四川大學
SICHUAN UNIVERSITY

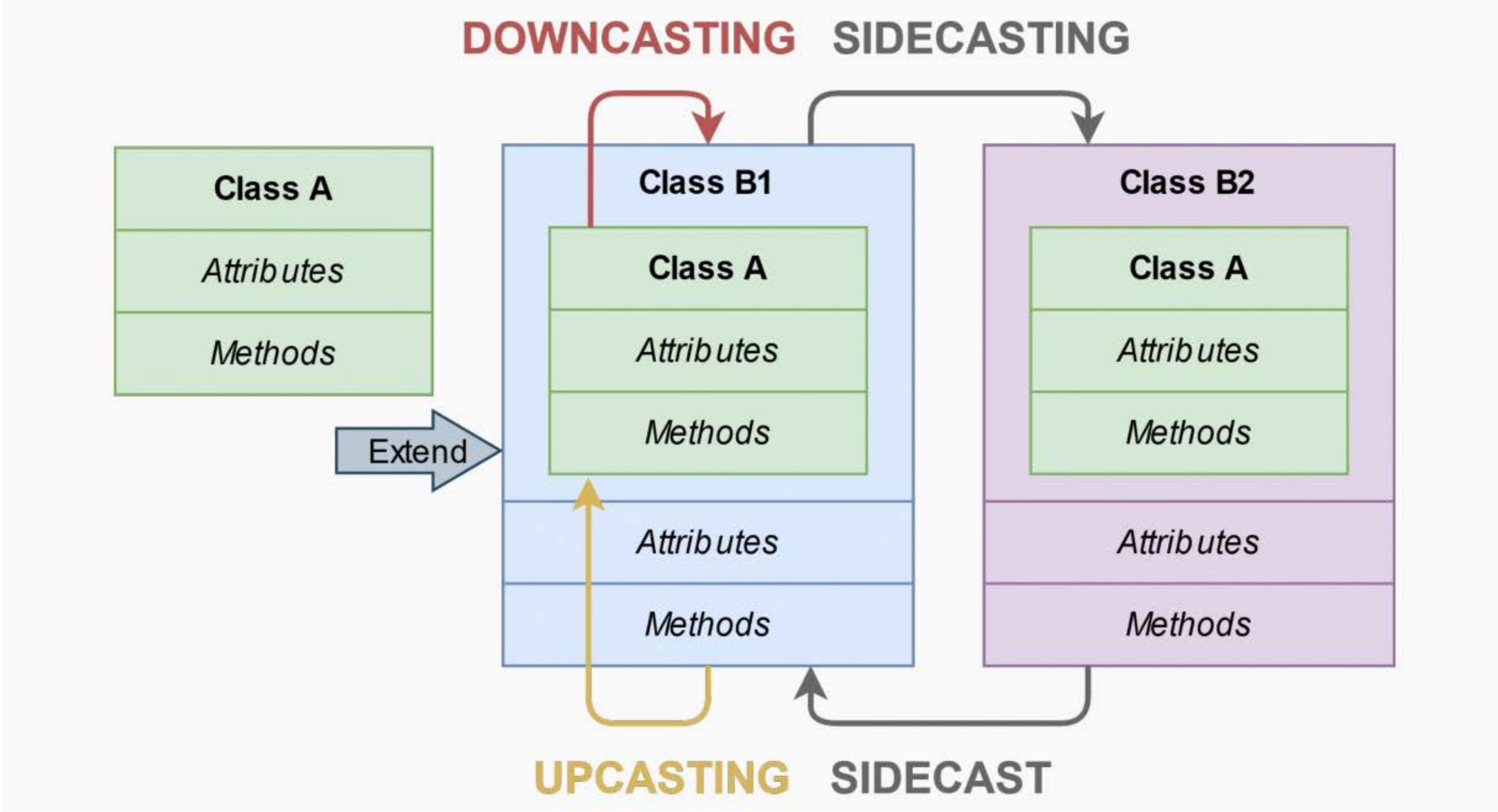
Lecture10 Inheritance Casting and Run-time Type Identification & Operator Overloading

Qijun Zhao
College of Computer Science
Sichuan University

Spring 2023

Hierarchy Casting

Class-casting allows implicit or explicit conversion of a class into another one across its hierarchy



Hierarchy Casting

Upcasting Conversion between a <u>derived</u> class reference or pointer to a <u>base</u> class

- It can be *implicit* or *explicit*
- It is safe
- `static_cast` or `dynamic_cast` // see next slides

Downcasting Conversion between a <u>base</u> class reference or pointer to a <u>derived</u> class

- It is only *explicit*
- It can be dangerous
- `static_cast` or `dynamic_cast`

Sidecasting (*Cross-cast*) Conversion between a class reference or pointer to another class of the <u>same hierarchy level</u>

- It is only *explicit*
- It can be dangerous
- `dynamic_cast`

Upcasting and Downcasting Example

```
struct A {
   virtual void f() { cout << "A"; }
};

struct B : A {
   int var;
   void f() override { cout << "B"; }
};

A a;
B b;
A& a1 = b; // implicit cast upcasting

static_cast<A&>(b).f();          // print "B" upcasting
static_cast<B&>(a1).f();         // print "A" downcasting
cout << b.var;                   // print 3
cout << static_cast<B&>(a1).var; // potential segfault!!!
```

Sidecasting Example

```
struct A {
   virtual void f() { cout << "A"; }
};

struct B1 : A {
   void f() override { cout << "B1"; }
};

struct B2 : A {
   void f() override { cout << "B2"; }
};

B1 b1;
B2 b2;
dynamic_cast<B2&>(b1).f();   // print "B2", sidecasting
dynamic_cast<B1&>(b2).f();   // print "B1", sidecasting
// static_cast<B1&>(b2).f(); // compile error
```

Run-time Type Identification

RTTI

Run-Time Type Information (RTTI) is a mechanism that allows the type of an object to be *determined at runtime*

C++ expresses RTTI through three features:

- dynamic_cast keyword: conversion of polymorphic types
- typeid keyword: identifying the exact type of an object
- type_info class: type information returned by the typeid operator

RTTI is available only for classes that are *polymorphic*, which means they have *at least one* virtual method

type info and typeid

type_info class has the method name() which returns the name of the type

```
struct A {
   virtual f() {}
};

struct B : A {};

A  a;
B  b;
A& a1 = b;
cout << typeid(a).name();  // print "1A"
cout << typeid(b).name();  // print "1B"
cout << typeid(a1).name(); // print "1A"
```

dynamic cast

`dynamic_cast` , differently from `static_cast` , uses *RTTI* for deducing the correctness of the output type

This operation happens at run-time and it is expensive

`dynamic_cast<New>(Obj)` has the following properties:

- Convert between a <u>derived</u> class `Obj` to a <u>base</u> class `New` → *upcasting*. `New` , `Obj` are both pointers or references
- Throw `std::bad_cast` if `New` , `Obj` is a reference (`T&`) and `New` , `Obj` cannot be converted
- Returns `NULL` if `New` , `Obj` are pointers (`T*`) and `New` , `Obj` cannot be converted

dynamic cast Example 1

```
struct A {
   virtual void f() { cout << "A"; }
};

struct B : A {
   void f() override { cout << "B"; }
};

A a;
B b;
dynamic_cast<A&>(b).f();     // print "B" upcasting

// dynamic_cast<B&>(a).f(); // throw std::bad_cast
                             // wrong downcasting

dynamic_cast<B*>(&a);        // returns nullptr
                             // wrong downcasting
```

dynamic cast Example 2

```
struct A {
    virtual void f() { cout << "A"; }
};
struct B : A {
    void f() override { cout << "B"; }
};

A* get_object(bool selectA) {
    return (selectA) ? new A() : new B();
}

void g(bool value) {
    A* a = get_object(value);
    B* b = dynamic_cast<B*>(a); // downcasting + check
    if (b != nullptr)
        b->f();       // exectuted only when it is safe
}
```

Operator Overloading

Operator overloading is a special case of polymorphism in which some *operators* are treated as polymorphic functions and have different behaviors depending on the type of its arguments

```
struct Point {
    int x, y;

    Point operator+(const Point& p) const {
        return {x + p.x, y + p.x};
    }
};

Point a{1, 2};
Point b{5, 3};
Point c = a + b; // "c" is (6, 5)
```

Operator Overloading

Categories not in bold are rarely used in practice

- **Arithmetic:** `+` `-` `*` `\` `%` `++` `--`
- **Comparison:** `==` `!=` `<` `<=` `>` `>=`
- Bitwise: `|` `&` `^` `~` `<<` `>>`
- Logical: `!` `&&` `||`
- **Compound assignment:** `+=` `<<=` `*=`, etc.
- **Subscript**: `[]`
- Address-of, Reference, Dereferencing: `&` `->` `->*` `*`
- Memory: `new` `new[]` `delete` `delete[]`
- Comma: `,`

Operators which cannot be overloaded: `?` `.` `.*` `::` `sizeof` `typeof`

Subscript Operator operator[]

The **array subscript** operator[] allows accessing to an object in an array-like fashion

The operator accepts everything as parameter, not just integers

```
struct A {
    char permutation[] {'c', 'b', 'd', 'a', 'h', 'y'};

    char& operator[](char c) { // read/write
        return permutation[c - 'a'];
    }
    char operator[](char c) const { // read only
        return permutation[c - 'a'];
    }
};

A a;
a['d'] = 't';
```

Comparison Operator operator<

Relational and comparison operators operator<, <=, ==, >= > are used for comparing two objects

In particular, the operator< is used to determine the ordering of a set of objects (e.g. sort)

```cpp
#include <algorithm>
struct A {
    int x;

    bool operator<(A a) const {
        return x * x < a.x * a.x;
    }
};
A array[] = {5, -1, 4, -7};
std::sort(array, array + 4);
// array: {-1, 4, 5, -7}
```

Spaceship Operator operator<=>

C++20 allows overloading the **spaceship operator** `<=>` for replacing <u>all</u> comparison operators `operator<, <=, ==, >= >`

```
struct A {
    bool operator==(const A&) const;
    bool operator!=(const A&) const;
    bool operator<(const A&)  const;
    bool operator<=(const A&) const;
    bool operator>(const A&)  const;
    bool operator>=(const A&) const;
};

// replaced by
struct B {
    int operator<=>(const B&) const;
};
```

Spaceship Operator operator<=>

```cpp
#include <compare>

struct Obj {
    int x;

    auto operator<=>(const Obj& other) {
        return x - other.x; // or even better "x <=> other.x"
    }
};

Obj a{3};
Obj b{5};
a < b;       // true, even if the operator< is not defined
a == b;      // false
a <=> b < 0; // true
```

Spaceship Operator operator<=>

The compiler can also generate the code for the *spaceship operator* = default , even for multiple fields and arrays, by using the default comparison semantic of its members

```
#include <compare>

struct Obj {
    int   x;
    char y;
    short z[2];

    auto operator<=>(const Obj&) const = default;
    // if x == other.x, then compare y
    // if y == other.y, then compare z
    // if z[0] == other.z[0], then compare z[1]
};
```

Spaceship Operator operator<=>

The *spaceship operator* can use one of the following ordering:

strong ordering
- if a is equivalent to `b` , `f(a)` is also equivalent to `f(b)`
- exactly one of `<` , `==` , or `>` must be true
  - integral types, e.g. `int` , `char`

weak ordering
- if a is equivalent to `b` , `f(a)` may not be equivalent to `f(b)`
- exactly one of `<` , `==` , or `>` must be true
  - rectangles, e.g. `R{2, 5} == R{5, 2}`

partial ordering
- if `a` is equivalent to `b` , `f(a)` may not be equivalent to `f(b)`
- `<` , `==` , or `>` may all be false
  - floating-point `float` , e.g. `NaN`

Function Call Operator operator()

The **function call operator** operator() is generally overloaded to create objects which behave like functions, or for classes that have a primary operation (see Basic Concepts IV lecture)

```cpp
#include <numeric>  // for std::accumulate

struct Multiply {
    int operator()(int a, int b) const {
        return a * b;
    }
};

int array[]    = { 2, 3, 4 };
int factorial = std::accumulate(array, array + 3, 1, Multiply{});
cout << factorial;  // 24
```

Conversion Operator operator T()

The **conversion operator** operator T() allows objects to be either implicitly or explicitly (casting) converted to another type

```
class MyBool {
    int x;
public:
    MyBool(int x1) : x{x1} {}

    operator bool() const {  // implicit return type
        return x == 0;
    }
};

MyBool my_bool{3};
bool b = my_bool;  // b = false, call operator bool()
```

Conversion Operator operator T()

C++11 **Conversion operators** can be marked explicit to prevent implicit conversions. It is a good practice as for class constructors

```
struct A {
    operator bool() { return true; }
};

struct B {
    explicit operator bool() { return true; }
};


A a;
B b;
bool    c1 = a;
// bool c2 = b;  // compile error: explicit
bool    c3 = static_cast<bool>(b);
```

Return Type Overloading Resolution

```
struct A {
    operator float() { return 3.0f; }
    operator int()   { return 2;    }
};

auto f() {
    return A{};
}

float x = f();
int   y = f();
cout << x << " " << y; // x=3.0f, y=2
```

Increment and Decrement Operators operator++/--

The increment and decrement operators operator++, operator-- are used to update the value of a variable by one unit

```
struct A {
    int* ptr;
    int  pos;
    A& operator++() {      // Prefix notation (++var):
        ++ptr;             //  returns the new copy of the object by-reference
        ++pos;
        return *this;
    }
    A operator++(int a) {  // Postfix notation (var++):
        A tmp = *this;     //  returns the old copy of the object by-value
        ++ptr;
        ++pos;
        return tmp;
    }
};
```

Assignment Operator operator=

The **assignment operator** operator= is used to copy values from one object to another *already existing* object

```
#include <algorithm>  //std::fill, std::copy
struct Array {
    char* array;
    int   size;

    Array(int size1, char value) : size{size1} {
        array = new char[size];
        std::fill(array, array + size, value);
    }
    ~Array() { delete[] array; }

    Array& operator=(const Array& x) { .... } // --> see next slide
};
Array a{5, 'o'}; // ["ooooo"]
Array b{3, 'b'}; // ["bbb"]
a = b;           // a = ["bbb"] <-- goal
```

Assignment Operator operator=

- First option:

```
Array& operator=(const Array& x) {
    if (this == &x)            // (1) Check for self assignment
        return *this;
    delete[] array;            // (2) Release class resources
    size  = x.size;            // (3) Re-initialize class resources
    array = new int[x.size];
    std::copy(x.array, x.array + size, array); // (4) deep copy
    return *this;
}
```

- Second option (less intuitive):

```
Array& operator=(Array x) { // pass by-value
    swap(this, x);          // now we need a swap function for A
    return *this;           // x is destroyed at the end
}                           // --> see next slide
```

Assignment Operator operator=

swap method:

```
friend void swap(A& x, A& y) {
    using std::swap;
    swap(x.size, y.size);
    swap(x.array, y.Array);
}
```

- **why using** std::swap ? if swap(x, y) finds a better match, it will use that instead of std::swap
- **why** friend ? it allows the function to be used from outside the structure/class scope

Stream Operator operator<<

The **stream operation** operator<< can be overloaded to perform input and output for user-defined types

```
#include <iostream>

struct Point {
    int x, y;

    friend std::ostream& operator<<(std::ostream& stream,
                                    const Point&  point) {
        stream << "(" << point.x << "," << point.y << ")";
        return stream;
    }
    // operator<< is a member of std::ostream -> need friend
}; // implementation and definition can be splitted (not suggested for operator<<)
Point point{1, 2};
std::cout << point; // print "(1, 2)"
```

Operators Precedence

```
struct MyInt {
    int x;

    int operator^(int exp) { // exponential
        int ret = 1;
        for (int i = 0; i < exp; i++)
            ret *= x;
        return ret;
    }
};
MyInt x{3};
int   y = x^2;
cout << y;  // 9
int   z = x^2 + 2;
cout << z;  // 81 !!!
```

Binary Operators Note

Binary operators should be implemented as friend methods

```
struct A {}; struct C {};

struct B : A {
    bool operator==(const A& x) { return true; }
};
struct D : C {
    friend bool operator==(const C& x, const C& y);
};
bool operator==(const C& x, const C& y); { return true; }

A a;  B b;  C c; D d;
b == a;     // ok
// a == b; // compile error // "A" does not have == operator
c == d;     // ok, use operator==(const C&, const C&)
d == c;     // ok, use operator==(const C&, const C&)
```

四川大學
SICHUAN UNIVERSITY

Coding for love,
Coding for the world

Qijun Zhao

qjzhao@scu.edu.cn