Polymorphism in C++

Lecture 13-1

Learn to be comfortable in the uncomfortable. Learn to be peaceful, not bored. Learn to be inspired, not scared.
Maxime Lagacé

Overview

- Polymorphism in C++
- Overloading
- Overriding
- Type Conversion
- Templates

Polymorphism in Java

- Overloading (Ad-hoc polymorphism)
  - Multiple methods with the same name, different signatures
  - Same operators, different signatures
- Overriding (Subtype/ Inclusion polymorphism)
  - Replacing an inherited method with another method having the same signature
- Other polymorphic characteristics
  - Casting (Polymorphic coercion)
  - Template (Parametric polymorphism)

Function Overloading

- Multiple method in a class can have the same name, but different signatures.
- Which functions to call is determined based on types of parameters.

Function Overloading Example

```cpp
#include <iostream>

void function(int n) {
    std::cout << "function(int " << n << ");\n";
}

void function(double d) {
    std::cout << "function(double " << d << ");\n";
}

int main() {
    function(3);
    function(2.718);
}
```

Output

```
function(int 3);
function(double 2.718);
```

Constructor Overloading

- We can define multiple constructors with different parameter formats.
- You can’t call another constructor with this() in a constructor to reduce the redundancy. (Different from Java)
- Instead there is delegating constructor in C++.

Constructor Overloading Example

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

class Foo {
  public:
    Foo(char x, int y) { cout << x << y << endl; }

    // Foo(int) delegates to Foo(char,int)
    Foo(int y) : Foo('a', y) {
      cout << "target" << endl;
    }
};

int main() {
  Foo f(3);
}
```

Output
a3 target

Operator Overloading

- C++ has the ability to provide the operators with a special meaning for a data type, and this ability is known as operator overloading.
- We can make operators to work for user defined classes.
- Use “operator” keyword to overload an operator.
- Go to the link for more details: https://en.cppreference.com/w/cpp/language/operators

How Does `cout <<` Works?

- cout is a global `ostream` class object, and `<<` operator is overloaded in `ostream` class and many other types.
  - For example, when `cout << 12;` is executed, `basic_ostream<_CharT, _Traits>::operator<<(int __n)` is called.
- How does `<<` chains works on `cout`?
  - `cout << 1 << 2 << 'a' << endl;`
  - Think of the return type of the above operator definition.

Make Your Own Class Printable

- How to make your own class printable like overriding toString() method in Java?
⇒ Overload the operator `<<` with a `ostream` class parameter and your own class parameter.

Make Your Own Class Printable

- Override << operator to print a class conveniently.

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

class Complex {
  float real, imag;
  public:
    Complex(float r, float i): real(r), imag(i) {}
    friend ostream&
    operator<< (ostream &os, const Complex& c);
};
ostream& operator<< (ostream &os, const Complex& c) {
    return os << c.real << '+' << c.imag << 'i';
}

int main() { Complex c(3, 4); cout << c << endl; }
```

Output

```
3+4i
```

Type Conversion (Casting)

- There are various ways to convert a type in C++:
  - Implicit Type Conversion (Automatic)
  - Explicit Type Conversion (Manually)
    - Conversion with C-style cast expression.
    - Conversion with functional cast expression.
    - Conversion using cast operator.
      - Static Cast
      - Dynamic Cast
      - Const Cast
      - Reinterpret Cast

Implicit Type Conversion

- Implicit conversions are automatically performed when a value is copied to a compatible type.
- Unlike Java, both widening and narrowing implicit conversion is possible in C++.
- It is possible for implicit conversions to lose information, signs can be lost, and overflow can occur. (narrowing conversion)

Implicit Type Conversion

- Implicit type conversion is not recommended, because your code may run not as you intended.
- See https://en.cppreference.com/w/cpp/language/implicit_conversion for more details.

Implicit Type Conversion Example

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

int main() {
  int x = 10;
  char y = 'a';
  x = x + y;
  long z = 3.56;
  cout << "x = " << x << endl
       << "y = " << y << endl
       << "z = " << z << endl;
}
```

Output

```
x = 107
y = a
z = 3
```

Explicit Type Conversion: C-Style

- We can explicitly cast a variable with parentheses as in C:

```
int sum = 17, count = 5;
float mean = (float) sum / count;
```

- C-style casting has a complex internal logic.
  - See https://en.cppreference.com/w/cpp/language/explicit_cast
- When C++ compiler encounter a C-style expression, the compiler attempts to interpret it as one of the casting operators.
- C-style casting is not recommended in C++ as its behaviour is not easily comprehensible.

Explicit Type Conversion: Functional Expression

- Functional casting:

```
float mean = float(sum) / count;
```

- Functional casting is equivalent to C-style casting.
- If the target casting type is a class, the constructor is called.

Explicit Type Conversion: Operator

- You can use cast operator to cast a variable explicitly.
- There are four types of cast operators:
  - Static Cast
  - Dynamic Cast
  - Const Cast
  - Reinterpret Cast

static_cast

- static_cast applies a set of rules to convert types. This makes the casting much safer, compared to the C-style casting.
- Simply speaking, it allows conversion between types when the conversion logically makes sense.
- static_cast is done at the compile time, so static_cast doesn’t guarantee validity of dynamic bindings.

static_cast Example

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

int main() {
    int i;
    int* ptr1 = (int*) i; // Error is not raised.
    int* ptr2 = static_cast<int*>(i); // Error is raised.
}
```

dynamic_cast

- Safely converts pointers and references to classes up, down, and sideways along the inheritance hierarchy.
- dynamic_cast can only be used with pointers and references to classes.

dynamic_cast Example

Output

const_cast

- const_cast is used to cast away constness or volatility (non-constness).
- For example, in order to pass a const pointer to a function that expects a non-const argument.

HCSLAB

const_cast Example

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

void print(char* str) {
    cout << str << endl;
}

int main() {
    const char* c = "sample text";
    //print(c); // Error occurs if uncommented
    print(const_cast<char*>(c));
}
```

Output
sample text

reinterpret_cast

- Converts between types by reinterpreting the underlying bit pattern.
- This casting is used only for low-level data manipulation.
- Not recommended for normal coding.
- More details:
  https://en.cppreference.com/w/cpp/language/reinterpret_cast

Type Conversion Summary

- Implicit type conversion, C-style casting, functional casting, and reinterpret_cast are not recommended. You should be really careful when you use these castings.
- Use dynamic_cast to guarantee safe method bindings (for overridden methods).
- Use static_cast to check casting errors at the compile time.
- Use const_cast to reference const type variables.

Templates

- Templates are a feature of C++ that allows functions, classes, variables, and aliases to operate with generic types, which are similar to Java generic.
- Templates help you to raise your code abstraction level.
- Types of templates:
  - Function Templates
  - Class Templates
  - Variable Templates (since C++14)
  - Alias Templates (since C++11)

Function Templates

- A function template defines a family of functions.
- A function template itself is not a function, and it should be instantiated with type parameters.
- You can use multiple type parameters.
- Use `template` keyword to define a function template.
- Declaration:

```
template<class identifier1, ...>
template<typename identifier1, ...>
```

- Instantiation:

```
template<identifier1, ...>(arg1, …);
```

Function Templates Example

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

template<typename T1, typename T2>
T1 add(T1 t1, T2 t2) {
    return t1 + t2;
}

int main() {
    cout << add<int, float>(3.5, 4.5) << endl;
    cout << add<float, int>(3.5, 4.5) << endl;
}
```

Output
7 7.5

Class Templates

- A class template defines a family of classes.
- A class template itself is not a type or a class, and it should be instantiated with type parameters.

Class Templates Example

```cpp
#include <iostream>
using namespace std;
template<class T>
class Pair {
  private: T values[2];
  public:
    Pair(T first, T second) {
      values[0] = first; values[1] = second;
    }
    void print() {
      cout << values[0] << ' ' << values[1] << endl;
    }
}
int main() {
    Pair<float> p(3.5, 4.1);
    p.print();
}
```

Output
3.5 4.1

Variable Templates

- A variable template defines a family of variables or static data members.
- Use `template` keyword to define a variable template.

```
template<class identifier1, ...>
template<typename identifier1, ...>
```

Variable Templates Example

```cpp
#include <iostream>
#include <iomanip> // for setprecision
using namespace std;

template<class T>
const T pi = T(3.1415926535897932385L);

int main() {
    cout << setprecision(10) << pi<int> << endl;
    cout << setprecision(10) << pi<float> << endl;
    cout << setprecision(10) << pi<double> << endl;
}
```

Output
3 3.141592741 3.141592654

Type Aliases (for Alias Templates)

- Type alias is a name that refers to a previously defined type (similar to typedef).
- Use `using` keyword to declare a type alias.

```
#include <iostream>
#include <vector>
using namespace std;
using vector2d = vector<vector<int>>;
int main() {
    vector2d v { { 0, 1, 2 }, { 3, 4, 5 }, { 6, 7, 8 } };
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) { cout << v[i][j]; }
    }
}
```

Output
012345678

Alias Templates

- Alias template is a name that refers to a family of types.
- Use `template` keyword to define an alias template.
- Syntax:

```
template<class identifier1, ...>
template<typename identifier1, ...>
```

Alias Templates

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

template<typename T>
using vector2d = vector<vector<T>>;

int main() {
  vector2d<char> v {
      { 'a', 'b', 'c' },
      { 'd', 'e', 'f' },
      { 'g', 'h', 'i' }
  };
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) { cout << v[i][j]; }
  }
}
```

Output
abcdefghi