Programming in Modern C++

Module M31: Virtual Function Table

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Weekly Recap

- Understood type casting – implicit as well as explicit – for built-in types, unrelated types, and classes on a hierarchy
- Understood the notions of upcast and downcast
- Understood Static and Dynamic Binding for Polymorphic type
- Understood `virtual` destructors, Pure Virtual Functions, and Abstract Base Class
- Designed the solution for a staff salary processing problem using iterative refinement – starting with a simple C solution and repeatedly refining finally to an easy, efficient, and extensible C++ solution based on flexible polymorphic hierarchy

Module Objectives

- Introduce a new C solution with function pointers
- Understand Virtual Function Table for dynamic binding (polymorphic dispatch)

Module Outline

Staff Salary Processing: New C Solution

Staff Salary Processing: Problem Statement: RECAP (Module 29)

- An organization needs to develop a salary processing application for its staff
- At present it has an engineering division only where Engineers and Managers work. Every Engineer reports to some Manager. Every Manager can also work like an Engineer
- The logic for processing salary for Engineers and Managers are different as they have different salary heads
- In future, it may add Directors to the team. Then every Manager will report to some Director. Every Director could also work like a Manager
- The logic for processing salary for Directors will also be distinct
- Further, in future it may open other divisions, like Sales division, and expand the workforce
- **Make a suitable extensible design**

C Solution: Function Pointers Engineer + Manager + Director: RECAP (Module 29)

- How to represent Engineers, Managers, and Directors?
  - Collection of `struct`s
- How to initialize objects?
  - Initialization functions
- How to have a collection of mixed objects?
  - Array of `union`
- How to model variations in salary processing algorithms?
  - `struct`-specific functions
- How to invoke the correct algorithm for a correct employee type?
  - Function switch
  - **Function pointers**

C Solution: Function Pointers: Engineer + Manager + Director

- In Module 29, we have developed a flat C Solution using *function switch*
- In Module 30, we refined the C Solution to develop two types of C++ Solution using
  - Non-polymorphic hierarchy - employing *function switch*
  - Polymorphic hierarchy - eomploying *virtual function*
- In Module 29, we had mentioned that in the flat C Solution it is not easy to use function pointers as the processing functions `void ProcessSalaryEngineer(Engineer *)`, `void ProcessSalaryManager(Manager *)`, and `void ProcessSalaryDirector(Director *)` all have different types of arguments and therefore a common function pointer type cannot be defined
- We can work around this by:
  - Passing the staff object as `void *`, instead of `Engineer *`, `Manager *`, or `Director *`
  - Cast it to respective object type in the respective function. That is, cast to `Engineer *` in `ProcessSalaryEngineer(Engineer *)` and so on
  - We can then use a function pointer type `void (*)(void *)`
- We illustrate in the Solution

C Solution: Function Pointers: Engineer + Manager + Director


```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
typedef enum E_TYPE { Er, Mgr, Dir } E_TYPE; // Staff tag type
typedef void (*psFuncPtr)(void *);          // Processing func. ptr. type, passing the object by void *
typedef struct Engineer { char *name_; } Engineer; // Engineer Type
Engineer *InitEngineer(const char *name) { Engineer *e = (Engineer *)malloc(sizeof(Engineer));
    e->name_ = strdup(name); return e;
}
void ProcessSalaryEngineer(void *v) { Engineer *e = (Engineer *)v; // Cast explicitly to the staff object
    printf("%s: Process Salary for Engineer\n", e->name_);
}
typedef struct Manager { char *name_; Engineer *reports_[10]; } Manager; // Manager Type
Manager *InitManager(const char *name) { Manager *m = (Manager *)malloc(sizeof(Manager));
    m->name_ = strdup(name); return m;
}
void ProcessSalaryManager(void *v) { Manager *m = (Manager *)v; // Cast explicitly to the staff object
    printf("%s: Process Salary for Manager\n", m->name_);
}
typedef struct Director { char *name_; Manager *reports_[10]; } Director; // Director Type
Director *InitDirector(const char *name) { Director *d = (Director *)malloc(sizeof(Director));
    d->name_ = strdup(name); return d;
}
void ProcessSalaryDirector(void *v) { Director *d = (Director *)v; // Cast explicitly to the staff object
    printf("%s: Process Salary for Director\n", d->name_);
}
```

C Solution: Function Pointers: Engineer + Manager + Director

```
typedef struct Staff {
    E_TYPE type_; // Staff tag type
    void *p;      // Pointer to staff object
} Staff;          // Staff object wrapper
int main() {
    // Array of function pointers
    psFuncPtr psArray[] = { ProcessSalaryEngineer, ProcessSalaryManager, ProcessSalaryDirector };

    // Array of staffs
    Staff staff[] = { { Er, InitEngineer("Rohit") }, { Mgr, InitEngineer("Kamala") },
                      { Mgr, InitEngineer("Rajib") }, { Er, InitEngineer("Kavita") },
                      { Er, InitEngineer("Shambhu") }, { Dir, InitEngineer("Ranjana") } };

    for (int i = 0; i < sizeof(staff) / sizeof(Staff); ++i)
        psArray[staff[i].type_] // Pick the right processing function for the tag - staff type
            (staff[i].p);       // Pass the pointer to the object - implicitly cast to void*
}
```

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
Ranjana: Process Salary for Director
```

C Solution: Advantages and Disadvantages: RECAP (Module 26) Annotated for Function Pointers

- **Advantages**
  - Solution exists!
  - Code is well structured – has patterns
- **Disadvantages**
  - Employee data has scope for better organization
    - No encapsulation for data
    - Duplication of fields across types of employees – possible to mix up types for them (say, `char *` and `string`)
    - Employee objects are created and initialized dynamically through `Init...` functions. How to release the memory?
  - Types of objects are managed explicitly by `E_Type`:
    - Difficult to extend the design – addition of a new type needs to:
      - Add new type code to `enum E_Type`
      - Add a new pointer field in `struct Staff` for the new type
      - Add a new case (`if-else` or `case`) based on the new type: **Removed using function pointer**
    - Error prone – developer has to decide to call the right processing function for every type (`ProcessSalaryManager` for `Mgr` etc.): **Removed using function pointer**
  - Unable to use Function Pointers as each processing function takes a parameter of different type - no common signature for dispatch
- **Recommendation**
  - Use classes for encapsulation on a hierarchy

Staff Salary Processing: C++ Solution

C++ Solution: Polymorphic Hierarchy: RECAP Engineer + Manager + Director: (Module 30)

- How to represent Engineers, Managers, and Directors?
  - Polymorphic `class` hierarchy
- How to initialize objects?
  - Constructor / Destructor
- How to have a collection of mixed objects?
  - `array` of base class pointers
- How to model variations in salary processing algorithms?
  - Member functions
- How to invoke the correct algorithm for a correct employee type?
  - Virtual Functions

C++ Solution: Polymorphic Hierarchy: RECAP Engineer + Manager + Director: (Module 30)


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

class Engineer {
protected:
    string name_;
public:
    Engineer(const string& name) : name_(name) { }
    virtual ~Engineer() { }
    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; }
};
```

C++ Solution: Polymorphic Hierarchy: RECAP Engineer + Manager + Director: (Module 30)


```
int main() {
    Engineer e1("Rohit"), e2("Kavita"), e3("Shambhu");
    Manager m1("Kamala"), m2("Rajib");
    Director d("Ranjana");
    Engineer *staff[] = { &e1, &m1, &m2, &e2, &e3, &d };

    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i)
        staff[i]->ProcessSalary();
}
```

```
Rohit: Process Salary for Engineer
Kamala: Process Salary for Manager
Rajib: Process Salary for Manager
Kavita: Process Salary for Engineer
Shambhu: Process Salary for Engineer
Ranjana: Process Salary for Director
```

C and C++ Solutions: A Comparison

C and C++ Solutions: A Comparison

C Solution

- How to represent Engineers, Managers, and Directors?
  - structs
- How to initialize objects?
  - Initialization functions
- How to have a collection of mixed objects?
  - array of union wrappers
- How to model variations in salary processing algorithms?
  - functions for structs
- How to invoke the correct algorithm for a correct employee type?
  - Function pointers

C++ Solution

- How to represent Engineers, Managers, and Directors?
  - Polymorphic hierarchy
- How to initialize objects?
  - Ctor / Dtor
- How to have a collection of mixed objects?
  - array of base class pointers
- How to model variations in salary processing algorithms?
  - class member functions
- How to invoke the correct algorithm for a correct employee type?
  - Virtual Functions

C and C++ Solutions: A Comparison

C Solution (Function Pointer)	C++ Solution (Virtual Function)

C Solution (Function Pointer)

```
typedef enum E_TYPE { Er, Mgr, Dir } E_TYPE;
typedef void (*psFuncPtr)(void *);
typedef struct { E_TYPE type_; void *p; } Staff;
typedef struct { char *name_; } Engineer;
Engineer *InitEngineer(const char *name);
void ProcessSalaryEngineer(void *v);
typedef struct { char *name_; } Manager;
Manager *InitManager(const char *name);
void ProcessSalaryManager(void *v);
typedef struct { char *name_; } Director;
Director *InitDirector(const char *name);
void ProcessSalaryDirector(void *v);
int main() { psFuncPtr psArray[] = {
    ProcessSalaryEngineer,     // Function
    ProcessSalaryManager,      // pointer
    ProcessSalaryDirector };   // array
    Staff staff[] = {
    { Er, InitEngineer("Rohit") },
    { Mgr, InitEngineer("Kamala") },
    { Dir, InitEngineer("Ranjana") } };
    for (int i = 0; i <
        sizeof(staff)/sizeof(Staff); ++i)
        psArray[staff[i].type_](staff[i].p);
}
```

C++ Solution (Virtual Function)

```
class Engineer { protected: string name_;
public: Engineer(const string& name);
     virtual void ProcessSalary(); };
     virtual ~Engineer(); };
class Manager : public Engineer {
public: Manager(const string& name);
    void ProcessSalary(); };
class Director : public Manager {
public: Director(const string& name);
    void ProcessSalary(); };
int main() {
    // Function pointer array is subsumed in
    // virtual function tables of classes

    Engineer e1("Rohit");
    Manager m1("Kamala");
    Director d("Ranjana");
    Engineer *staff[] = { &e1, &m1, &d };
    for(int i = 0; i <
        sizeof(staff)/sizeof(Engineer*); ++i)
        staff[i]->ProcessSalary();
}
```

Virtual Function Pointer Table

How do virtual functions work?

- The C Solution with function pointers gives us the lead to implement virtual functions. Here
  - We have used an array of function pointers (`psFuncPtr psArray[]`) to keep the processing functions (`void ProcessSalaryEngineer(Engineer *)`, `void ProcessSalaryManager(Manager *)`, and `void ProcessSalaryDirector(Director *)`) indexed by the type tag (`enum E_TYPE { Er, Mgr, Dir }`)
  - In C++, every class is a separate type - so the tag can be removed if we bind this table (**Virtual Function Table** or VFT) with the class
  - Every class can have a VFT with its appropriate processing function pointer put there
  - By override, all these functions can have the same signature (`void ProcessSalary()`) and can be called through the same expression (`(Engineer *)->ProcessSalary()`)
- We now illustrate Virtual Function Table through simple examples to show how does it work for inherited, overridden and overloaded member functions

Virtual Function Pointer Table

Base Class

```
class B {
    int i;
public:
    B(int i_): i(i_) { }
        void f(int); // B::f(B*const, int)
virtual void g(int); // B::g(B*const, int)
};

B b(100);
B *p = &b;
```

b Object Layout

	Object		*VFT*
vft	→	0	B::g(B*const, int)
B::i	100		

Source Expression	**Compiled Expression**
`b.f(15);`	`B::f(&b, 15);`
`p->f(25);`	`B::f(p, 25);`
`b.g(35);`	`B::g(&b, 35);`
`p->g(45);`	`p->vft[0](p, 45);`

Derived Class

```
class D: public B {
    int j;
public:
    D(int i_, int j_): B(i_), j(j_) { }
        void f(int); // D::f(D*const, int)
        void g(int); // D::g(D*const, int)
};

D d(200, 500);
B *p = &d;
```

d Object Layout

	Object		*VFT*
vft	→	0	D::g(D*const, int)
B::i	200		
D::j	500		

Source Expression	**Compiled Expression**
`d.f(15);`	`D::f(&d, 15);`
`p->f(25);`	`B::f(p, 25);`
`d.g(35);`	`D::g(&d, 35);`
`p->g(45);`	`p->vft[0](p, 45);`

Virtual Function Pointer Table

- Whenever a class defines a `virtual` function a hidden member variable is added to the class which points to an array of pointers to (`virtual`) functions called the **Virtual Function Table** (VFT)
- VFT pointers are used at run-time to invoke the appropriate function implementations, because at compile time it may not yet be known if the base function is to be called or a derived one implemented by a class that inherits from the base class
- VFT is class-specific – all instances of the class has the same VFT
- VFT carries the **Run-Time Type Information** (RTTI) of objects

Virtual Function Pointer Table

```
class A { public:
    virtual void f(int) { }
    virtual void g(double) { }
    int h(A *) { }
};
class B: public A { public:
    void f(int) { }
    virtual int h(B *) { }
};
class C: public B { public:
    void g(double) { }
    int h(B *) { }
};
A a; B b; C c;
A *pA; B *pB;
```

Source Expression	Compiled Expression
`pA->f(2);`	`pA->vft[0](pA, 2);`
`pA->g(3.2);`	`pA->vft[1](pA, 3.2);`
`pA->h(&a);`	`A::h(pA, &a);`
`pA->h(&b);`	`A::h(pA, &b);`
`pB->f(2);`	`pB->vft[0](pB, 2);`
`pB->g(3.2);`	`pB->vft[1](pB, 3.2);`
`pB->h(&a);`	`pB->vft[2](pB, &a);`
`pB->h(&b);`	`pB->vft[2](pB, &b);`

a Object Layout

Object: vft → *VFT*

0	`A::f(A*const, int)`	Defined
1	`A::g(A*const, double)`	Defined

b Object Layout

Object: vft → *VFT*

0	`B::f(B*const, int)`	Overridden
1	`A::g(A*const, double)`	Inherited
2	`B::h(B*const, B*)`	Overloaded

c Object Layout

Object: vft → *VFT*

0	`B::f(B*const, int)`	Inherited
1	`C::g(C*const, double)`	Overridden
2	`C::h(C*const, B*)`	Overridden

Module Summary

- Leveraging an innovative solution to the Salary Processing Application in C using function pointers, we compare C and C++ solutions to the problem
- The new C solution with function pointers is used to explain the mechanism for dynamic binding (polymorphic dispatch) based on `virtual` function tables

Programming in Modern C++

Module M32: Type Casting & Cast Operators: Part 1

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Leveraging an innovative solution to the Salary Processing Application in C using function pointers, we compare C and C++ solutions to the problem
- The new C solution with function pointers is used to explain the mechanism for dynamic binding (polymorphic dispatch) based on `virtual` function tables

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

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 my 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

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

```
#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

Programming in Modern C++

Module M33: Type Casting & Cast Operators: Part 2

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- 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 Objectives

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

Module Outline

1. Cast Operators
   - `static_cast`
     - Built-in Types
     - Class Hierarchy
     - Hierarchy Pitfall
     - Unrelated Classes
   - `reinterpret_cast`
2. Module Summary

Cast Operators

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 to rvalue references
  - 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)
- The conversions that can be performed by `reinterpret_cast` but not by `static_cast` are low-level operations based on reinterpreting the binary representations of the types, which on most cases results in code which is system-specific, and thus non-portable

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

Programming in Modern C++

Module M34: Type Casting & Cast Operators: Part 3

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

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

Module Objectives

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

Module Outline

1. Cast Operators
   - `dynamic_cast`
     - Pointers
     - References
2. `typeid` Operator
   - Polymorphic Hierarchy
   - Non-Polymorphic Hierarchy
   - `bad_typeid`
3. Run-Time Type Information (RTTI)
4. Module Summary

Cast Operators

Cast Operators

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++

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

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>
#include <typeinfo>
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

- `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


```cpp
#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++

class X: class X *
class X *: class X
class Y: class Y *
class X *: class X
class X: class X
```

```
Onlinegdb

1X: P1X
P1X: 1X
1Y: P1Y
P1X: 1X
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 (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
  - Introspection should not be confused with *reflection*, which is the ability for a program to manipulate the values, metadata, properties, and functions 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
- The use of `typeid`, in a non-polymorphic context, is often preferred over `dynamic_cast<class_type>` for efficiency
- Objects of class `std::bad_typeid` are thrown when the expression for `typeid` is the result of applying the unary * operator on a null pointer

Module Summary

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

Module M35

Partha Pratim Das

Programming in Modern C++

Module M35: Multiple Inheritance

Partha Pratim Das

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

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

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

Module Objectives

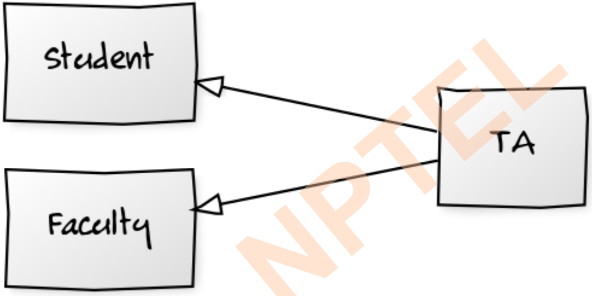
- Understand Multiple Inheritance in C++

Module Outline

Multiple Inheritance in C++

Multiple Inheritance in C++: Hierarchy

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

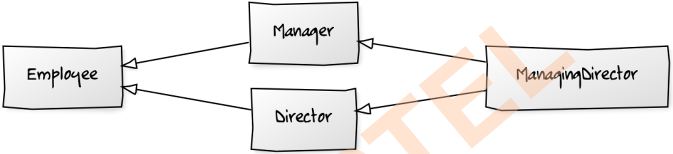



```
class Student;                                    // Base Class = Student
class Faculty;                                    // Base Class = Faculty
class TA: public Student, public Faculty; // Derived Class = TA
```

- 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

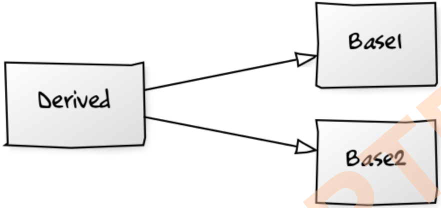



```
class Employee;                                        // Base Class = Employee -- Root
class Manager: public Employee;                        // Derived Class = Manager
class Director: public Employee;                       // Derived Class = Director
class ManagingDirector: public Manager, public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

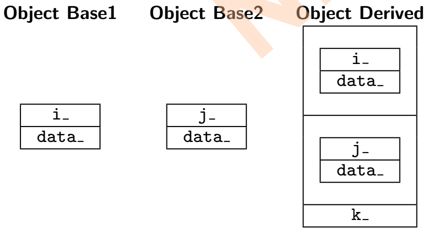
Multiple Inheritance in C++: Data Members and Object Layout

- 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	Unambiguous Calls

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 constructors* of the Base classes to construct the Base class instances of the Derived class
  - Base class *constructors* are called in *listing order*
  - The *destructor* of the Derived class *must* call the *destructors* of the Base classes to destruct the Base class instances of the Derived class

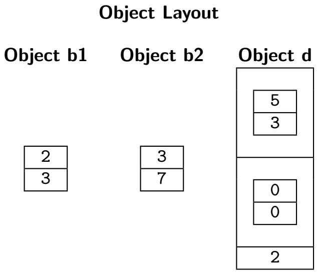
Multiple Inheritance in C++: Constructor & Destructor


```
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
```

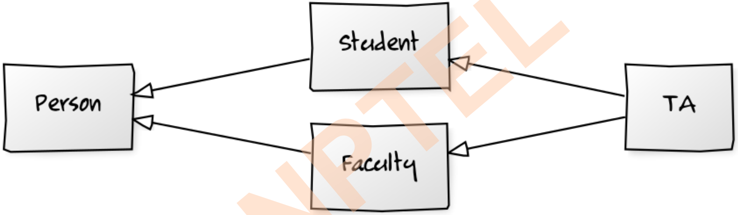
- First construct base class objects, then derived class object
- First destruct derived class object, then base class objects

Diamond Problem

Diamond Problem

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

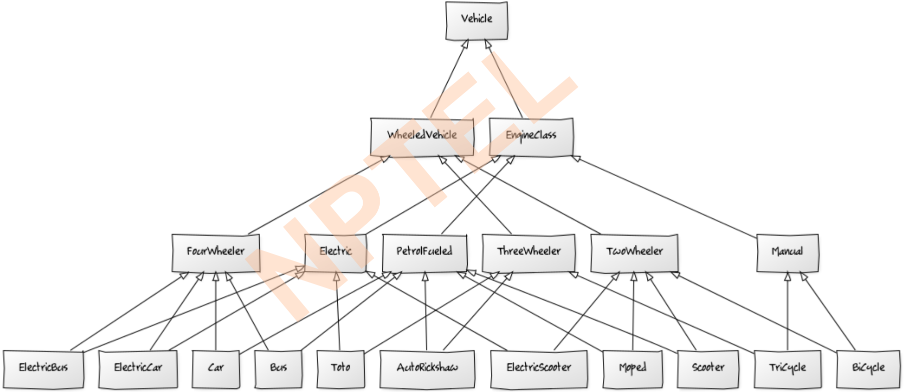
```
class A {
public:
    virtual ~A() { cout << "A::~A()" << endl; }
    virtual void foo() { cout << "A::foo()" << endl; }
};
class B: public virtual A {
public:
    virtual ~B() { cout << "B::~B()" << endl; }
    virtual void foo() { cout << "B::foo()" << endl; }
};
class C: public virtual A {
public: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

Design Choice: Inheritance or Composition

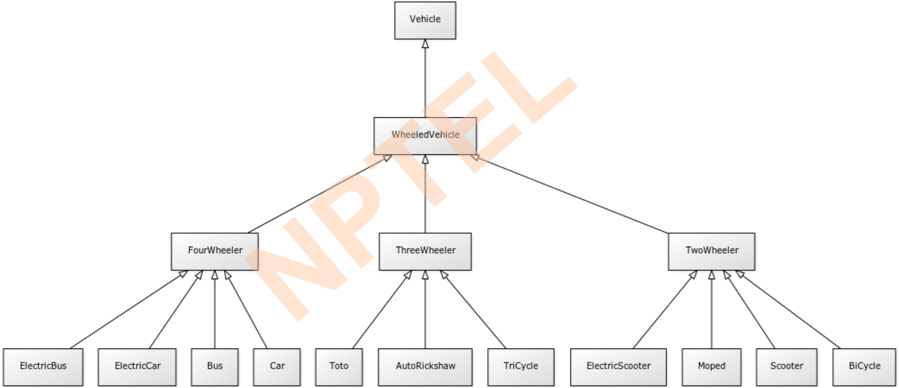
- Vehicle Hierarchy



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

Design Choice: Inheritance or Composition

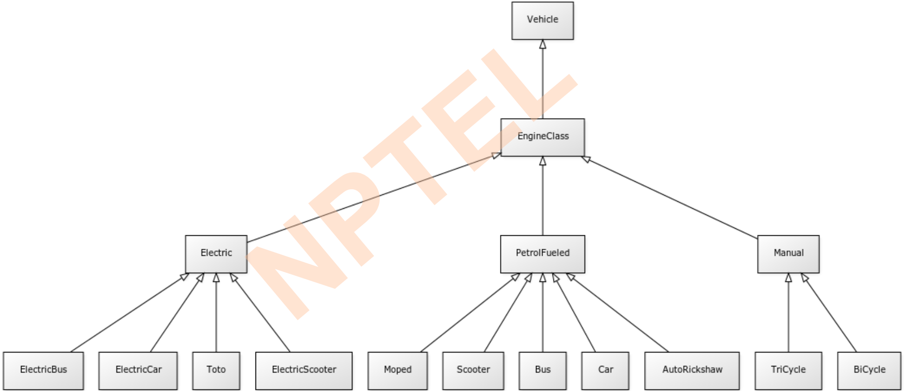
- Vehicle Hierarchy



- 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