TDDD38/726G82 -Advanced programming in C++ Class design Christoffer Holm

Department of Computer and information science



- 2 Classes
- 3 Lifetime Management
- 4 Operator Overloading
- 5 Aggregates



- 2 Classes
- 3 Lifetime Management
- 4 Operator Overloading
- 5 Aggregates



const

```
int x { 5 };
int const y { 7 };
int const* v { &x };
int* const w { &x };
x = 7; // allowed
y = 5; // not allowed
v = &y; // allowed
w = &y; // not allowed
*v = 8; // not allowed
*w = 10; // allowed
```



const

- A variable declared const cannot be modified after initialization
- A pointer to a const object can be modified, but it cannot modify the underlying object
- A const pointer cannot change what they point to
- A non-const object can be converted to a const version, but **not** vice versa.



const

Rule of thumb: const applies to the left:

int const * const



const

Rule of thumb: const applies to the left:

int const $\frac{*}{4}$ const



const

Rule of thumb: const applies to the left:

<u>int</u> const * const



const

... Except when it's at the start:

const int



const

... Except when it's at the start:





Value categories & References

- Т&
- T const&
- T&&
- T const&&



Value categories & References

- T&
 - Called Ivalue-reference;
 - Used to alias existing object;
 - Can only bind to *lvalues*.
- T const&
- T&&
- T const&&



Value categories & References

- T&
- T const&
 - Called const lvalue-reference;
 - Can bind to all const objects;
 - can bind to all non-const objects.
- T&&
- T const&&



Value categories & References

- T&
- T const&
- T&&
 - Called rvalue-reference;
 - Used to extend the lifetime of temporary objects;
 - Binds to all rvalues turning them into xvalue.
- T const&&



Value categories & References

- T&
- T const&
- T&&
- T const&&
 - Called const rvalue-reference;
 - Is a weaker version of const lvalue-reference;
 - can only bind to rvalues that are const.



What will happen? Why?

```
void fun(int const&) { cout << 1; }
void fun(int&) { cout << 2; }
void fun(int&&) { cout << 3; }
int main()
{
    int a;
    int const c{};
    fun(23);
    fun(a);
    fun(c);
}</pre>
```



2 Classes

- 3 Lifetime Management
- 4 Operator Overloading
- 5 Aggregates



The Anatomy of a Class Declaration

- Declared with either class or struct;
- Has data members;
- Has member functions;
- Each member has an *access level*.



The Anatomy of a Class Declaration

Declared with either class or struct;

class My_Class
{
};

```
struct My_Struct
{
```

};

- Has data members;
- Has member functions;
- Each member has an access level.



The Anatomy of a Class Declaration

- Declared with either class or struct;
 - class and struct only have minor differences;
 - All members in a class are by default private;
 - All members in a struct are by default *public*;
 - Inheritance has respective access level.
- Has data members;
- Has member functions;
- Each member has an *access level*.



The Anatomy of a Class Declaration

- Declared with either class or struct;
- Has data members;

```
class Cls
{
   int number;
   std::string text;
};
```

- Has member functions;
- Each member has an access level.



The Anatomy of a Class Declaration

- Declared with either class or struct;
- Has data members;
- Has member functions;

```
class Cls
{
    void foo(int);
    void foo(double);
    void foo();
};
```

• Each member has an *access level*.



The Anatomy of a Class Declaration

- Declared with either class or struct;
- Has data members;
- Has member functions;
- Each member has an *access level*.

```
class Cls
{
public:
    void foo(int);
private:
    int number;
};
```



Class Scope

- Each class defines its own *scope*;
- All members belong to said scope;
- The name of the members can be access with the *scope resolution operator* : :



Class Scope

```
// class declaration
class Cls;
// class definition
class Cls
{
public:
  // member function declaration
  void foo();
};
// member function definition
void Cls::foo() { cout << "foo" << endl; }</pre>
```



The Object Model

- Each class in C++ defines a type;
- Values/expressions with this type are called *objects*;
- Creating an object of a class type is called *instantiation*.



The Object Model

```
class Cls
{
public:
   void set(int n) {
     num = n;
   }
   int get() {
     return num;
   }
private:
   int num;
};
```

```
int main()
{
  Cls o1;
  Cls o2;
  o1.set(1);
  o2.set(2);
  cout << o1.get() << ' '</pre>
        << o2.get()
        << endl;
}
```



The Object Model

```
class Cls
{
public:
   void set(int n) {
     this->num = n;
   }
   int get() {
     return this->num;
   }
private:
   int num;
};
```

```
int main()
{
  Cls o1;
  Cls o2;
  o1.set(1);
  o2.set(2);
  cout << o1.get() << ' '
       << o2.get()
       << endl;
}
```



Mental Model

```
class Cls
{
public:
  void set(int n);
private:
  int num;
};
int main()
{
  Cls obj;
  obj.set(5);
}
```



Mental Model

```
class Cls
{
public:
  void set(int n);
private:
  int num;
};
int main()
{
  Cls obj;
  obj.set(5);
}
```

```
struct Cls
{
  int num;
};
void set(Cls* this,
          int n);
int main()
{
  Cls obj;
  set(&obj, 5);
}
```



Constant Member Functions

```
class Cls
{
public:
   void fun() const;
private:
   int data;
};
void Cls::fun() const
{
   // not allowed
   data = 5;
}
```



Constant Member Functions & Mental Model

```
class Cls
{
public:
   void fun() const;
private:
   int data;
};
void Cls::fun() const
{
   // not allowed
   data = 5;
}
```

```
struct Cls
{
  int data;
};
void fun(Cls const* this)
{
  // not allowed
  this->data = 5;
}
```



Ref-qualifiers

```
class Cls
{
public:
   void fun() &;
   void fun() &&;
   void fun() const&;
};
```

- indicate what type of object this is;
- pointers can only point to glvalues;
- mental model breaks down.



Ref-qualifiers

```
class Cls
{
public:
   void fun() &;
   void fun() &&;
   void fun() const&;
};
```



Ref-qualifiers

```
class Cls
{
public:
   void fun() &;
   void fun() &&;
   void fun() const&;
};
```

Cls c1{}; c1.fun(); Cls{}.fun(); Cls const c2{}; c2.fun();



2 Classes

3 Lifetime Management

- 4 Operator Overloading
- 5 Aggregates



Constructors

```
class Cls
{
public:
   Cls(int a) : val1{a}, val3{2}
   {
    // can execute code here as well
   }
private:
   int val1;
   int val2 {2+3};
   int val3 {4};
};
```



Constructors

```
int main()
{
    Cls obj1{5};
    Cls obj2(5);
    Cls* ptr{new Cls{5}};
    Cls(5); // prvalue
}
```



Constructors

- Avoid initializing members in the body of the constructor;
- const-members must be initialized in the member-initializer-list;
- Initializing in the body is an *assignment*.



```
class Cls
{
public:
   Cls(int x = 0) : data{new int{x}} { }
   ~Cls()
   {
    delete data;
   }
private:
   int* data;
};
```



```
Cls global{0}; // static storage
void fun()
{
    static Cls other{1}; // static storage
    Cls cls{2};
}
int main()
{
    Cls c{3};
    fun();
    c.~Cls(); // don't do this
}
```



- Objects that have *static storage* are destroyed at the end of the program.
- Global variables are created at the start of the program,
- Static variables in functions are constructed the first time that function is called and will persist between all future calls.



- Even though destructors can be called explicitly it should be avoided:
- Once the lifetime ends the destructor will be called automatically by the compiler;
- Meaning, if you have called it yourself before that point the destructor will be called twice which will (in most cases) cause issues.



```
class Cls
{
public:
   Cls(); // default constructor
   Cls(Cls const&); // copy constructor
   Cls(Cls&&); // move constructor
   ~Cls(); // destructor
   Cls& operator=(Cls const&); // copy assignment
   Cls& operator=(Cls&&); // move assignment
};
```



Special Member Functions

The compiler can generate these functions, unless:

- a constructor declared; no default constructor
- copy operations declared; no move operations
- move operations declared; no copy operations



Special Member Functions

The compiler can generate these functions, unless:

- a constructor declared; no default constructor
- copy operations declared; no move operations
- move operations declared; no copy operations
- Possible to bypass these rules with =default and =delete.



- rule of three
- rule of five
- rule of zero



- rule of three
 - Before C++11 (Note this concept is not valid in C++11 or later);
 - If a class require a destructor or copy operation;
 - it should (probably) implement the destructor, copy constructor and copy assignment.
- rule of five
- rule of zero



- rule of three
- rule of five
 - C++11 and onwards;
 - If a class requires a destructor, copy or move operations;
 - it should implement a destructor, copy operations and move operations.
- rule of zero



- rule of three
- rule of five
- rule of zero
 - If all resources used in the class take care of their own data;
 - the class should *not* have to implement any destructor, copy or move operations.



```
class Cls
{
public:
    Cls(int); // remove default ctor
    Cls() = default; // generate it anyway
    Cls(Cls const&) = delete;// remove copy ctor
    Cls(Cls&&) = default; // generate move ctor
};
```



```
Cls identity(Cls obj)
{
    return obj;
}
int main()
{
    Cls obj1{};
    Cls obj2 = Cls{};
    obj1 = identity(obj1);
    obj1 = obj2;
}
```



```
Cls identity(Cls obj)
{
   return obj;
}
int main()
{
   Cls obj1{;
   Cls obj2 = Cls{};
   obj1 = identity(obj1);
   obj1 = obj2;
}
```



```
Cls identity(Cls obj)
{
   return obj;
}
int main()
{
   Cls obj1{;
   Cls obj2 = Cls{};
   obj1 = identity(obj1);
   obj1 = obj2;
}
```

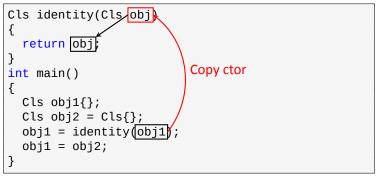


```
Cls identity(Cls obj)
{
   return obj;
}
int main()
{
   Cls obj1{};
   Cls obj2 = Cls{};
   obj1 = identity(obj1);
   obj1 = obj2;
}
```

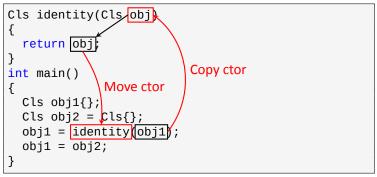


```
Cls identity(Cls obj)
{
   return obj;
}
int main()
{
   Cls obj1{};
   Cls obj2 = Cls{};
   obj1 = identity(obj1);
   obj1 = obj2;
}
```

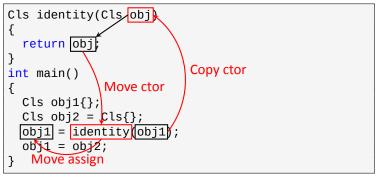




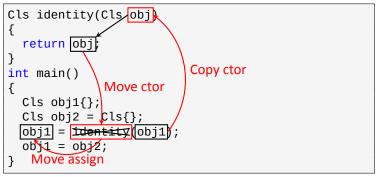




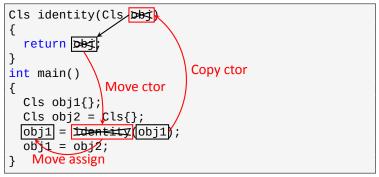














```
Cls identity(Cls D+b;)
{
    return D+b;;
}
int main()
{
    Cls obj1{};
    Cls obj2 = Cls{};
    obj1 = identity(obj1);
    obj1 = [obj2];
}
```



```
Cls identity(Cls Debg)
{
    return Debg;
}
int main()
{
    Cls obj1{};
    Cls obj2 = Cls{};
    obj1 = identity(obj1);
    Obj1 = obj2;
} Copy assign
```



```
Cls identity(Cls D+b;)
{
    return D+b;;
}
int main()
{
    Cls obj1{};
    Cls obj2 = Cls{};
    obj1 = Identity(obj1);
    obj1 = obj2;
}
```



```
Cls identity(Cls D+5;)
{
    return D+5;;
}
int main()
{
    Cls obj1{};
    Cls D+52 = Cls{};
    obj1 = Identity(obj1);
    obj1 = D+52;
}
```



```
Cls identity(Cls D+5;)
{
    return D+5;;
}
int main()
{
    Cls D+5;1{;
    Cls D+5;2 = Cls{};
    D+5;1 = identity(D+5;1);
    D+5;1 = D+5;2;
}
```



```
Cls identity(Cls D+5;)
{
    return D+5;;
}
int main()
{
    Cls D+5;1{;
    Cls D+5;2 = Cls{};
    D+5;1 = identity(D+5;1);
    D+5;1 = D+5;2;
}
```



As if rule

- The compiler is allowed to modify the code however it want;
- As long as the *observable behaviour* is exactly the same.



As if rule

- The compiler is allowed to modify the code however it want;
- As long as the *observable behaviour* is exactly the same.
- Copy elision is an exception to the as if rule;
- it allows the compiler to remove calls to copy or move constructors.



Copy elision

```
int main()
{
    Cls t1{};
    Cls t2{t1};
    Cls t3{Cls{}};
}
```



What will happen? Why?

```
struct Cls
{
  Cls() = default;
 Cls(Cls const&) { cout << "C"; }
  Cls(Cls&&) { cout << "M"; }
 ~Cls() = default;
};
Cls ident(Cls c)
{
  return c;
}
int main()
{
  Cls c1{Cls{}};
  Cls c2{ident(c1)};
  Cls c3{c2};
}
```



1 References & const

- 2 Classes
- 3 Lifetime Management
- 4 Operator Overloading
- 5 Aggregates



Operators

- Most operators can be overloaded;
- the exceptions are . . * :: ?:



- Given any binary operator @;
- x@y becomes x.operator@(y) or operator@(x,y).



- Given any binary operator @;
- x@y becomes x.operator@(y) or operator@(x,y).
- Example:

```
struct Cls
{
   Cls operator+(Cls b);
};
int main()
{
   Cls a, b;
   Cls c{a+b};
}
```



- Given any binary operator @;
- x@y becomes x.operator@(y) or operator@(x,y).
- Example:

```
struct Cls
{
    Cls operator+(Cls b);
};
int main()
{
    Cls a, b;
    Cls c{a.operator+(b)};
}
```



- Given any binary operator @;
- x@y becomes x.operator@(y) or operator@(x,y).
- Example:

```
struct Cls
{
};
Cls operator+(Cls a, Cls b);
int main()
{
    Cls a, b;
    Cls c{a+b};
}
```



- Given any binary operator @;
- x@y becomes x.operator@(y) or operator@(x,y).
- Example:

```
struct Cls
{
};
Cls operator+(Cls a, Cls b);
int main()
{
    Cls a, b;
    Cls c{operator+(a, b)};
}
```



Rule of thumb

- Do I need this operator?
- What is the operators behaviour?



Rule of thumb

- Do I need this operator? The operator should make sense.
- What is the operators behaviour?



Rule of thumb

- Do I need this operator? The operator should make sense.
- What is the operators behaviour? Should be similar to the built in types.



Type conversions

```
class Cls
{
public:
   Cls(int i) : i{i} { }
   operator int() const
   {
     return i;
   }
private:
   int i;
};
```



Type conversions

- A constructor that can take **one** argument is called a *type converting constructor*;
- these constructors can be used by the compiler to perform conversions.
- The special operator Cls::operator TYPE() is called whenever the class Cls is converted to TYPE;
- the compiler is allowed to use this operator to perform implicit type conversions;
- but can also be explicitly called through casting.



Explicit keyword

```
class Cls
{
public:
    explicit Cls(int i) : i{i} { }
    explicit operator int() const
    {
       return i;
    }
private:
    int i;
};
```



Explicit keyword

- Declaring type converting constructors or operators as explicit means;
- the compiler is **not** allowed to use these functions for implicit type conversion;
- with the exception of operator bool which can be used for *contextual conversion*.



Contextual Conversion

```
struct Cls
{
  explicit operator bool() const { return flag; }
  bool flag{};
};
int main()
{
  Cls c{};
  if (c)
    // ...
```





- 5 Aggregates

Aggregates

What is an Aggregate?

An *aggregate* denotes a simple kind of data type with the following properties;

- An array- or class type;
- no user-provided constructors;
- no private or static data members;
- no virtual functions;
- no private base classes.



Aggregates

Basic Aggregate

```
struct Person
ł
  string name{"unknown"};
  int age{};
};
int main()
{
  Person bob{"Bob", 37};
  Person robin{"Robin"};
  Person unknown{};
  Person sara{.name = "Sara", .age = 29};// C++20
```





