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

Type Casting

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

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