Object-Oriented Programming

C++ Constructor Specifiers

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Constructor Specifiers

- `explicit`
- `mutable`
- `delete`
- `default`

Original Code

```
class Buffer {
public:
    Buffer(int n);
};

void useBuffer(const Buffer&);

int main() {

    Buffer b(500);

    useBuffer(4);

    return 0;
}
```

What The Compiler Does

```cpp
class Buffer {
public:
    Buffer(int n);
};

void useBuffer(const Buffer&);

int main() {

    Buffer b(500);

    useBuffer(4);

    return 0;
}
```

```cpp
class Buffer {
public:
    Buffer(int n);
};

void useBuffer(const Buffer&);

int main() {

    Buffer b(500);

    useBuffer(Buffer(4));

    return 0;
}
```

Problem

```
class Buffer {
public:
    Buffer(int n);
};

void useBuffer(const Buffer&);

int main() {

    Buffer b(500);

    useBuffer(Buffer(4));

    return 0;
}
```

- Compiler can perform a single *implicit conversion* to get a type

- Any single parameter constructors can be automatically called to get the type of an argument

- Single-parameter constructor `Buffer(int n)` is being used

Why?

```
void operation(std::string s);
// ...
operation("abc");
```

Use of `explicit`

```
class Buffer {
public:
    explicit Buffer(int n);
};

void useBuffer(const Buffer&);

int main() {

    Buffer b(500);

    useBuffer(4);

    return 0;
}
```

Use of `explicit`

```
ConstructorExplicit.cpp:12:5: error: no matching function for call to 'useBuffer'
    useBuffer(4);
    ^~~~~~~~~
ConstructorExplicit.cpp:6:6: note: candidate function not viable: no known conversion from 'int' to 'const Buffer' for 1st
      argument
void useBuffer(const Buffer&);
     ^
```

`explicit` Notes

```
class Buffer {
public:
    explicit Buffer(int n);
};

void useBuffer(const Buffer&);

int main() {

    Buffer b(500);

    useBuffer(4);

    return 0;
}
```

- The `explicit` specifier prevents the compiler from using that constructor for implicit conversion

- Consider it a good practice of always using `explicit` on single-parameter constructors until it causes a problem

Code

```
class Buffer {
public:
    explicit Buffer(int n);

    void operation() const {
        // record times called
        ++counter;

        // do work ...
    }
private:
    int counter = 0;
};

int main() {

    const Buffer b(500);

    b.operation();

    return 0;
}
```

Code Error

```
ConstructorMutable.cpp:7:6: error: cannot assign to non-static data member within const member function 'operation'
        ++counter;
        ^ ~~~~~~~~
ConstructorMutable.cpp:5:10: note: member function 'Buffer::operation' is declared const here
    void operation() const {
    ~~~~~^~~~~~~~~~~~~~~~~~
```

Use of `mutable`

```
class Buffer {
public:
    explicit Buffer(int n);

    void operation() const {
        // record times called
        ++counter;

        // do work ...
    }
private:
    mutable int counter = 0;
};

int main() {

    const Buffer b(500);

    b.operation();

    return 0;
}
```

`mutable` Notes

```
class Buffer {
public:
    explicit Buffer(int n);

    void operation() const {
        // record times called
        ++counter;

        // do work ...
    }
private:
    mutable int counter = 0;
};

int main() {

    const Buffer b(500);

    b.operation();

    return 0;
}
```

- Data members with `mutable` can change and still allow the object to be const

- *bitwise const*

  Not a single bit can change

- *logical const*

  The behavior of the object does not change

  bitwise const implies *logical const*

Constructor Scenario

```cpp
int main() {

    // default (void) constructor A::A()
    A a;

    // copy constructor A::A(const A&)
    A b(a);

    // assignment operator A::operator=(const A&)
    a = b;

    return 0;
}
```

Class I

```cpp
class A {
public:
    void command();
};

int main() {

    // default (void) constructor A::A()
    A a;

    // copy constructor A::A(const A&)
    A b(a);

    // assignment operator A::operator=(const A&)
    a = b;

    return 0;
}
```

- Implicit default constructor: A::A()
- Implicit copy constructor: A::A(const A&)
- Implicit assignment operator: A::operator=(const A&)

Class II

```
class A {
public:
    A(int);
    void command();
};

int main() {

    // default (void) constructor A::A()
    A a;

    // copy constructor A::A(const A&)
    A b(a);

    // assignment operator A::operator=(const A&)
    a = b;

    return 0;
}
```

Class II Error

```
ctor2.cpp:12:7: error: no matching constructor for initialization of 'A'
    A a;
```

Class II

```cpp
class A {
public:
    A(int);
    void command();
};

int main() {

    // default (void) constructor A::A()
    A a;

    // copy constructor A::A(const A&)
    A b(a);

    // assignment operator A::operator=(const A&)
    a = b;

    return 0;
}
```

- Implicit copy constructor: `A::A(const A&)`

- Implicit assignment operator: `A::operator=(const A&)`

- ~~Implicit default constructor:~~ `A::A()`

Class II A ( )

```
class A {
public:
    A();
    A(int);
    void command();
};

int main() {

    // default (void) constructor A::A()
    A a;

    // copy constructor A::A(const A&)
    A b(a);

    // assignment operator A::operator=(const A&)
    a = b;

    return 0;
}
```

Class II `default`

```
class A {
public:
    A() = default;
    A(int);
    void command();
};

int main() {

    // default (void) constructor A::A()
    A a;

    // copy constructor A::A(const A&)
    A b(a);

    // assignment operator A::operator=(const A&)
    a = b;

    return 0;
}
```

`default`

```
class A {
public:
    A() = default;
    A(int);
    void command();
};

int main() {

    // default (void) constructor A::A()
    A a;

    // copy constructor A::A(const A&)
    A b(a);

    // assignment operator A::operator=(const A&)
    a = b;

    return 0;
}
```

- Indicates that the compiler-provided constructor is used, even if it does not appear according to the constructor rules
- Can be applied to other constructors, standard methods, and destructors

POD: Plain Old Data

- A POD class (or struct) is a set of values without any object-oriented features (no vtable, for example)
- All scalar types (e.g., `int`, `double`) are POD
- POD class:

  Has no user-defined copy assignment operator

  Has no user-defined destructor

  Has no non-static data members that are not themselves POD

  Does not use virtual

default vs {}

```cpp
#include <type_traits>

class A {
public:
    A() = default;
};

class B {
public:
    B() {}
};

int main() {
    static_assert(std::is_trivial<A>::value, "A should be trivial");
    static_assert(std::is_pod<A>::value, "A should be POD");

    static_assert(!std::is_trivial<B>::value, "B should not be trivial");
    static_assert(!std::is_pod<B>::value, "B should not be POD");
}
```

Full Application of `default`

```cpp
class A {
public:
    // void constructor
    A() = default;

    // construtor
    A(int size);

    // copy constructor
    A(const A&) = default;

    // move constructor
    A(A&&) = default;

    // assignment
    A& operator=(const A&) = default;

    // move
    A& operator=(A&&) = default;

    // destructor
    ~A() = default;
};
```

```cpp
int main() {

    // default (void) constructor A::A()
    A a;

    // copy constructor A::A(const A&)
    A b(a);

    // assignment operator A::operator=(const A&)
    a = b;

    // move assignment operator A::operator=(&&A)
    a = std::move(b);

    return 0;
}
```

POD with `default`

```cpp
#include <type_traits>

class A {
public:
    // void constructor
    A() = default;

    // construtor
    A(int size);

    // copy constructor
    A(const A&) = default;

    // move constructor
    A(A&&) = default;

    // assignment
    A& operator=(const A&) = default;

    // move
    A& operator=(A&&) = default;

    // destructor
    ~A() = default;
};
```

```cpp
int main() {
    static_assert(std::is_trivial<A>::value, "A should be trivial");
    static_assert(std::is_pod<A>::value, "A should be POD");

    return 0;
}
```

delete

```
class A {
public:
    A() = delete;
    A(const A&) = delete;
    A(A&&) = delete;
    A& operator=(const A&) = delete;
    A& operator=(A&&) = delete;
    virtual ~A() = delete;
    A(int);
    void command();
};

int main() {

    // default (void) constructor A::A()
//    A a;

    // copy constructor A::A(const A&)
//    A b(a);

    // assignment operator A::operator=(const A&)
//    a = b;

    A a(1);

    return 0;
}
```

Error

```
DefaultCtor6.cpp:24:7: error: attempt to use a deleted function
    A a(1);
      ^
DefaultCtor6.cpp:8:13: note: '~A' has been explicitly marked deleted here
    virtual ~A() = delete;
            ^
```

delete

```
class A {
public:
    A() = delete;
    A(const A&) = delete;
    A(A&&) = delete;
    A& operator=(const A&) = delete;
    A& operator=(A&&) = delete;
    virtual ~A() = delete;
    A(int);
    void command();
};

int main() {

    // default (void) constructor A::A()
//    A a;

    // copy constructor A::A(const A&)
//    A b(a);

    // assignment operator A::operator=(const A&)
//    a = b;

    A a(1);

    return 0;
}
```

```
class A {
public:
    A() = delete;
    A(const A&) = delete;
    A(A&&) = delete;
    A& operator=(const A&) = delete;
    A& operator=(A&&) = delete;
//    virtual ~A() = delete;
    A(int);
    void command();
};

int main() {

    // default (void) constructor A::A()
//    A a;

    // copy constructor A::A(const A&)
//    A b(a);

    // assignment operator A::operator=(const A&)
//    a = b;

    A a(1);

    return 0;
}
```