C/C++: Lecture 6

Vorobev D.V

09.10.2020

Templates

Function template overloading

```
template<class T1, class T2>
void foo(T1 a, T2 b) { std::cout << 1; };

template<class T1>
void foo(T1 a, int b) { std::cout << 2; };

int main() {
    int a = 1;
    double b = 1.0;

    // 1
    foo(a, b);
}
```

Function template overloading

```cpp
template<class T1, class T2>
void foo(T1 a, T2 b) { std::cout << 1; };

template<class T1>
void foo(int a, T1 b) { std::cout << 2; };

int main() {
    int a = 1;
    double b = 1.0;

    // 2
    foo(a, b);
    return 0;
}
```

An explicit specialization corresponds to the first above-declared primary template the parameter of which generalizes the argument from the specialization

```cpp
template<class T>
void foo(T) { std::cout << 1; };

template<class T>
void foo(T*) { std::cout << 2; };

// specializes foo(T*)
template<>
void foo(int*) { std::cout << 3; };

int main() {
    int* a = new int(1);
    // 3
    foo(a);
}
```

```cpp
template<class T>
void foo(T) { std::cout << 1; };

// specializes foo(T)
template<>
void foo(int*) { std::cout << 3; };

template<class T>
void foo(T*) { std::cout << 2; };

int main() {
    int* a = new int(1);
    // 2
    foo(a);
}
```

```
template<class T>
void foo(T) { std::cout << 1; };

template<class T>
void foo(T*) { std::cout << 2; };

// specializes foo(T)
// Reason: T* does not generalize int
template<>
void foo(int) { std::cout << 3; };

int main() {
    int a{1};
    // 3
    foo(a);
}
```

Variadic templates

Template parameter pack

A template parameter that accepts $\geq \mathbf{0}$ template arguments

Variadic template

A template with $\geq \mathbf{1}$ template parameter pack

Examples

Variadic class template

```
template<typename ... Tail>
struct X {};

int main() {
    X<> x;
    X<int> y;
    X<int, double> z;
}
```

Variadic function template

```
template<typename ... Tail>
void foo()(Tail ... args) {}

int main() {
    foo();
    foo(1);
    foo(1, 1);
}
```

Iterating over a list of arguments

```
template<typename Tail>
void Print(Tail tail) {
    std::cout << tail;
}

template<typename Tail, typename ... Head>
void Print(Tail tail, Head ... head) {
    std::cout << tail;
    Print(head...);
}
```

std::sort

A comparator is passed as the third argument

```
template<class RandomIt, class Compare>
void sort(RandomIt first, RandomIt last, Compare comp);
```

std::less

```
template<typename T>
struct less {
    bool operator()(const T& a, const T& b) const {
        return a < b;
    }
};
```

std::greater

```
template<typename T>
struct greater {
    bool operator()(const T& a, const T& b) const {
        return a > b;
    }
};
```

Exceptions

throw, try, catch keywords

```
int main() {
    try {
        throw 1;
    } catch (int i) {
        std::cout << i;
    }
}
```

exception and runtime error differences

RunTime error

It is a drift from the program standard execution scenario.

Exception

It is a drift from the program standard execution scenario that can be handled

A runtime error that can not be handled

Segmentation fault

```
int main() {
    int x[10];
    try {
        x[100000000000] = 20;
    } catch(...) {
        std::cout << 1;
    }
}
```

new may throw

```
int main() {
    try {
        int* x = new int[100000000000];
    } catch(std::bad_alloc& e) {
        std::cout << e.what();
    }
}
```

Rethrowing the exception

```
int main() {
    try {
        int* x = new int[100000000000];
    } catch(std::bad_alloc& e) {
        std::cout << e.what();
        // пробросили
        throw e;
    }
}
```

Implicit conversions

Implicit conversions do not take place for built in types

```
int main() {
    try {
        // output: "int"
        throw 1;
    } catch (double x) {
        std::cout << "double";
    } catch (int x) {
        std::cout << "int";
    }
}
```

Exception matching

If catch-clause matches an exceptions, no other catch-clauses are considered

```
int main() {
    try {
        throw 1;
    } catch (int x) {
        // This catch matches an exception
        std::cout << 1;
    } catch (...) {
        // This one is not considered
        std::cout << 2;
    }
}
```

Exceptions and Inheritance

Implicit conversions take place for derived classes

```
int main() {
    try {
        throw Derived();
    } catch (Base& e) {
        // This catch matches an exception
        std::cout << "d";
    }  catch (Derived& d) {
        std::cout << "d";
    }
}
```

Catching exceptions rule

Catch-clauses must be declared from the most concrete type to the most general type

```
int main() {
    try {
        int* x = new int[100000000000];
    } catch (std::bad_alloc& e) {
        // The most concrete type
        // This catch matches an exception
        std::cout << "bad_alloc";
    } catch (std::exception& e) {
        // The most general type
        std::cout << "exception";
    } catch (...) {
        std::cout << "all";
    }
}
```

Catching by reference

```cpp
class Container {};

int main() {
    try {
        Container cont;
        // 1. making a copy
        throw cont;
    } catch (Container& e) {
        // 2. no copying
    }
}
```

Catching by value

```
class Container {};

 int main() {
    try {
        Container cont;
        // 1. making a copy
        throw s;
    } catch (Container e) {
        // 2. making a copy
    }
}
```