.what();
 }
```

reinterpret_cast

```
// reinterpret_cast  - перетворює що завгодно на що завгодно
// тому користуватися їм не радять ….
class A { /* ... */ };
class B { /* ... */ };
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

```
#include <iostream>
int main(){
  int* i;
  char *p = "This is a string";
// вказівники різного типу
  i = reinterpret_cast<int*> (p);

  std::cout << *i;
  return 0;
}
```

```
#include <iostream>
int main(){
  int i;
  char *p = "This is a string";
  // вказівник до цілого
  i = reinterpret_cast<int*> (p);
  // g++ 1.cpp -fpermissive
  std::cout << *i;
  return 0;
}
```