

ОБЪЕКТНО- ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ



Лекция № 2 / 7
15.10.2019 г.

TYPE CASTING OPERATORS

- `static_cast`
- `dynamic_cast`
- `const_cast`
- `reinterpret_cast`

STATIC_CAST

static_cast is compile time cast.

Implicit conversions between types (such as `int` to `float`, or *pointer* to `void*`)

```
float f = 3.5; //OK
int a = static_cast<int>(f); //OK
f = static_cast<float>(a); //OK

void* void_ptr = static_cast<void*>(&a); //OK
int* int_ptr = static_cast<int*>(void_ptr); //OK

unsigned* unsigned_ptr = static_cast<unsigned*>(int_ptr); //Compile error!
```


STATIC_CAST

static_cast is compile time cast.


Implicit conversions between types (such as `int` to `float`, or *pointer* to `void*`)

```
float f = 3.5; //OK
int a = static_cast<int>(f); //OK
f = static_cast<float>(a); //OK

void* void_ptr = static_cast<void*>(&a); //OK
int* int_ptr = static_cast<int*>(void_ptr); //OK

unsigned* unsigned_ptr = static_cast<unsigned*>(int_ptr); //Compile error!
```

**How cast int*
to unsigned* ???**



STATIC_CAST

static_cast is compile time cast.

Implicit conversions between types (such as `int` to `float`, or *pointer* to `void*`)

```
float f = 3.5; //OK
int a = static_cast<int>(f); //OK
f = static_cast<float>(a); //OK

void* void_ptr = static_cast<void*>(&a); //OK
int* int_ptr = static_cast<int*>(void_ptr); //OK

unsigned* unsigned_ptr = static_cast<unsigned*>(void_ptr); //OK
```

STATIC_CAST

```
class Int {  
    int x;  
public:  
  
    Int(int x_in = 0) : x{ x_in } {}  
  
    operator std::string()  
    {  
        return std::to_string(x);  
    }  
};
```

```
Int obj(3);  
std::string str = obj;  
  
std::string str2 = static_cast<std::string>(obj); //???  
obj = static_cast<Int>(30); //???
```


STATIC_CAST

```
class Int {  
    int x;  
public:  
  
    Int(int x_in = 0) : x{ x_in } {}  
  
    operator std::string()  
    {  
        return std::to_string(x);  
    }  
};
```

```
Int obj(3);  
std::string str = obj;  
  
std::string str2 = static_cast<std::string>(obj); //OK  
obj = static_cast<Int>(30); //OK
```

STATIC_CAST

static_cast can perform *upcasts/downcasts* conversions between pointers to related classes.

No checks are performed during runtime!

```
class Base {};  
class Derived: public Base {};  
Base* a = new Base;  
Derived* b = static_cast<Derived*>(a); //It's compiled.
```


STATIC_CAST

static_cast can also perform the following:

- Convert *integers*, *floating-point* values and **enum** types to **enum** types.
- Convert to *rvalue* references.
- Convert **enum class** values into *integers* or *floating-point* values.
- Convert any type to **void**, evaluating and discarding the value.

DYNAMIC_CAST

dynamic_cast can only be used with *pointers* and *references* to classes (or with `void*`).

- Pointer upcast (from `Derived*` to `Base*`).
- Downcast (from `Base*` to `Derived*`) **polymorphic classes** (those with virtual members). Compile error if classes are non-polymorphic.

If **dynamic_cast** can't cast, it will return `nullptr` in the case of a pointer, or `throw std::bad_cast` in the case of a reference.

DYNAMIC_CAST

```
class Base {};  
class Derived: public Base {};  
  
int main () {  
    Derived* pba = new Derived;  
    Base* pbb = dynamic_cast<Base*>(pba);  
    pba = dynamic_cast<Derived*>(pbb);  
  
    return 0;  
}
```

???

DYNAMIC_CAST

```
class Base {};  
class Derived: public Base {};  
  
int main () {  
    Derived* pba = new Derived;  
    Base* pbb = dynamic_cast<Base*>(pba);  
    pba = dynamic_cast<Derived*>(pbb); //Compile error!  
  
    return 0;  
}
```

DYNAMIC_CAST

```
class Base { virtual void dummy() {} };
class Derived: public Base { int a; };

int main () {
    Base* pba = new Derived;
    Base* pbb = new Base;
    Derived* pd;

    pd = dynamic_cast<Derived*>(pba);
    if (pd==nullptr) cout << "Null pointer on first type-cast.\n";

    pd = dynamic_cast<Derived*>(pbb);
    if (pd==nullptr) cout << "Null pointer on second type-cast.\n";

    return 0;
}
```

CONST_CAST

const_cast can be used to remove or add `const` to variable.

Modifying a formerly `const` value is only undefined if the original variable is `const`.

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

const char* c = "sample text";
print(const_cast<char*>(c));
```


REINTERPRET_CAST

reinterpret_cast converts any pointer type to any other pointer type, even of unrelated classes.

The operation result is a simple binary copy of the value from one pointer to the other.

It can also cast pointers to or from integer types.

REINTERPRET_CAST

```
int a = 5;  
unsigned* unsigned_ptr = reinterpret_cast<unsigned*>(&a); //It's compiled.
```

```
class A {};  
class B {};  
A* a = new A;  
B* b = reinterpret_cast<B*>(a); //It's compiled.
```

TYPEID

typeid allows to check the type of an expression in runtime:

`typeid (expression)`

This operator ignore `const` qualifiers and returns a reference to a constant object of type `type_info`.

typeid

```
int* a{}, b{};

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';
}
```

typeid

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

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';
```

Console output:

```
???
```

```
???
```

```
???
```

```
???
```

typeid

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

Console output:

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


MOVE & NOEXCEPT

```
class NoexceptMoveCtorAssign {};  
  
class NotNoexceptMoveCtorAssign { //std::is_nothrow_move_constructible_v =  
                                //false;  
    std::vector<int> member;  
};  
  
class NotNoexceptMoveAssign { //std::is_nothrow_move_constructible_v = true;  
public:  
    NotNoexceptMove(NotNoexceptMove&&) = default;  
};
```

```
class Sample {  
public:  
    ...  
    Sample(Sample&&) noexcept;  
    Sample& operator=(Sample&&) noexcept;  
};
```

**std::vector use check
on std::move_if_noexcept**

CONSTEXPR

- **constexpr** objects are **const** and are initialized with values known during compilation.
- **constexpr** functions can produce compile-time results when called with arguments whose values are known during compilation.
- **constexpr** objects and functions may be used in a wider range of contexts than non-constexpr objects and functions.
- **constexpr** is part of an object's or function's interface.

CONSTEXPR

```
int sz;  
  
constexpr auto arraySize1 = sz; // ???  
  
std::array<int, sz> data1; // ???  
  
constexpr auto arraySize2 = 10; // ???  
  
std::array<int, arraySize2> data2; // ???
```


CONSTEXPR

```
int sz; // non-constexpr variable

constexpr auto arraySize1 = sz; // error! sz's value not
                                // known at compilation

std::array<int, sz> data1; // error! same problem

constexpr auto arraySize2 = 10; // fine, 10 is a compile-time constant

std::array<int, arraySize2> data2; // fine, arraySize2 is constexpr
```


CONSTEXPR

```
constexpr
int pow(int base, int exp) noexcept // pow's a constexpr func
{ // that never throws
    ... // impl is below
}

constexpr auto numConds = 5;

std::array<int, pow(3, numConds)> results; // results has 3^numConds
// elements
```

```
auto base = readFromDB("base"); // get these values at runtime

auto exp = readFromDB("exponent");

auto baseToExp = pow(base, exp); // call pow function at runtime
```


CONSTEXPR

constexpr until C++14:

- `constexpr` functions may contain no more than a single executable statement: a `return`. Therefore, use `?:` operator and recursion.

CONSTEXPR

constexpr until C++14:

- `constexpr` functions may contain no more than a single executable statement: a `return`. Therefore, use `?:` operator and recursion.

```
constexpr int pow(int base, int exp) noexcept //until C++14
{
    return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
```

CONSTEXPR

constexpr until C++14:

- **constexpr** functions may contain no more than a single executable statement: a **return**. Therefore, use **?:** operator and recursion.

```
constexpr int pow(int base, int exp) noexcept //until C++14
{
    return (exp == 0 ? 1 : base * pow(base, exp - 1));
}
```

```
constexpr int pow(int base, int exp) noexcept //since C++14
{
    auto result = 1;
    for (int i = 0; i < exp; ++i) result *= base;
    return result;
}
```


CONSTEXPR

constexpr functions are limited to taking and returning literal types.

```
class Point {  
public:  
    constexpr Point(double xVal = 0, double yVal = 0) noexcept  
        : x(xVal), y(yVal){}  
  
    constexpr double xValue() const noexcept { return x; }  
    constexpr double yValue() const noexcept { return y; }  
    void setX(double newX) noexcept { x = newX; }  
    void setY(double newY) noexcept { y = newY; }  
  
private:  
    double x, y;  
};
```

← C++11 limitations
const function & void is not literal

```
constexpr Point p1(1.0, 2.0);
```

CONSTEXPR

constexpr functions are limited to taking and returning literal types.

```
class Point {  
public:  
    constexpr Point(double xVal = 0, double yVal = 0) noexcept  
        : x(xVal), y(yVal){}  
  
    constexpr double xValue() const noexcept { return x; }  
    constexpr double yValue() const noexcept { return y; }  
    constexpr void setX(double newX) noexcept { x = newX; }  
    constexpr void setY(double newY) noexcept { y = newY; }  
  
private:  
    double x, y;  
};
```

C++14

```
constexpr Point p1(1.0, 2.0);
```

CONSTEXPR

```
// return reflection of p with respect to the origin (C++14)
constexpr Point reflection(const Point& p) noexcept
{
    Point result; // create non-const Point

    result.setX(-p.xValue()); // set its x and y values
    result.setY(-p.yValue());

    return result; // return copy of it
}

constexpr Point p1(1.0, 2.0);
constexpr auto reflectedP1 = reflection(p1);
```


CONSTEXPR-IF

```
if constexpr (condition1)
    statement1
else if constexpr(condition2)
    statement2
else
    statement3
```

The static-if
since C++17!

```
template <typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer_v<T>)
        return *t;
    else
        return t;
}
```