Module 34: Programming in C++

Type Casting & Cast Operators: Part 3

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ernet.in

Tanwi Mallick
Srijoni Majumdar
Himadri B G S Bhuyan

NPTEL

Module Objectives

- Understand casting in C and C++

NPTEL

Module Outline

- Casting: C-Style: RECAP
  - Upcast & Downcast
- Cast Operators in C++
  - const_cast Operator
  - static_cast Operator
  - reinterpret_cast Operator
  - dynamic_cast Operator
- typeid Operator

NPTEL

Casting in C and C++

- Casting in C
  - Implicit cast
  - Explicit C-Style cast
  - Loses type information in several contexts
  - Lacks clarity of semantics
- Casting in C++
  - Performs fresh inference of types without change of value
  - Performs fresh inference of types with change of value
    - Using implicit computation
    - Using explicit (user-defined) computation
  - Preserves type information in all contexts
  - Provides clear semantics through cast operators:
    - `const_cast`
    - `static_cast`
    - `reinterpret_cast`
    - `dynamic_cast`
  - Cast operators can be grep-ed in source
  - C-Style cast must be avoided in C++

NPTEL

dynamic_cast Operator

- dynamic_cast can only be used with pointers and references to classes (or with void*)
- Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type
- This naturally includes pointer upcast (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an implicit conversion
- But dynamic_cast can also downcast (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if-and-only-if the pointed object is a valid complete object of the target type
- If the pointed object is not a valid complete object of the target type, dynamic_cast returns a null pointer
- 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 instead
- dynamic_cast can also perform the other implicit casts allowed on pointers: casting null pointers between pointers types (even between unrelated classes), and casting any pointer of any type to a void* pointer

NPTEL

dynamic_cast Operator: Pointers


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

class A { public: virtual ~A() {} };
class B: public A { };
class C { public: virtual ~C() {} };
int main() {
    A a; B b; C c; A *pA;  B *pB; C *pC; void *pV;

    pB = &b; pA = dynamic_cast<A*>(pB);
    cout << pB << " casts to " << pA << ": Up-cast: Valid" << endl;

    pA = &b; pB = dynamic_cast<B*>(pA);
    cout << pA << " casts to " << pB << ": Down-cast: Valid" << endl;

    pA = &a; pB = dynamic_cast<B*>(pA);
    cout << pA << " casts to " << pB << ": Down-cast: Invalid" << endl;

    pA = (A*)&c; pC = dynamic_cast<C*>(pA);
    cout << pA << " casts to " << pC << ": Unrelated-cast: Invalid" << endl;

    pA = 0; pC = dynamic_cast<C*>(pA);
    cout << pA << " casts to " << pC << ": Unrelated-cast: Valid for null" << endl;

    pA = &a; pV = dynamic_cast<void*>(pA);
    cout << pA << " casts to " << pV << ": Cast-to-void: Valid" << endl;

    //pA = dynamic_cast<A*>(pV); // error: 'void *': invalid expression type for dynamic_cast

    return 0;
}
```

```
Output:
00EFFCA8 casts to 00EFFCA8: Up-cast: Valid
00EFFCA8 casts to 00EFFCA8: Down-cast: Valid
00EFFCB4 casts to 00000000: Down-cast: Invalid
00EFFC9C casts to 00000000: Unrelated-cast: Invalid
00000000 casts to 00000000: Unrelated: Valid for null
00EFFCB4 casts to 00EFFCB4: Cast-to-void: Valid
```


NPTEL

dynamic_cast Operator: References

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

class A { public: virtual ~A() {} };
class B: public A { };
class C { public: virtual ~C() {} };

int main() {
    A a; B b; C c;
    try {
        B &rB1 = b;
        A &rA2 = dynamic_cast<A&>(rB1);
        cout << "Up-cast: Valid" << endl;

        A &rA3 = b;
        B &rB4 = dynamic_cast<B&>(rA3);
        cout << "Down-cast: Valid" << endl;

        try {
            A &rA5 = a;
            B &rB6 = dynamic_cast<B&>(rA5);
        } catch (bad_cast e) { cout << "Down-cast: Invalid: " << e.what() << endl; }

        try {
            A &rA7 = (A&)c;
            C &rC8 = dynamic_cast<C&>(rA7);
        } catch (bad_cast e) { cout << "Unrelated-cast: Invalid: " << e.what() << endl; }
    } catch (bad_cast e) { cout << "Bad-cast: " << e.what() << endl; }
    return 0;
}
```

```
Output:
Up-cast: Valid
Down-cast: Valid
Down-cast: Invalid: Bad dynamic_cast!
Unrelated-cast: Invalid: Bad dynamic_cast!
```

NPTEL

typeid Operator

- typeid operator is used where the dynamic type of a polymorphic object must be known and for static type identification
- typeid operator can be applied on a type or an expression
- typeid operator returns const std::type_info. The major members are:
  - operator==, operator!=: checks whether the objects refer to the same type
  - name: implementation-defined name of the type
- typeid operator works for polymorphic type only (as it uses RTTI – virtual function table)
- If the polymorphic object is bad, the typeid throws bad_typeid exception

Using typeid Operator: Polymorphic Hierarchy


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

// Polymorphic Hierarchy
class A { public: virtual ~A() {} };
class B : public A {};

int main() {
    A a;
    cout << typeid(a).name() << ": " << typeid(&a).name() << endl; // Static
    A *p = &a;
    cout << typeid(p).name() << ": " << typeid(*p).name() << endl; // Dynamic

    B b;
    cout << typeid(b).name() << ": " << typeid(&b).name() << endl; // Static
    p = &b;
    cout << typeid(p).name() << ": " << typeid(*p).name() << endl; // Dynamic

    A &r1 = a; A &r2 = b;
    cout << typeid(r1).name() << ": " << typeid(r2).name() << endl;

    return 0;
}
-----
class A: class A *
class A *: class A
class B: class B *
class A *: class B
class A: class B
```

Using typeid Operator: Staff Salary Application: Polymorphic Hierarchy


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

class Engineer { protected: string name_;
public: Engineer(const string& name) : name_(name) {}
    virtual void ProcessSalary() { cout << name_ << ": Process Salary for Engineer" << endl; }
};
class Manager : public Engineer { Engineer *reports_[10];
public: Manager(const string& name) : Engineer(name) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Manager" << endl; }
};
class Director : public Manager { Manager *reports_[10];
public: Director(const string& name) : Manager(name) {}
    void ProcessSalary() { cout << name_ << ": Process Salary for Director" << endl; }
};
int main() {
    Engineer e("Rohit"); Manager m("Kamala"); Director d("Ranjana");
    Engineer *staff[] = { &e, &m, &d };
    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
        cout << typeid(staff[i]).name() << ": " << typeid(*staff[i]).name() << endl;
    }

    return 0;
}
-----
class Engineer *: class Engineer
class Engineer *: class Manager
class Engineer *: class Director
```

Using typeid Operator: Non-Polymorphic Hierarchy

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

// Non-Polymorphic Hierarchy
class X {};
class Y : public X {};

int main() {
    X x;
    cout << typeid(x).name() << ": " << typeid(&x).name() << endl; // Static
    X *q = &x;
    cout << typeid(q).name() << ": " << typeid(*q).name() << endl; // Dynamic

    Y y;
    cout << typeid(y).name() << ": " << typeid(&y).name() << endl; // Static
    q = &y;
    cout << typeid(q).name() << ": " << typeid(*q).name() << endl; // Dynamic -- FAILS

    X &r1 = x; X &r2 = y;
    cout << typeid(r1).name() << ": " << typeid(r2).name() << endl;

    return 0;
}
-----
class X: class X *
class X *: class X
class Y: class Y *
class X *: class X
class X: class X
```


NPTEL

Using typeid Operator: bad_typeid Exception


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

class A { public: virtual ~A() {} };
class B : public A {};

int main() {
    A *pA = new A;
    try {
        cout << typeid(pA).name() << endl;
        cout << typeid(*pA).name() << endl;
    } catch (const bad_typeid& e) { cout << "caught " << e.what() << endl; }

    delete pA;
    try {
        cout << typeid(pA).name() << endl;
        cout << typeid(*pA).name() << endl;
    } catch (const bad_typeid& e) { cout << "caught " << e.what() << endl; }

    pA = 0;
    try {
        cout << typeid(pA).name() << endl;
        cout << typeid(*pA).name() << endl;
    }
    catch (const bad_typeid& e) { cout << "caught " << e.what() << endl; }

    return 0;
}
```

```
Output:
class A *
class A
class A *
caught Access violation - no RTTI data!
class A *
caught Attempted a typeid of NULL pointer!
```


NPTEL

Module Summary

- Understood casting at run-time
- Studied `dynamic_cast` with examples
- Understood RTTI and `typeid` operator

NPTEL

Instructor and TAs

Name	Mail	Mobile
Partha Pratim Das, *Instructor*	ppd@cse.iitkgp.ernet.in	9830030880
Tanwi Mallick, *TA*	tanwimallick@gmail.com	9674277774
Srijoni Majumdar, *TA*	majumdarsrijoni@gmail.com	9674474267
Himadri B G S Bhuyan, *TA*	himadribhuyan@gmail.com	9438911655

NPTEL