Module 32: Programming in C++

Type Casting & Cast Operators: Part 1

Instructors: Abir Das and Jibesh Patra

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

{*abir, jibesh*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

Module Objectives

- Understand casting in C and C++
- Understand `const_cast` operator

Module Outline

1. Type Casting
   - Upcast & Downcast
2. Cast Operators
   - `const_cast`
3. Module Summary

Type Casting

- Why type casting?
  - Type casts are used to convert the type of an object, expression, function argument, or return value to that of another type
- (Silent) Implicit conversions
  - The standard C++ conversions and user-defined conversions
- Explicit conversions
  - Often the type needed for an expression that cannot be obtained through an implicit conversion. There may be more than one standard conversion that may create an ambiguous situation or there may be disallowed conversion. We need explicit conversion in such cases
- To perform a type cast, the compiler
  - Allocates temporary storage
  - Initializes temporary with value being cast

```
double f (int i,int j) { return (double) i / j; }

// compiler generates
double f (int i, int j) {
    double temp_i = i; // Explicit conversion by (double) in temporary
    double temp_j = j; // Implicit conversion in temporary to support mixed mode
    return temp_i / temp_j;
}
```

Casting: C-Style: RECAP (Module 26)

- Various type castings are possible between built-in types

```
int i = 3;
double d = 2.5;

double result = d / i; // i is cast to double and used
```

- Casting rules are defined between numerical types, between numercial types and pointers, and between pointers to different numerical types and `void`
- Casting can be **implicit** or **explicit**

```
int i = 3;
double d = 2.5, *p = &d;

d = i;          // implicit: int to double
i = d;          // implicit: warning: '=' : conversion from 'double' to 'int': possible loss of data

d = (double)i; // explicit: int to double
i = (int)d;    // explicit: double to int

i = p;          // error: '=' : cannot convert from 'double *' to 'int'
i = (int)p;    // explicit: double * to int
```

Casting: C-Style: RECAP (Module 26)

- (**Implicit**) Casting between *unrelated classes is not permitted*

```
class A { int i; };
class B { double d; };

A a;
B b;

A *p = &a;
B *q = &b;

a = b;      // error: binary '=' : no operator which takes a right-hand operand of type 'B'
a = (A)b;   // error: 'type cast' : cannot convert from 'B' to 'A'

b = a;      // error: binary '=' : no operator which takes a right-hand operand of type 'A'
b = (B)a;   // error: 'type cast' : cannot convert from 'A' to 'B'

p = q;      // error: '=' : cannot convert from 'B *' to 'A *'
q = p;      // error: '=' : cannot convert from 'A *' to 'B *'

p = (A*)&b; // explicit on pointer: type cast is okay for the compiler
q = (B*)&a; // explicit on pointer: type cast is okay for the compiler
```

Casting: C-Style: RECAP (Module 26)

- **Forced** Casting between *unrelated classes is dangerous*

```
class A { public: int i; };
class B { public: double d; };

A a;
B b;

a.i = 5;
b.d = 7.2;

A *p = &a;
B *q = &b;

cout << p->i << endl; // prints 5
cout << q->d << endl; // prints 7.2

p = (A*)&b; // Forced casting on pointer: Dangerous
q = (B*)&a; // Forced casting on pointer: Dangerous

cout << p->i << endl; // prints -858993459:    GARBAGE
cout << q->d << endl; // prints -9.25596e+061: GARBAGE
```

Casting on a Hierarchy: C-Style: RECAP (Module 26)

- Casting on a **hierarchy** is *permitted in a limited sense*

```
class A { };
class B : public A { };

A *pa = 0;
B *pb = 0;
void *pv = 0;

pa = pb; // UPCAST: Okay

pb = pa; // DOWNCAST: error: '=' : cannot convert from 'A *' to 'B *'

pv = pa; // Okay, but lose the type for A * to void *
pv = pb; // Okay, but lose the type for B * to void *

pa = pv; // error: '=' : cannot convert from 'void *' to 'A *'
pb = pv; // error: '=' : cannot convert from 'void *' to 'B *'
```

Casting on a Hierarchy: C-Style: RECAP (Module 26)

- **Up-Casting** is *safe*

```
class A { public: int dataA_; };
class B : public A { public: int dataB_; };

A a;
B b;

a.dataA_ = 2;
b.dataA_ = 3;
b.dataB_ = 5;

A *pa = &a;
B *pb = &b;

cout << pa->dataA_ << endl;                         // prints 2
cout << pb->dataA_ << " " << pb->dataB_ << endl; // prints 3 5

pa = &b;

cout << pa->dataA_ << endl;                         // prints 3
cout << pa->dataB_ << endl;                         // error: 'dataB_' : is not a member of 'A'
```

Cast Operators

Cast Operators

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 (searched by cast operator name) in source
  - C-Style cast must be avoided in C++

Cast Operators

- A cast operator takes an expression of source type (*implicit* from the expression) and converts it to an expression of target type (*explicit* in the operator) following the semantics of the operator
- Use of cast operators increases robustness by generating errors in static or dynamic time

Cast Operators

- `const_cast` operator: `const_cast<type>(expr)`
    - Explicitly *overrides* `const` *and/or volatile* in a cast
    - Usually *does not perform computation or change value*
- `static_cast` operator: `static_cast<type>(expr)`
    - Performs a *non-polymorphic cast*
    - Usually *performs computation to change value* – implicit or user-defined
- `reinterpret_cast` operator: `reinterpret_cast<type>(expr)`
    - Casts between *unrelated pointer types* or *pointer and integer*
    - *Does not perform computation yet reinterprets value*
- `dynamic_cast` operator: `dynamic_cast<type>(expr)`
    - Performs a *run-time cast* that verifies the validity of the cast
    - *Performs pre-defined computation*, sets null or throws exception

const_cast Operator

- const_cast converts between types with different cv-qualification
- Only const_cast may be used to cast away (remove) const-ness or volatility
- Usually does not perform computation or change value

const_cast Operator

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

class A { int i_;
public: A(int i) : i_(i) { }
    int get() const { return i_; }
    void set(int j) { i_ = j; }
};
void print(char * str) { cout << str; }

int main() {
    const char * c = "sample text";
    // print(c); // error: 'void print(char *)': cannot convert argument 1 from 'const char *' to 'char *'

    print(const_cast<char *>(c)); // Okay

    const A a(1);
    a.get();

    // a.set(5); // error: 'void A::set(int)': cannot convert 'this' pointer from 'const A' to 'A &'

    const_cast<A&>(a).set(5); // Okay

    // const_cast<A>(a).set(5); // error: 'const_cast': cannot convert from 'const A' to 'A'
}
```

const_cast Operator vis-a-vis C-Style Cast


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

class A { int i_;
public: A(int i) : i_(i) { }
    int get() const { return i_; }
    void set(int j) { i_ = j; }
};
void print(char * str) { cout << str; }

int main() {
    const char * c = "sample text";

    // print(const_cast<char *>(c));
    print((char *)(c));          // C-Style Cast

    const A a(1);

    // const_cast<A&>(a).set(5);
    ((A&)a).set(5);              // C-Style Cast

    // const_cast<A>(a).set(5); // error: 'const_cast': cannot convert from 'const A' to 'A'
    ((A)a).set(5);               // C-Style Cast
}
```

const_cast Operator

```
#include <iostream>
struct type { type(): i(3) { }
    void m1(int v) const {
        //this->i = v; // error C3490: 'i' cannot be modified -- accessed through a const object
        const_cast<type*>(this)->i = v; // Okay as long as the type object isn't const
    }
    int i;
};
int main() { int i = 3;                                   // i is not declared const
    const int& cref_i = i; const_cast<int&>(cref_i) = 4; // Okay: modifies i
    std::cout << "i = " << i << '\n';

    type t; // note, if this is const type t;, then t.m1(4); may be undefined behavior
    t.m1(4);
    std::cout << "type::i = " << t.i << '\n';

    const int j = 3;                           // j is declared const
    int* pj = const_cast<int*>(&j); *pj = 4;   // undefined behavior! Value of j and *pj may differ
    std::cout << j << " " << *pj << std::endl;

    void (type::*mfp)(int) const = &type::m1; // pointer to member function
    //const_cast<void(type::*)(int)>(mfp);    // error C2440: 'const_cast': cannot convert from
                                              // 'void (__thiscall type::* )(int) const' to
                                              // 'void (__thiscall type::* )(int)' const_cast does not work
}                                             // on function pointers
```

Output:
```
i = 4
type::i = 4
3 4
```

Module Summary

- Understood casting in C and C++
- Explained cast operators in C++ and discussed the evils of C-style casting
- Studied `const_cast` with examples

Module 33: Programming in C++

Type Casting & Cast Operators: Part 2

Instructors: Abir Das and Jibesh Patra
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{*abir, sourangshu*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

Module Objectives

- Understand casting in C and C++
- Understand `static_cast`, and `reinterpret_cast` operators

Module Outline

Casting in C and C++: RECAP (Module 32)

- 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 (searched by cast operator name) in source
  - C-Style cast must be avoided in C++

static_cast Operator

- static_cast performs all conversions allowed implicitly (not only those with pointers to classes), and also the opposite of these. It can:
  - Convert from void* to any pointer type
  - Convert integers, floating-point values to enum types
  - Convert one enum type to another enum type
- static_cast can perform conversions between pointers to related classes:
  - Not only up-casts, but also down-casts
  - No checks are performed during run-time to guarantee that the object being converted is in fact a full object of the destination type
- Additionally, static_cast can also perform the following:
  - Explicitly call a single-argument constructor or a conversion operator – The User-Defined Cast
  - Convert enum values into integers or floating-point values
  - Convert any type to void, evaluating and discarding the value

static_cast Operator: Built-in Types

```
#include <iostream>
using namespace std;
int main() { // Built-in Types
    int i = 2; long j; double d = 3.7; int *pi = &i; double *pd = &d; void *pv = 0;

    i = d;                      // implicit -- warning
    i = static_cast<int>(d);    // static_cast -- okay
    i = (int)d;                 // C-style -- okay

    d = i;                      // implicit -- okay
    d = static_cast<double>(i); // static_cast -- okay
    d = (double)i;              // C-style -- okay

    pv = pi;                    // implicit -- okay
    pi = pv;                    // implicit -- error
    pi = static_cast<int*>(pv); // static_cast -- okay
    pi = (int*)pv;              // C-style -- okay

    j = pd;                     // implicit -- error
    j = static_cast<long>(pd);  // static_cast -- error
    j = (long)pd;               // C-style -- okay: sizeof(long) = 8 = sizeof(double*)
                                // RISKY: Should use reinterpret_cast
    i = (int)pd;                // C-style -- error: sizeof(int) = 4 != 8 = sizeof(double*)
                                // Refer to Module 26 for details
}
```

static_cast Operator: Class Hierarchy

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

// Class Hierarchy
class A { };
class B: public A { };

int main() {
    A a;
    B b;

    // UPCAST
    A *p = 0;
    p = &b;                    // implicit -- okay
    p = static_cast<A*>(&b); // static_cast -- okay
    p = (A*)&b;                // C-style -- okay

    // DOWNCAST
    B *q = 0;
    q = &a;                    // implicit -- error
    q = static_cast<B*>(&a); // static_cast -- okay: RISKY: Should use dynamic_cast
    q = (B*)&a;                // C-style -- okay
}
```

static_cast Operator: Pitfall


```
class Window { public:
    virtual void onResize(); ...
}
class SpecialWindow: public Window { // derived class
public:
    virtual void onResize() { // derived onResize impl;
        static_cast<Window>(*this).onResize(); // cast *this to Window, then call its onResize;
            // this doesn't work!

        ... // do SpecialWindow-specific stuff
    }
    ...
};
```

Slices the object, creates a temporary and calls the method!

```
class SpecialWindow: public Window { // derived class
public:
    virtual void onResize() { // derived onResize impl;
        Window::onResize();   // Direct call works

        ... // do SpecialWindow-specific stuff
    }
    ...
};
```

static_cast Operator: Unrelated Classes


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

// Un-related Types
class B;
class A { public:


};
class B { };

int main() {
    A a; B b;
    int i = 5;

    // B ==> A
    a = b;                // error
    a = static_cast<A>(b); // error
    a = (A)b;             // error

    // int ==> A
    a = i;                // error
    a = static_cast<A>(i); // error
    a = (A)i;             // error
}
```

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

// Un-related Types
class B;
class A { public:
    A(int i = 0) { cout << "A::A(i)\n"; }
    A(const B&) { cout << "A::A(B&)\n"; }
};
class B { };

int main() {
    A a; B b;
    int i = 5;

    // B ==> A
    a = b;                // Uses A::A(B&)
    a = static_cast<A>(b); // Uses A::A(B&)
    a = (A)b;             // Uses A::A(B&)

    // int ==> A
    a = i;                // Uses A::A(int)
    a = static_cast<A>(i); // Uses A::A(int)
    a = (A)i;             // Uses A::A(int)
}
```

static_cast Operator: Unrelated Classes


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

// Un-related Types
class B;
class A { int i_; public:


};
class B { public:

};
int main() { A a; B b; int i = 5;

    // B ==> A
    a = b;                  // error
    a = static_cast<A>(b); // error
    a = (A)b;               // error

    // A ==> int
    i = a;                    // error
    i = static_cast<int>(a); // error
    i = (int)a;               // error
}
```

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

// Un-related Types
class B;
class A { int i_; public:
    A(int i = 0) : i_(i) { cout << "A::A(i)\n"; }
    operator int() { cout << "A::operator int()\n"; return i_; }
};
class B { public:
    operator A() { cout << "B::operator A()\n"; return A(); }
};
int main() { A a; B b; int i = 5;

    // B ==> A
    a = b;                  // B::operator A()
    a = static_cast<A>(b); // B::operator A()
    a = (A)b;               // B::operator A()

    // A ==> int
    i = a;                    // A::operator int()
    i = static_cast<int>(a); // A::operator int()
    i = (int)a;               // A::operator int()
}
```

reinterpret_cast Operator

- `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
- All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked
- It can also cast pointers to or from integer types
- The format in which this integer value represents a pointer is platform-specific
- The only guarantee is that a pointer cast to an integer type large enough to fully contain it (such as `intptr_t`), is guaranteed to be able to be cast back to a valid pointer (Refer to Module 26)

reinterpret_cast Operator

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

class A { };
class B { };

int main() {
    long i = 2;
    double d = 3.7;
    double *pd = &d;

    i = pd;                          // implicit -- error
    i = reinterpret_cast<long>(pd);  // reinterpret_cast -- okay
    i = (long)pd;                    // C-style -- okay
    cout << pd << " " << i << endl;

    A *pA;
    B *pB;

    pA = pB;                         // implicit -- error
    pA = reinterpret_cast<A*>(pB);   // reinterpret_cast -- okay
    pA = (A*)pB;                     // C-style -- okay
}
```

Module Summary

- Studied `static_cast`, and `reinterpret_cast` with examples

Module 34: Programming in C++

Type Casting & Cast Operators: Part 3

Instructors: Abir Das and Jibesh Patra
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

{*abir, sourangshu*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

Module Objectives

- Understand casting in C and C++
- Understand `dynamic_cast` and `typeid` operators
- Understand RTTI

Module Outline

Module 34
Instructors: Abir Das and Jibesh Patra

Objectives & Outlines
Cast Operators
dynamic_cast
Pointers
References
typeid Operator
Polymorphic Hierarchy
Non-Polymorphic Hierarchy
bad_typeid
Run-Time Type Information
Module Summary

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 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;
    B* pB = &b;  A *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; 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; void *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
}
```

```
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
```

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; }
}
```

```
MSVC++

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

Onlinegdb
Up-cast: Valid
Down-cast: Valid
Down-cast: Invalid: std::bad_cast
Unrelated-cast: Invalid: std::bad_cast
```

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; // Dynamic
}
```

MSVC++	Onlinegdb
class A: class A *	1A: P1A
class A *: class A	P1A: 1A
class B: class B *	1B: P1B
class A *: class B	P1A: 1B
class A: class B	1A: 1B

Using typeid Operator: Polymorphic Hierarchy: Staff Salary Application


```
#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;
    }
}
```

MSVC++

```
class Engineer *: class Engineer
class Engineer *: class Manager
class Engineer *: class Director
```

Onlinegdb

```
P8Engineer: 8Engineer
P8Engineer: 7Manager
P8Engineer: 8Director
```

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; // Dynamic
}
```

MSVC++	Onlinegdb
class X: class X *	1X: P1X
class X *: class X	P1X: 1X
class Y: class Y *	1Y: P1Y
class X *: class X	P1X: 1X
class X: class X	1X: 1X

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; }
}
```

MSVC++

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

Onlinegdb

```
P1A
1A
P1A
```

Run-Time Type Information (RTTI)

- *Run-Time Type Information* or *Run-Time Type Identification* (RTTI) exposes information about an object's data type at runtime
- RTTI is a specialization of a more general concept called *Type Introspection*
- *Type Introspection* helps to examine the type or properties of an object at runtime
- RTTI can be used to do safe typecasts, using the `dynamic_cast<>` operator, and to manipulate type information at runtime, using the `typeid` operator and `std::type_info` class
- RTTI is available only *polymorphic* classes, with at least one virtual method (destructor)
- Some compilers have *flags to disable RTTI* to reduce the size of the application
- `typeid` keyword is used to determine the class of an object at run time. It returns a reference to `std::type_info` object, which exists until the end of the program

Module Summary

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

Module 35: Programming in C++

Multiple Inheritence

Instructors: Abir Das and Jibesh Patra

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

{*abir, sourangshu*}*@cse.iitkgp.ac.in*

Slides taken from NPTEL course on Programming in Modern C++

by **Prof. Partha Pratim Das**

Module Objectives

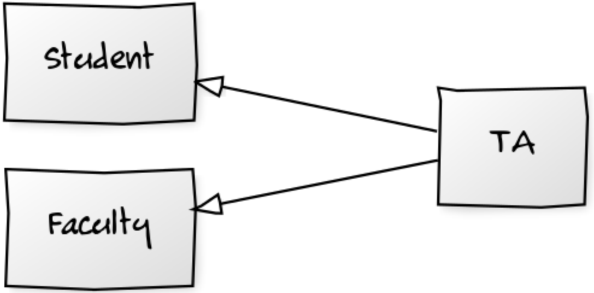
- Understand Multiple Inheritance in C++

Module Outline

Multiple Inheritance in C++: Hierarchy

Module 35
Instructors: Abir Das and Jibesh Patra

- TA **ISA** Student; TA **ISA** Faculty

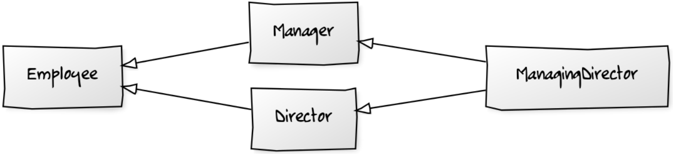



```
class Student;                                  // Base Class = Student
class Faculty;                                  // Base Class = Faculty
class TA: public Student, public Faculty; // Derived Class = TA
```

- TA inherits properties and operations of both Student as well as Faculty

Multiple Inheritance in C++: Hierarchy

- Manager **ISA** Employee, Director **ISA** Employee, ManagingDirector **ISA** Manager, ManagingDirector **ISA** Director

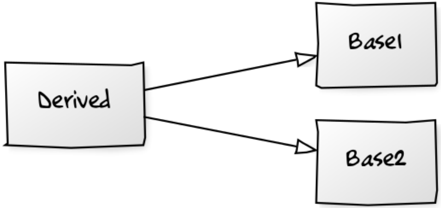



```
class Employee;                                    // Base Class = Employee -- Root
class Manager: public Employee;                    // Derived Class = Manager
class Director: public Employee;                   // Derived Class = Director
class ManagingDirector: public Manager, public Director; // Derived Class = ManagingDirector
```

- Manager inherits properties and operations of Employee
- Director inherits properties and operations of Employee
- ManagingDirector inherits properties and operations of both Manager as well as Director
- ManagingDirector, by transitivity, inherits properties and operations of Employee
- **Multiple inheritance hierarchy usually has a common base class**
- This is known as the **Diamond Hierarchy**

Multiple Inheritance in C++: Semantics

- Derived **ISA** Base1, Derived **ISA** Base2




```
class Base1;                                     // Base Class = Base1
class Base2;                                     // Base Class = Base2
class Derived: public Base1, public Base2;       // Derived Class = Derived
```

- Use keyword `public` after class name to denote inheritance
- Name of the Base class follow the keyword
- There may be more than two base classes
- `public` and `private` inheritance may be mixed

Multiple Inheritance in C++: Semantics

- Data Members
  - Derived class *inherits all* data members of *all* Base classes
  - Derived class may *add* data members of its own
- Member Functions
  - Derived class *inherits all* member functions of *all* Base classes
  - Derived class may *override* a member function of *any* Base class by *redefining* it with the same signature
  - Derived class may *overload* a member function of *any* Base class by *redefining* it with the *same name*; but *different signature*
- Access Specification
  - Derived class *cannot access* private members of *any* Base class
  - Derived class *can access* protected members of *any* Base class
- Construction-Destruction
  - A *constructor* of the Derived class *must first* call *all constructor*s of the Base classes to construct the Base class instances of the Derived class – Base class *constructor*s are called in *listing order*
  - The *destructor* of the Derived class *must* call the *destructor*s of the Base classes to destruct the Base class instances of the Derived class

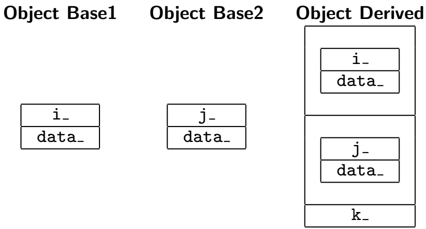
Multiple Inheritance in C++: Data Members and Object Layout

- Data Members
  - Derived class *inherits* *all* data members of *all* Base classes
  - Derived class may *add* data members of its own
- Object Layout
  - Derived class *layout* contains instances of *each* Base class
  - Further, Derived class layout will have data members of its own
  - C++ does not guarantee the relative position of the Base class instances and Derived class members

Multiple Inheritance in C++: Data Members and Object Layout

```
class Base1 { protected:
    int i_, data_;
public: // ...
};
class Base2 { protected:
    int j_, data_;
public: // ...
};
class Derived: public Base1, public Base2 { // Multiple inheritance
    int k_;
public: // ...
};
```

Object Layout



- Object Derived has two data_ members!
- Ambiguity to be resolved with base class name: Base1::data_ & Base2::data_

Multiple Inheritance in C++: Member Functions – Overrides and Overloads

- Derived **ISA** Base1, Base2
- Member Functions
  - Derived class *inherits all* member functions of *all* Base classes
  - Derived class may *override* a member function of *any* Base class by *redefining* it with the same signature
  - Derived class may *overload* a member function of *any* Base class by *redefining* it with the *same name*; but *different signature*
- Static Member Functions
  - Derived class *does not inherit* the static member functions of *any* Base class
- Friend Functions
  - Derived class *does not inherit* the friend functions of *any* Base class

Multiple Inheritance in C++: Member Functions – Overrides and Overloads

```
class Base1 { protected: int i_, data_;
public: Base1(int a, int b): i_(a), data_(b) { }
    void f(int) { cout << "Base1::f(int) \n"; }
    void g() { cout << "Base1::g() \n"; }
};
class Base2 { protected: int j_, data_;
public: Base2(int a, int b): j_(a), data_(b) { }
    void h(int) { cout << "Base2::h(int) \n"; }
};
class Derived: public Base1, public Base2 { int k_;
public: Derived(int x, int y, int u, int v, int z): Base1(x, y), Base2(u, v), k_(z) { }
    void f(int) { cout << "Derived::f(int) \n"; }       // -- Overridden Base1::f(int)
    // -- Inherited Base1::g()
    void h(string) { cout << "Derived::h(string) \n"; } // -- Overloaded Base2:: h(int)
    void e(char) { cout << "Derived::e(char) \n"; }     // -- Added Derived::e(char)
};

Derived c(1, 2, 3, 4, 5);

c.f(5);       // Derived::f(int)    -- Overridden Base1::f(int)
c.g();        // Base1::g()          -- Inherited Base1::g()
c.h("ppd");   // Derived::h(string) -- Overloaded Base2:: h(int)
c.e('a');     // Derived::e(char)   -- Added Derived::e(char)
```

Inheritance in C++: Member Functions – using for Name Resolution

Ambiguous Calls

```
class Base1 { public:
    Base1(int a, int b);
    void f(int) { cout << "Base1::f(int) "; }
    void g() { cout << "Base1::g() "; }
};
class Base2 { public:
    Base2(int a, int b);
    void f(int) { cout << "Base2::f(int) "; }
    void g(int) { cout << "Base2::g(int) "; }
};
class Derived: public Base1, public Base2 {
public: Derived(int x, int y, int u, int v, int z);


};
Derived c(1, 2, 3, 4, 5);

c.f(5); // Base1::f(int) or Base2::f(int)?
c.g(5); // Base1::g() or Base2::g(int)?
c.f(3); // Base1::f(int) or Base2::f(int)?
c.g();  // Base1::g() or Base2::g(int)?
```

Unambiguous Calls

```
class Base1 { public:
    Base1(int a, int b);
    void f(int) { cout << "Base1::f(int) "; }
    void g() { cout << "Base1::g() "; }
};
class Base2 { public:
    Base2(int a, int b);
    void f(int) { cout << "Base2::f(int) "; }
    void g(int) { cout << "Base2::g(int) "; }
};
class Derived: public Base1, public Base2 {
public: Derived(int x, int y, int u, int v, int z);
    using Base1::f; // Hides Base2::f
    using Base2::g; // Hides Base1::g
};
Derived c(1, 2, 3, 4, 5);

c.f(5);          // Base1::f(int)
c.g(5);          // Base2::g(int)
c.Base2::f(3);   // Base2::f(int)
c.Base1::g();    // Base1::g()
```

- Overload resolution does not work between Base1::g() and Base2::g(int)
- using hides other candidates; Explicit use of base class name can resolve (weak solution)

Multiple Inheritance in C++: Access Members of Base: `protected` Access

- Access Specification
  - Derived class *cannot access* private members of *any* Base class
  - Derived class *can access* protected members of *any* Base class

Multiple Inheritance in C++: Constructor & Destructor

- Constructor-Destructor
  - Derived class *inherits* *all* Constructors and Destructor of Base classes (but in a different semantics)
  - Derived class *cannot overload* a Constructor or *cannot override* the Destructor of *any* Base class
- Construction-Destruction
  - A *constructor* of the Derived class *must first* call *all constructor*s of the Base classes to construct the Base class instances of the Derived class
  - Base class *constructor*s are called in *listing order*
  - The *destructor* of the Derived class *must* call the *destructor*s of the Base classes to destruct the Base class instances of the Derived class

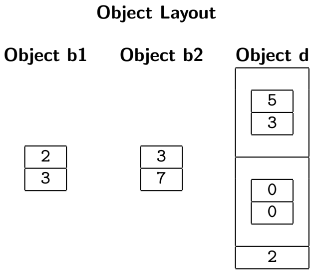
Multiple Inheritance in C++: Constructor & Destructor

```
class Base1 { protected: int i_; int data_;
public: Base1(int a, int b): i_(a), data_(b) { cout << "Base1::Base1() "; }
    ~Base1() { cout << "Base1::~Base1() "; }
};

class Base2 { protected: int j_; int data_;
public: Base2(int a = 0, int b = 0): j_(a), data_(b) { cout << "Base2::Base2() "; }
    ~Base2() { cout << "Base2::~Base2() "; }
};

class Derived: public Base1, public Base2 { int k_;
public: Derived(int x, int y, int z):
            Base1(x, y), k_(z) { cout << "Derived::Derived() "; }
            // Base1::Base1 explicit, Base2::Base2 default
    ~Derived() { cout << "Derived::~Derived() "; }
};

Base1 b1(2, 3);
Base2 b2(3, 7);
Derived d(5, 3, 2);
```



Multiple Inheritance in C++: Object Lifetime

```
class Base1 { protected: int i_; int data_;
public: Base1(int a, int b): i_(a), data_(b)
        { cout << "Base1::Base1() " << i_ << ' ' << data_ << endl; }
    ~Base1() { cout << "Base1::~Base1() " << i_ << ' ' << data_ << endl; }
};
class Base2 { protected: int j_; int data_;
public: Base2(int a = 0, int b = 0): j_(a), data_(b)
        { cout << "Base2::Base2() " << j_ << ' ' << data_ << endl; }
    ~Base2() { cout << "Base2::~Base2() " << j_ << ' ' << data_ << endl; }
};
class Derived: public Base1, public Base2 { int k_; public:
    Derived(int x, int y, int z): Base1(x, y), k_(z)
        { cout << "Derived::Derived() " << k_ << endl; }
        // Base1::Base1 explicit, Base2::Base2 default
    ~Derived() { cout << "Derived::~Derived() " << k_ << endl; }
};

Derived d(5, 3, 2);
```

Construction O/P

```
Base1::Base1(): 5, 3  // Obj. d.Base1
Base2::Base2(): 0, 0  // Obj. d.Base2
Derived::Derived(): 2 // Obj. d
```

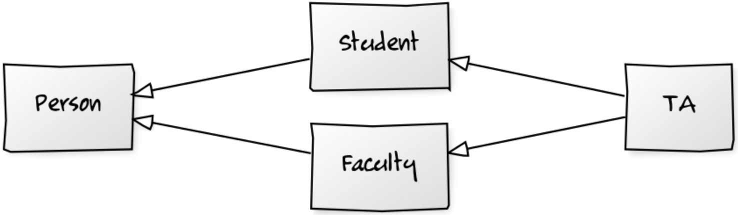
Destruction O/P

```
Derived::~Derived(): 2 // Obj. d
Base2::~Base2(): 0, 0  // Obj. d.Base2
Base1::~Base1(): 5, 3  // Obj. d.Base1
```

- First construct base class objects, then derived class object
- First destruct derived class object, then base class objects

Multiple Inheritance in C++: Diamond Problem

- Student **ISA** Person
- Faculty **ISA** Person
- TA **ISA** Student; TA **ISA** Faculty




```
class Person;                                  // Base Class = Person -- Root
class Student: public Person;                  // Base / Derived Class = Student
class Faculty: public Person;                  // Base / Derived Class = Faculty
class TA: public Student, public Faculty;      // Derived Class = TA
```

- Student inherits properties and operations of Person
- Faculty inherits properties and operations of Person
- TA inherits properties and operations of both Student as well as Faculty
- TA, by transitivity, inherits properties and operations of Person

Multiple Inheritance in C++: Diamond Problem


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

class Person { // data members of person
    public: Person(int x)  { cout << "Person::Person(int)" << endl; }
};
class Faculty: public Person { // data members of Faculty
    public: Faculty(int x): Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};
class Student: public Person { // data members of Student
    public: Student(int x): Person(x) { cout << "Student::Student(int)" << endl; }
};
class TA: public Faculty, public Student {
    public: TA(int x): Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
};
int main() { TA ta(30);
}
```

```
Person::Person(int)
Faculty::Faculty(int)
Person::Person(int)
Student::Student(int)
TA::TA(int)
```

- **Two instances of base class object (Person) in a TA object!**

Multiple Inheritance in C++: virtual Inheritance – virtual Base Class


```
#include<iostream>
using namespace std;
class Person { // data members of person
    public: Person(int x) { cout << "Person::Person(int)" << endl; }
    Person() { cout << "Person::Person()" << endl; } // Default ctor for virtual inheritance
};
class Faculty: virtual public Person { // data members of Faculty
    public: Faculty(int x): Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};
class Student: virtual public Person { // data members of Student
    public: Student(int x): Person(x) { cout << "Student::Student(int)" << endl; }
};
class TA: public Faculty, public Student {
    public: TA(int x): Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
};
int main()  { TA ta(30); }
```

```
Person::Person()
Faculty::Faculty(int)
Student::Student(int)
TA::TA(int)
```

- Introduce a default constructor for root base class Person
- Prefix every inheritance of Person with virtual
- **Only one instance of base class object (Person) in a TA object!**

Multiple Inheritance in C++: virtual Inheritance with Parameterized Ctor


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

class Person {
    public: Person(int x) { cout << "Person::Person(int)" << endl; }
    Person() { cout << "Person::Person()" << endl; }
};
class Faculty: virtual public Person {
    public: Faculty(int x): Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};
class Student: virtual public Person {
    public: Student(int x): Person(x) { cout << "Student::Student(int)" << endl; }
};
class TA: public Faculty, public Student {
    public: TA(int x): Student(x), Faculty(x), Person(x) { cout << "TA::TA(int)" << endl; }
};
int main() { TA ta(30); }
```

```
Person::Person(int)
Faculty::Faculty(int)
Student::Student(int)
TA::TA(int )
```

- **Call parameterized constructor of root base class Person from constructor of TA class**

Multiple Inheritance in C++: Ambiguity


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

class Person {
    public: Person(int x) { cout << "Person::Person(int)" << endl; }
    Person() { cout << "Person::Person()" << endl; }
    virtual ~Person();
    virtual void teach() = 0;
};
class Faculty: virtual public Person {
    public: Faculty(int x): Person(x) { cout << "Faculty::Faculty(int)" << endl; }
    virtual void teach();
};
class Student: virtual public Person {
    public: Student(int x): Person(x) { cout << "Student::Student(int)" << endl; }
    virtual void teach();
};
class TA: public Faculty, public Student {
    public: TA(int x):Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
    virtual void teach();
};
```

- **In the absence of `TA::teach()`, which of `Student::teach()` or `Faculty::teach()` should be inherited?**

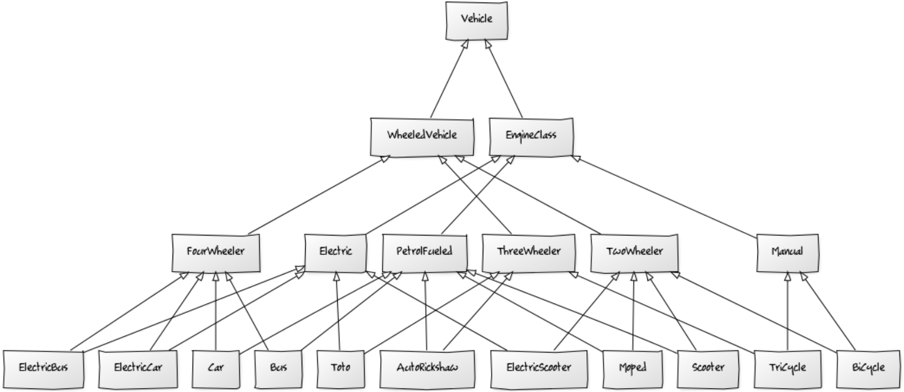
Multiple Inheritance in C++: Exercise

```
class A {
public:
    virtual ~A() { cout << "A::~A()" << endl; }
    virtual void foo() { cout << "A::foo()" << endl; }
};
class B: public virtual A {
public:
    virtual ~B() { cout << "B::~B()" << endl; }
    virtual void foo() { cout << "B::foo()" << endl; }
};
class C: public virtual A {
public:
    virtual ~C() { cout << "C::~C()" << endl; }
    virtual void foobar() { cout << "C::foobar()" << endl; }
};
class D: public B, public C {
public:
    virtual ~D() { cout << "D::~D()" << endl; }
    virtual void foo() { cout << "D::foo()" << endl; }
    virtual void foobar() { cout << "D::foobar()" << endl; }
};
```

- **Consider the effect of calling `foo` and `foobar` for various objects and various pointers**

Design Choice: Inheritance or Composition

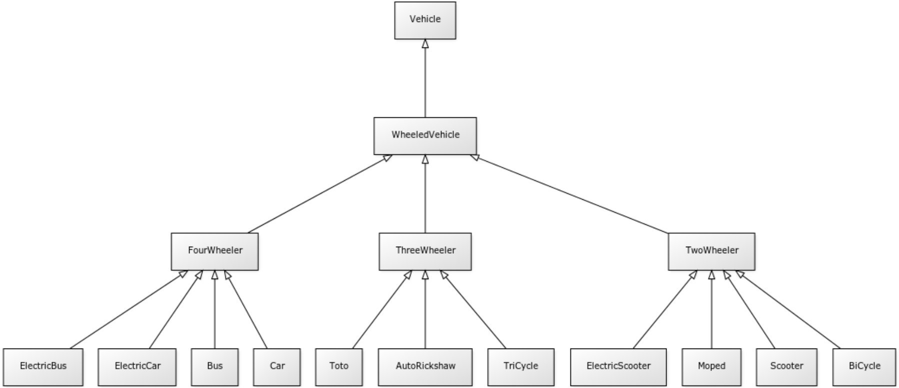
- Vehicle Hierarchy



- Wheeled Hierarchy and Engine Hierarchy interact
- Large number of cross links!
- Multiplicative options make modeling difficult

Design Choice: Inheritance or Composition

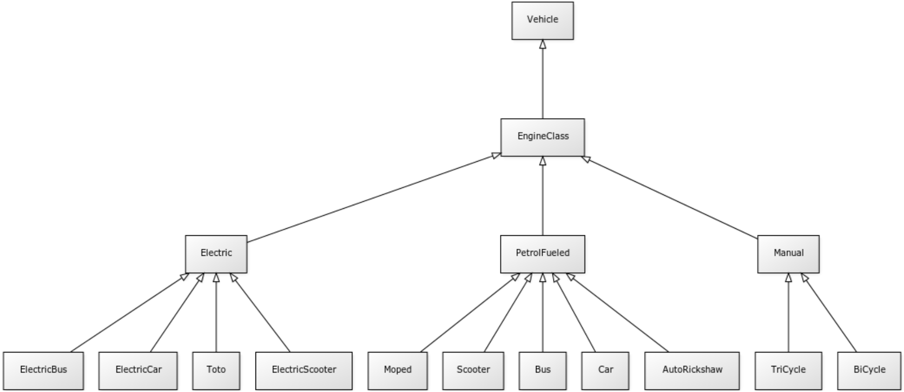
- Vehicle Hierarchy



- Wheeled Hierarchy use Engine as Component
- Linear options to simplify models
- Is this dominant?

Design Choice: Inheritance or Composition

- Vehicle Hierarchy



- Engine Hierarchy use Wheeled as Component
- Linear options to simplify models
- Is this dominant?

Module Summary

- Introduced the Semantics of Multiple Inheritance in C++
- Discussed the Diamond Problem and solution approaches
- Illustrated the design choice between inheritance and composition