Module 26: Programming in C++

Polymorphism: Part 1: Type Casting

Intructors: Abir Das and Sourangshu Bhattacharya

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 type casting and the difference between implicit and explicit casting
- Understand type casting in a class hierarchy
- Understand the notions of upcast and downcast

Module Outline

Type Casting: Basic Notions

- Casting is performed when a *value (variable) of one type* is used in place of *some other type*. Converting an expression of a given type into another type is known as **type-casting**

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

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

- Casting can be **implicit** or **explicit**

```
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
```

- Casting Rules can be grossly classified for:
  - Built-in types
  - Unrelated types
  - Inheritance hierarchy (static)
  - Inheritance hierarchy (dynamic)

Comparison of Implicit and Explicit Casting

Implicit Casting	**Explicit Casting**
• Done *automatically*	• Done *programatically*
• *No data loss*, for promotion Compiler will be *silent*	• Data loss *may or may not take place* Compiler will be *silent*
• *Possible data loss*, for demotion Compiler will issue *warning*	
• Requires *no special syntax*	• Requires *cast operator* for conversion *C style operator*: (< *type* >) *C++ style operators*: `const_cast`, `static_cast`, `dynamic_cast`, and `reinterpret_cast`
• *Avoid*, *if possible*	• *Avoid C style cast* *Use C++ style cast*
• *Possible only* in *static* time	• *Possible* in *static* as well as *dynamic* time
• *May be disallowed* for *User-Defined Types*, but *cannot be disallowed* for *built-in types*	• *May be defined* for *User-Defined Types*

Type Casting Rules: Built-in Type

- 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
```

Type Casting Rules: Built-in Type: Numerical Types

- Casting is *safe* for *promotion* (All the data types of the variables are upgraded to the data type of the variable with larger data type)

  `bool` → `char` → `short int` → `int` → `unsigned int` → `long` → `unsigned long long` → `float` → `double` → `long double`

- Casting in built-in types *does not invoke* any conversion function. It only *re-interprets* the binary representation
- Casting is *unsafe* for *demotion* – may lead to loss of data

Type Casting Rules: Built-in Type: Pointer Types

- Implicit casting between different pointer types is not allowed
- Any pointer can be implicitly cast to `void*` (with loss of type); but `void*` cannot be implicitly cast to any pointer type
- Conversion between array and corresponding pointer is not type casting – these are two different syntactic forms for accessing the same data

```
int i = 1, *p = &i, a[10]; double d = 1.1, *q = &d; void *r;

q = p;          // error: cannot convert ‘int*’ to ‘double*’
p = q;          // error: cannot convert ‘double*’ to ‘int*’
q = (double*)p; // Okay
p = (int*)q;    // Okay

r = p;          // Okay to convert from ‘int*’ to ‘void*’
p = r;          // error: invalid conversion from ‘void*’ to ‘int*’
p = (int*)r;    // Okay

p = a;          // Okay by array pointer duality. p[i], a[i], *(p+i), *(a+i) are equivalent
a = p;          // error: incompatible types in assignment of ‘int*’ to ‘int[10]’
```

Type Casting Rules: Built-in Type: Pointer Types

- Implicit casting between pointer type and numerical type is *not allowed*
- However, explicit casting between pointer and integral type (`int` or `long` etc.) is a common practice to support various tasks like *serialization* (save a file) and *de-serialization* (open a file)
- Care should be taken with these explicit cast to ensure that the integral type is of the *same size* as of the pointer. That is: `sizeof(void*)` = `sizeof(`$<$ *integraltype* $>$`)`

```
int i, *p = 0; long j;

// sizeof(i) = sizeof(int) = 4
// sizeof(j) = sizeof(long) = 8
// sizeof(p) = sizeof(int*) = sizeof(void*) = 8

i = p;       // error: invalid conversion from ‘int*’ to ‘int’
p = i;       // error: invalid conversion from ‘int’ to ‘int*’

i = (int)p;  // error: cast from ‘int*’ to ‘int’ loses precision
p = (int*)i; // warning: cast to pointer from integer of different size

j = (long)p; // Okay
p = (int*)j; // Okay
```

- Here, the conversion should be done between int* and long and not between int* and int

Type Casting Rules: Unrelated Classes

- (**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
```

Type Casting Rules: Unrelated Classes

- **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
```

Type Casting Rules: Inheritance Hierarchy

- 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 *'
```

Type Casting Rules: Inheritance Hierarchy

- **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'
```

Type Casting Rules: Inheritance Hierarchy

- **Down-Casting** is *risky*

```
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;

B *pb = (B*)&a;             // Forced downcast

cout << pb->dataA_ << endl; // prints 2
cout << pb->dataB_ << endl; // Compilation okay. Prints garbage for 'dataB_' -- no 'dataB_' in 'A' object
```

Module Summary

- Introduced type casting
- Understood the difference between implicit and explicit type casting
- Introduced the notions of Casting in a class hierarchy – upcast and downcast