

TDDD38/726G82 - Advanced programming in C++

Inheritance & Polymorphism

Christoffer Holm

Department of Computer and information science

- 1 Inheritance
- 2 Polymorphism
- 3 Exception Handling
- 4 Smart Pointers

- 1 Inheritance
- 2 Polymorphism
- 3 Exception Handling
- 4 Smart Pointers

Inheritance

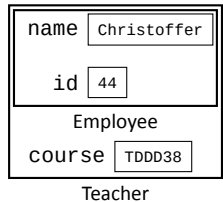
Mental Model

```
class Employee
{
    string name{"Christoffer"};
    int id{44};
};
class Teacher : public Employee
{
    string course{"TDDD38"};
};
Teacher c{};
```

Inheritance

Mental Model

```
class Employee
{
    string name{"Christoffer"};
    int id{44};
};
class Teacher : public Employee
{
    string course{"TDDD38"};
};
Teacher c{};
```



Inheritance

Protected members

```
class Base
{
public:
    Base(int x)
        : x{x} { }

private:
    int x;
};
```

```
struct Derived : Base
{
    Derived(int x)
        : Base{x} { }
    int get()
    {
        return x; // Error!
    }
};
```

Inheritance

Protected members

```
class Base
{
public:
    Base(int x)
        : x{x} { }

protected:
    int x;
};
```

```
struct Derived : Base
{
    Derived(int x)
        : Base{x} { }
    int get()
    {
        return x; // OK!
    }
};
```

Inheritance

Protected members

`protected` members are:

- inaccessible outside the class;
- accessible within derived classes;
- accessible by friends of the class.

Inheritance

Constructors

```
class Base
{
public:
    Base(int x);
private:
    int x;
};

Base::Base(int x)
    : x{x}
{
}
```

```
class Derived : public Base
{
public:
    Derived(int x, double y);
private:
    double y;
};

Derived::Derived(int x, double y)
    : Base{x}, y{y}
{
}
```

Inheritance

Initialization & Destruction

```
class Base
{
    int x{1};
};
class Derived1 : public Base
{
    double y{2.34};
};
class Derived11 final
    : public Derived1
{
    int z{56};
};
```

Derived11 obj{};

Inheritance

Initialization & Destruction

```
class Base
{
    int x{1};
};
class Derived1 : public Base
{
    double y{2.34};
};
class Derived11 final
    : public Derived1
{
    int z{56};
};
```

Derived11 obj{};



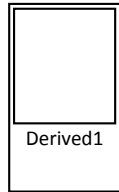
Derived11

Inheritance

Initialization & Destruction

```
class Base
{
    int x{1};
};
class Derived1 : public Base
{
    double y{2.34};
};
class Derived11 final
    : public Derived1
{
    int z{56};
};
```

Derived11 obj{};



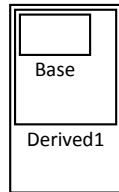
Derived11

Inheritance

Initialization & Destruction

```
class Base
{
    int x{1};
};
class Derived1 : public Base
{
    double y{2.34};
};
class Derived11 final
    : public Derived1
{
    int z{56};
};
```

Derived11 obj{};



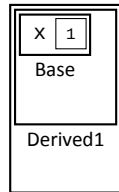
Derived11

Inheritance

Initialization & Destruction

```
class Base
{
    int x{1};
};
class Derived1 : public Base
{
    double y{2.34};
};
class Derived11 final
    : public Derived1
{
    int z{56};
};
```

Derived11 obj{};



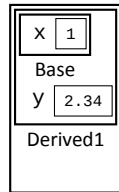
Derived11

Inheritance

Initialization & Destruction

```
class Base
{
    int x{1};
};
class Derived1 : public Base
{
    double y{2.34};
};
class Derived11 final
    : public Derived1
{
    int z{56};
};
```

Derived11 obj{};



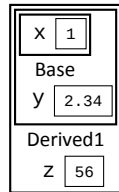
Derived11

Inheritance

Initialization & Destruction

```
class Base
{
    int x{1};
};
class Derived1 : public Base
{
    double y{2.34};
};
class Derived11 final
    : public Derived1
{
    int z{56};
};
```

Derived11 obj{};



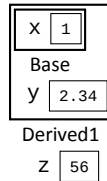
Derived11

Inheritance

Initialization & Destruction

```
class Base
{
    int x{1};
};
class Derived1 : public Base
{
    double y{2.34};
};
class Derived11 final
    : public Derived1
{
    int z{56};
};
```

Derived11 obj{};

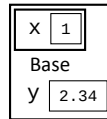


Inheritance

Initialization & Destruction

```
class Base
{
    int x{1};
};
class Derived1 : public Base
{
    double y{2.34};
};
class Derived11 final
    : public Derived1
{
    int z{56};
};
```

Derived11 obj{};



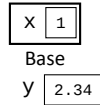
Derived1

Inheritance

Initialization & Destruction

```
class Base
{
    int x{1};
};
class Derived1 : public Base
{
    double y{2.34};
};
class Derived11 final
    : public Derived1
{
    int z{56};
};
```

Derived11 obj{};



Inheritance

Initialization & Destruction

```
class Base
{
    int x{1};
};
class Derived1 : public Base
{
    double y{2.34};
};
class Derived11 final
    : public Derived1
{
    int z{56};
};
```

Derived11 obj{};



Base

Inheritance

Initialization & Destruction

```
class Base
{
    int x{1};
};
class Derived1 : public Base
{
    double y{2.34};
};
class Derived11 final
    : public Derived1
{
    int z{56};
};
```

Derived11 obj{};

x 1

Inheritance

Initialization & Destruction

```
class Base
{
    int x{1};
};
class Derived1 : public Base
{
    double y{2.34};
};
class Derived11 final
    : public Derived1
{
    int z{56};
};
```

Derived11 obj{};

Inheritance

Initialization & Destruction

An object is initialized in the following order:

1. initialize base classes (call constructors);
2. initialize all data members in declaration order.

An object is destroyed in the following order:

1. destroy all data members in reverse order;
2. destroy base classes in reverse order.

Inheritance

Types of Inheritance

- `public` inheritance
- `protected` inheritance
- `private` inheritance

Inheritance

Types of Inheritance

- `public` inheritance
 - `class Derived : public Base`
 - All public and protected members of Base are available as public and protected respectively in Derived.
- `protected` inheritance
- `private` inheritance

Inheritance

Types of Inheritance

- `public` inheritance
- `protected` inheritance
 - `class` Derived : `protected` Base
 - All public and protected members of Base are available as protected in Derived.
- `private` inheritance

Inheritance

Types of Inheritance

- `public` inheritance
- `protected` inheritance
- `private` inheritance
 - `class Derived : private Base`
 - All members of Base are inherited as private and therefore inaccessible from Derived.

Inheritance

What will happen? Why?

```
struct Base
{
    ~Base() { cout << "Base" << endl; }
};
struct Derived : public Base
{
    ~Derived() { cout << "Derived" << endl; }
};
int main()
{
    Derived d{};
}
```

- 1 Inheritance
- 2 **Polymorphism**
- 3 Exception Handling
- 4 Smart Pointers

Polymorphism

Dynamic dispatch

```
void print1()
{ cout << "1" << endl; }

struct Base
{
    Base() = default;
    void print()
    {
        foo();
    }

protected:
    using function_t = void (*);

    Base(function_t foo)
        : foo{foo} { }

private:
    function_t foo{print1};
};
```

```
void print2()
{ cout << "2" << endl; }

struct Derived : public Base
{
    // inherit constructors from Base
    using Base::Base;
    // override default constructor
    Derived()
        : Derived{print2} { }
};

int main()
{
    Base* bp {new Base{}};
    bp->print();
    delete bp;

    bp = new Derived{};
    bp->print();
}
```

Polymorphism

Easier dynamic dispatch

```
struct Base
{
    virtual void print()
    {
        cout << "1" << endl;
    }
};

struct Derived : public Base
{
    void print() override
    {
        cout << "2" << endl;
    }
};
```

```
int main()
{
    Base* bp {new Base{}};
    bp->print();
    delete bp;

    bp = new Derived{};
    bp->print();
}
```

Polymorphism

What will happen? Why?

```
struct Base
{
    ~Base() { cout << "Base" << endl; }
};
struct Derived : public Base
{
    ~Derived() { cout << "Derived" << endl; }
};
int main()
{
    Base* bp{new Derived()};
    delete bp;
}
```


Polymorphism

What will happen? Why?

```
struct Base
{
    virtual ~Base() { cout << "Base" << endl; }
};
struct Derived : public Base
{
    ~Derived() { cout << "Derived" << endl; }
};
int main()
{
    Base* bp{new Derived()};
    delete bp;
}
```

Polymorphism

Virtual destructor

- bp is of type Base* (the *static type* of bp);
- deleting bp will call the destructor of Base regardless of what the *dynamic type* of bp is;
- However, if the destructor of base is **virtual** the compiler will use dynamic dispatch to call the overridden destructor from Derived, which in turn will call the Base destructor.

Therefore we should always declare destructors as virtual for types which will be used through pointers.

Polymorphism

Virtual Table

```
struct Base
{
    virtual ~Base();
    virtual void fun();
    int val1{1};
    int val2{2};
};
struct Derived1 : public Base
{
    void fun() override;
    double d{3.4};
};
struct Derived11 : public Derived1
{
    void fun() final;
};
```

```
void Base::fun()
{
    cout << val1 << ' ' << val2;
}

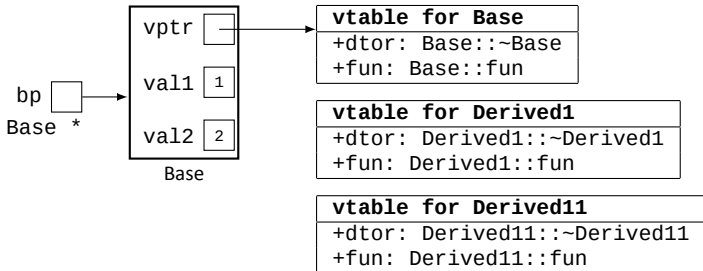
void Derived1::fun()
{
    Base::fun();
    cout << ' ' << d;
}

void Derived11::fun()
{
    cout << "Derived11 ";
    Derived1::fun();
}
```

Polymorphism

Virtual Table

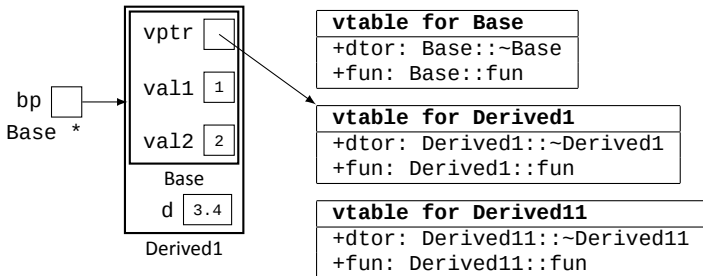
```
Base* bp{new Base{}};
```



Polymorphism

Virtual Table

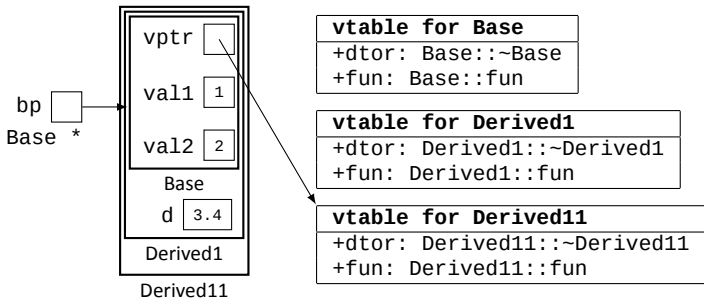
```
Base* bp{new Derived1{}};
```



Polymorphism

Virtual Table

```
Base* bp{new Derived1{}};
```



Polymorphism

Run-time type information (RTTI)

- Each entry in the vtable contains information about the dynamic type;
- This data is accessible with `typeid`.

```
struct Base { virtual ~Base() = default; };
struct Derived1 : public Base { };
struct Derived11 : public Derived1 { };
int main()
{
    Base b;
    Derived1 d1, d2;
    Derived11 d11;
    cout << typeid(b).name() << endl;
    cout << typeid(d1).hash_code() << endl;
    cout << (typeid(d1) == typeid(b)) << endl;
    cout << (typeid(d1) == typeid(d2)) << endl;
    cout << (typeid(d1) == typeid(d11)) << endl;
}
```

Polymorphism

Run-time type information (RTTI)

- `typeid` is used to check the *exact* dynamic type;
- We can use `dynamic_cast` to cast pointers or references to objects into some pointer or reference which is compatible with the dynamic type of the object.

```
struct Base { virtual ~Base() = default; };
struct Derived1 : public Base { };
struct Derived11 : public Derived1 { };
int main()
{
    Base* bp{new Derived1()};
    cout << (dynamic_cast<Base*>(bp) == nullptr) << endl;
    cout << (dynamic_cast<Derived11*>(bp) == nullptr) << endl;
}
```


Polymorphism

Run-time type information (RTTI)

```
struct Base
{
    virtual ~Base() = default;
};
struct Derived1 : public Base
{
    int foo() { return 1; }
};
struct Derived11 : public Derived1 { };
int main()
{
    Base* bp{new Derived1()};
    // won't work, since foo is a non-virtual function in Derived
    cout << bp->foo() << endl;
    // will work, since we converted bp to Derived* which has access to foo
    cout << dynamic_cast<Derived1*>(*bp).foo() << endl;
    // will throw an exception of type std::bad_cast
    cout << dynamic_cast<Derived11*>(*bp).foo() << endl;
}
```

Polymorphism

What will happen? Why?

```
struct Base { virtual ~Base() = default; };
struct Derived1 : public Base { };
struct Derived11 : public Derived1 { };
struct Derived2 : public Base { };
int main()
{
    Base* bp{new Derived1()};
    if (dynamic_cast<Base*>(bp))
        cout << "B ";
    if (dynamic_cast<Derived1*>(bp))
        cout << "D1 ";
    if (dynamic_cast<Derived11*>(bp))
        cout << "D11 ";
    if (dynamic_cast<Derived2*>(bp))
        cout << "D2 ";
}
```

Polymorphism

Slicing

```
struct Base
{
    virtual void print() {cout << x;}
    int x{1};
};
struct Derived : public Base
{
    void print() override {cout << y;}
    int y{2};
};
```

```
void print(Base b)
{
    b.print();
}

int main()
{
    Derived d{};
    print(d);
}
```

- Copying d into b will cause *slicing*;
- Will only copy the Base part of d and thus lose all information about d being a Derived.
- Always use references or pointers!

- 1 Inheritance
- 2 Polymorphism
- 3 Exception Handling**
- 4 Smart Pointers

Exception Handling

Model

```
int main()
{
    try
    {
        fun1();
        // ...
    }
    catch (std::exception& e)
    {
        cerr << e.what();
    }
}
```

```
void fun1()
{
    // ...
    fun2();
    // ...
    return;
}
```

```
void fun2()
{
    return;
}
```

Exception Handling

Model

```
int main()
{
    try
    {
        fun1();
        // ...
    }
    catch (std::exception& e)
    {
        cerr << e.what();
    }
}
```

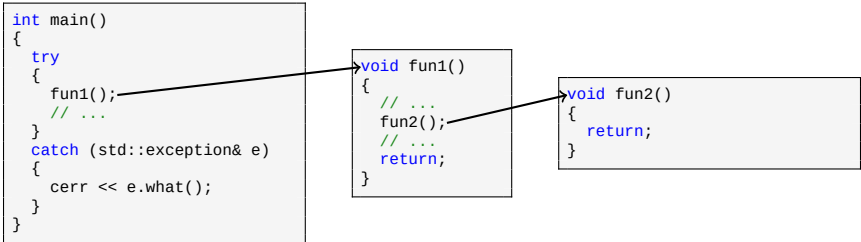
A black arrow originates from the `fun1();` line in the `main()` function and points to the `void fun1()` function definition box.

```
void fun1()
{
    // ...
    fun2();
    // ...
    return;
}
```

```
void fun2()
{
    return;
}
```

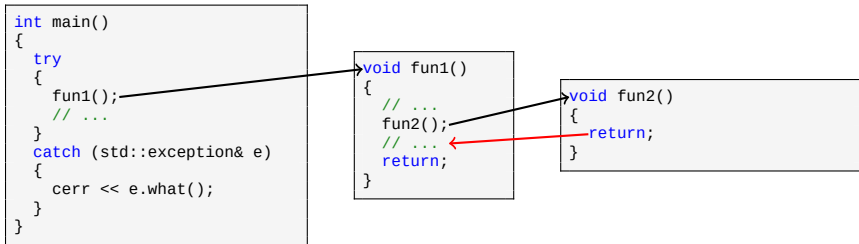
Exception Handling

Model



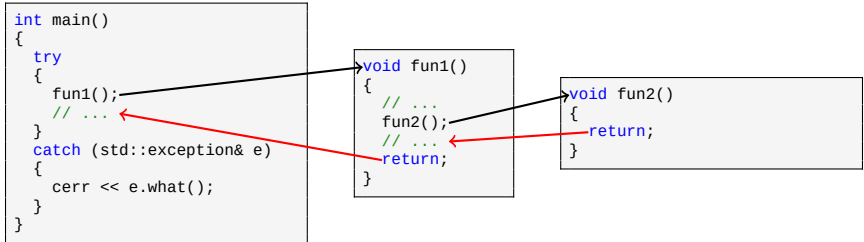
Exception Handling

Model



Exception Handling

Model



Exception Handling

Model

```
int main()
{
    try
    {
        fun1();
        // ...
    }
    catch (std::exception& e)
    {
        cerr << e.what();
    }
}
```

```
void fun1()
{
    // ...
    fun2();
    // ...
    return;
}
```

```
void fun2()
{
    throw std::exception{""};
}
```

Exception Handling

Model

```
int main()
{
    try
    {
        fun1();
        // ...
    }
    catch (std::exception& e)
    {
        cerr << e.what();
    }
}
```

A black arrow originates from the `fun1();` line in the `main()` function and points to the `void fun1()` function definition box.

```
void fun1()
{
    // ...
    fun2();
    // ...
    return;
}
```

```
void fun2()
{
    throw std::exception{""};
}
```

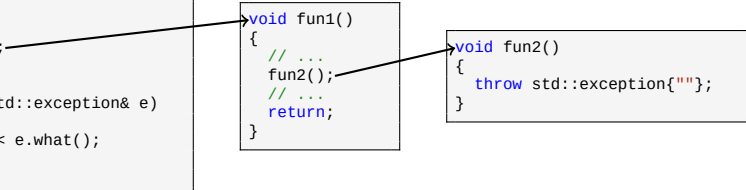
Exception Handling

Model

```
int main()
{
    try
    {
        fun1();
        // ...
    }
    catch (std::exception& e)
    {
        cerr << e.what();
    }
}
```

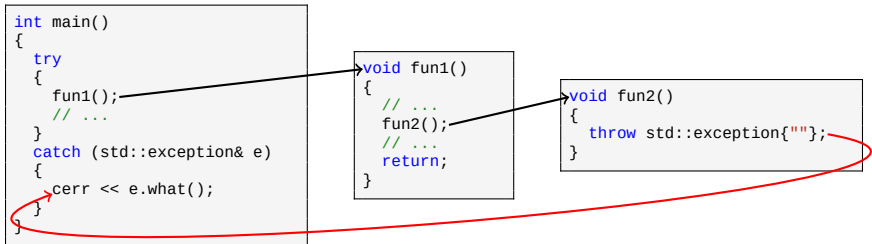
```
void fun1()
{
    // ...
    fun2();
    // ...
    return;
}
```

```
void fun2()
{
    throw std::exception{""};
}
```



Exception Handling

Model



Exception Handling

Exceptions

- Anything can be thrown;
- however the language and the standard library throws objects derived from `std::exception`;
- there are several exception classes defined in `<stdexcept>`;
- always throw by-value: don't throw pointers;
- always catch by-reference to avoid slicing;
- if an exception isn't caught `std::terminate` will be called, thus terminating the program immediately.

Exception Handling

Lifetime & Stack Unwinding

```
int foo()
{
    int z{};
    throw z;
}
struct Cls
{
    Cls() try
        : y{foo()}
    {
    }
    catch (int i)
    {
        cerr << i;
        throw "cls error";
    }
    int y;
};
```

```
int main() try
{
    int x{};
    Cls c{};
    // ...
}
catch (char const* str)
{
    cerr << str;
}
catch (std::exception& e)
{
    cerr << e.what();
}
catch (...)
{
    cerr << "Unknown error";
}
```

Exception Handling

Exception usage

- Exceptions are very slow when they are thrown;
- should only be thrown in *exceptional* situations;
- don't use exceptions for control flow, it will severely slow down your program.

Exception Handling

`noexcept`

- Due to stack unwinding, the compiler have to generate some extra code to handle exceptions;
- this extra generated code can be costly, especially if it is not used;
- the `noexcept`-specifier tells the compiler that no exceptions will be thrown from a function;
- declaring functions as `noexcept` will allow the compiler to not generate code for exception handling.

Exception Handling

noexcept

```
void fun() noexcept;
```

Exception Handling

noexcept

```
void fun() noexcept;
```

- A function declared `noexcept` is allowed to call throwing functions, as long as the exception is caught before it reaches the `noexcept` function;

Exception Handling

`noexcept`

```
void fun() noexcept;
```

- A function declared `noexcept` is allowed to call throwing functions, as long as the exception is caught before it reaches the `noexcept` function;
- If an exception is thrown inside a `noexcept` function, `std::terminate` is called, thus aborting the program.

- 1 Inheritance
- 2 Polymorphism
- 3 Exception Handling
- 4 **Smart Pointers**

Smart Pointers

Exceptions and Memory Management

```
int* get(int x)
{
    if (x < 0)
        throw std::out_of_range{""};
    return new int{x};
}

struct Cls
{
    Cls(int x, int y) : data1{get(x)}, data2{get(y)} { }
    ~Cls()
    {
        delete data1;
        delete data2;
    }
    int* data1;
    int* data2;
};
```

Smart Pointers

Exceptions and Memory Management

```
struct Cls
{
    Cls(int x, int y) try
        : data1{get(x)}, data2{get(y)}
    {
    }
    catch (...)
    {
        delete data1;
        throw;
    }
    ~Cls()
    {
        delete data1;
        delete data2;
    }
    int* data1;
    int* data2;
};
```

Smart Pointers

Exceptions and Memory Management

```
struct Cls
{
    Cls(int x, int y) try
        : data1{get(x)}, data2{get(y)}
    {
    }
    catch (...)
    {
        delete data1;
        throw;
    }
    ~Cls()
    {
        delete data1;
        delete data2;
    }
    int* data1;
    int* data2;
};
```

Segmentation Fault

Smart Pointers

Exceptions and Memory Management

```
struct Cls
{
    Cls(int x, int y) : data1{get(x)}
    {
        try
        {
            data2 = get(y);
        }
        catch (...)
        {
            delete data1;
            throw;
        }
    }
    ~Cls()
    {
        // ...
    }
    // ...
};
```

Smart Pointers

Exceptions and Memory Management

```
struct Cls
{
    Cls(int x, int y) : data1{get(x)}
    {
        try
        {
            data2 = get(y);
        }
        catch (...)
        {
            delete data1;
            throw;
        }
    }
    ~Cls()
    {
        // ...
    }
    // ...
};
```

Painfully Tedious

Smart Pointers

Smart Pointers

- Use RAII to automatically handle memory;
- reside in <memory>;
- `std::unique_ptr`
- `std::shared_ptr`

Smart Pointers

Smart Pointers

- Use RAII to automatically handle memory;
- reside in `<memory>`;
- `std::unique_ptr`
 - Represent ownership;
 - each `unique_ptr` points to a unique object;
 - when the pointer is destroyed, the object is deallocated;
 - cannot be copied, only moved.
- `std::shared_ptr`

Smart Pointers

Smart Pointers

- Use RAII to automatically handle memory;
- reside in `<memory>`;
- `std::unique_ptr`

```
// hand off manually allocated memory
std::unique_ptr<int> ptr1{new int{5}};
// let the smart pointer handle it
std::unique_ptr<int> ptr2{make_unique<int>(5)};
// move ptr2 to ptr3
std::unique_ptr<int> ptr3{std::move(ptr2)};
// ptr2 is now null
```

- `std::shared_ptr`

Smart Pointers

Smart Pointers

- Use RAI to automatically handle memory;
- reside in <memory>;
- `std::unique_ptr`
- `std::shared_ptr`
 - Represent shared ownership on an object;
 - Can be copied;
 - Will deallocate the memory when all shared pointers have been destroyed;
 - Should be avoided if possible since it is quite expensive.

Smart Pointers

Smart Pointers

- Use RAI to automatically handle memory;
- reside in <memory>;
- `std::unique_ptr`
- `std::shared_ptr`

```
std::shared_ptr<int> ptr1{new int{5}};  
std::shared_ptr<int> ptr2{make_shared<int>(5)};  
std::shared_ptr<int> ptr3{ptr2};  
// both ptr2 and ptr3 point to the same object  
// the object will be deallocated once both ptr2 and ptr3  
// have been destroyed.
```

Smart Pointers

Nice solution

```
std::unique_ptr<int> get(int x)
{
    if (x < 0)
        throw std::out_of_range{""};
    return std::make_unique<int>(x);
}
struct Cls
{
    Cls(int x, int y) : data1{get(x)}, data2{get(y)} { }
    ~Cls() = default;
    std::unique_ptr<int> data1;
    std::unique_ptr<int> data2;
};
```


Smart Pointers

Nice solution

```
std::unique_ptr<int> get(int x)
{
    if (x < 0)
        throw std::out_of_range("");
    return std::make_unique<int>(x);
}

struct Cls
{
    Cls(int x, int y) : data1{get(x)}, data2{get(y)} { }
    ~Cls() = default;
    std::unique_ptr<int> data1;
    std::unique_ptr<int> data2;
};
```

www.liu.se