C++ Type Casting

Dr. Md. Humayun Kabir
CSE Department, BUET

C++ Type Casting

- Converting a value of a given type into another type is known as *type-casting*.
- Conversion can be implicit or explicit

C++ Implicit Conversion

- Implicit conversions do not require any operator.
- They are automatically performed when a value is copied to a compatible type.

C++ Implicit Conversion

```
#include <iostream>
using namespace std;
class A {
    int x;
public:
    A(int x) {this->x=x;}
    int getX() {return x;}
};
class B{
    int x, y;
public:
    B(A a) {x=a.getX(); y=a.getX();}
    void getXY(int &x, int &y){x=this->x; y=this->y;}
};
```

C++ Implicit Conversion

```
int main(){
    int x, y;
    A a(10);
    B b=a;
    b.getXY(x,y);
    cout<<"x: "<<x<<" y: "<<y<<endl;
    return 0;
}
```

C++ Explicit Conversion

- Conversions that imply a different interpretation of the value, require an explicit conversion using type cast operator.

B b=(B) a; or B b=B (a);

- Using type cast operators indiscriminately on classes and pointers to classes can lead to code that while being syntactically correct can cause runtime errors.

C++ Explicit Conversion

```
#include <iostream>
using namespace std;
class CDummy { float i,j; };
class CAddition {
    int x,y;
public:
    CAddition (int a, int b) { x=a; y=b; }
    int result() { return x+y;}
};
```

C++ Explicit Conversion

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

C++ Explicit Conversion

- Traditional explicit type-casting allows to convert any pointer into any other pointer type, independently of the types they point to.
- The subsequent call to member function will produce either a run-time error or a unexpected result.
- In order to control these types of conversions between classes, there are four specific casting operators:
  - **dynamic_cast, reinterpret_cast, static_cast and const_cast.**

C++ dynamic_cast

- **dynamic_cast** can be used only with pointers and references to objects.
- Its purpose is to ensure that the result of the type conversion is a valid complete object of the requested class.
- **dynamic_cast** is always successful when we cast a class to one of its base classes

C++ dynamic_cast

```
class CBase { };
class CDerived: public CBase { };
CBase b;
CBase* pb;
CDerived d;
CDerived* pd;
pb = dynamic_cast<CBase*>(&d); // ok: derived-to-base
pd = dynamic_cast<CDerived*>(&b); // wrong: base-to-derived
```

C++ dynamic_cast

```
#include <iostream>
#include <exception>
using namespace std;
class CBase { virtual void dummy() {} };
class CDerived: public CBase { int a; };
int main () {
  try {
    CBase * pba = new CDerived;
    CBase * pbb = new CBase;
    CDerived * pd;
    pd = dynamic_cast<CDerived*>(pba);
    if (pd==0) cout << "Null pointer on first type-cast" << endl;
    pd = dynamic_cast<CDerived*>(pbb);
    if (pd==0) cout << "Null pointer on second type-cast" << endl;
  }
  catch (exception& e) {cout << "Exception: " << e.what();}
  return 0; }
```

C++ dynamic_cast

- When dynamic_cast cannot cast a pointer because it is not a complete object of the required class, it returns a null pointer to indicate the failure.
- If dynamic_cast is used to convert to a reference type and the conversion is not possible, an exception of type bad_cast is thrown.
- dynamic_cast can also cast null pointers even between pointers to unrelated classes.
- can also cast pointers of any type to void pointers (void*).

C++ static_cast

- **static_cast** can perform conversions between pointers to related classes, not only from the derived class to its base, but also from a base class to its derived.
- ensures that at least the classes are compatible if the proper object is converted
- but no safety check is performed during runtime to check if the object being converted is in fact a full object of the destination type.
- could lead to runtime errors

C++ static_cast

```
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a); //OK
```

C++ static_cast

- static_cast can also be used to perform any other non-pointer conversion that could also be performed implicitly
- Or any conversion between classes with explicit constructors or operator functions

C++ reinterpret_cast

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

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

C++ const_cast

- **const_cast** manipulates the constness of an object, either to be set or to be removed.

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

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

C++ typeid

- **typeid** allows to check the type of an expression
- returns a reference to a constant object of type **type_info** that is defined in the standard header file <typeinfo>
- This returned value can be compared with another one using operators == and !=
- can serve to obtain a null-terminated character sequence representing the data type or class name by using its name() member

C++ 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;
}
```

C++ typeid

```
#include <iostream>
#include <typeinfo>
#include <exception>
using namespace std;
class CBase { virtual void f(){} };
class CDerived : public CBase {};
int main () {
        try {   CBase* a = new CBase;
                CBase* b = new CDerived;
                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() << endl;
        }
        return 0;
}
```

C++ conversion function

- A conversion function that belongs to a class X specifies a conversion from the class type X to the type specified by the *conversion_type*.

```
class Y {
        int b;
public:
        operator int(){return b;}
};
void f(Y obj) {
        int i = int(obj);
        int j = (int)obj;
        int k = i + obj;
}
```

C++ conversion function restrictions

- Classes, enumerations, typedef names, function types, or array types cannot be declared or defined in the *conversion_type*.
- You cannot use a conversion function to convert an object of type A to type A, to a base class of A, or to void.
- Conversion functions have no arguments, and the return type is implicitly the conversion type.
- Conversion functions can be inherited.
- You can have virtual conversion functions but not static ones.

cout<<“Thank You”<<endl;

cout<<“**Have a Good Day**”<<endl;