

CS100

Introduction to Programming

Lectures 20: Structuring your code

Outline

- references
- const
- Namespaces
- pre-processor directives
- More on operators
- smart/shared pointers
- friend

Outline

- references
- const
- namespaces
- pre-processor directives
- More on operators
- smart/shared pointers
- friend

Passing by reference

- In C, there have been two ways to pass a variable to a function

- Pass-by-value:

```
void functionX( double val ) {  
    val = ...;  
}
```

- Pass-by-reference:

```
void functionX( double * val ) {  
    *val = ...; //de-referencing  
}
```

Passing by reference

- In C, there have been two ways to pass a variable to a function
 - Drawback of using pointers:
 - Can be uninitialized!
 - Can be invalid

```
double * val = NULL;
```

Passing by reference

- In C++, we can use references
 - Pass-by-value:

```
void functionX( double val ) {  
    val = ...;  
}
```

- Pass-by-reference:

```
void functionX( double & val ) {  
    val = ...; //no de-referencing!  
}
```

Passing by reference

- References are:
 - Valid types, just like pointers
 - Internally: just a pointer
 - Easier to manipulate
 - No de-referencing needed
- Safer
 - can only be initialized from a valid instance of an object

Where do references appear?

- Example 1: Simply in the code

```
int x = 5;
if( ... ) {
    int & y = x; //create reference
    y = 6;
}
//x = 6!
```


Where do references appear?

- Example 2: Through a function call

```
void functionX( double & val ) {  
    val = 6; //no de-referencing!  
}
```

```
int main {  
    int x = 5;  
    functionX(x) ;  
    //x = 6!  
    ...  
}
```

Where do references appear?

- Example 3: As a member of a class?

```
class Car {  
    public:  
        Car() ;  
    private:  
        Driver & driver;  
};
```

```
Car::Car() {};
```

Not permitted! A reference always needs to be initialized from an existing object!

Where do references appear?

- Example 3: What about this?

```
class Car {  
public:  
    Car( Driver & driver );  
private:  
    Driver & m_driver;  
};
```

```
Car::Car( Driver & driver ) {  
    m_driver = driver;  
}
```

- Problem 1: class members need to exist before the body of the constructor so they can be used.
- Problem 2: This syntax would not set m_driver as a reference to driver, but aim at copying the content of driver to where m_driver is referring to

Where do references appear?

- Example 3: Solution for class membership
 - Use an initializer list!

```
class Car {  
public:  
    Car( Driver & driver );  
private:  
    Driver & m_driver;  
};
```

```
Car::Car( Driver & driver ) : m_driver(driver)  
{ }
```

Initializer lists must be used for any members that do not have a default constructor!

Facts about references

- **Driver** & does not have a default constructor
 - → needs to be initialized in initializer list
- Interesting observation:
 - Car also can no longer have simple default constructor!
 - Explains why single constructor overload makes default constructor unavailable
 - default constructor can't work
 - overload becomes “default” constructor

Where do references appear?

- Example 3: And what about this?

```
class Car {  
public:  
    Car();  
    Car( Driver & driver );  
private:  
    Driver & m_driver;  
};
```

```
Car::Car() : m_driver( *(new Driver()) ) {}  
Car::Car( Driver & driver ) : m_driver(driver) {}
```

- Works, but useless!
- Composition! (Car would own driver in this case)
→ no reference needed!

Outline

- references
- **const**
- namespaces
- pre-processor directives
- More on operators
- smart/shared pointers
- friend

Using const

- Let us assume to have a class called **MyVectorA** with a resize function

```
class MyVectorA {  
public:  
    ...  
    void resize( MyVectorB & that );  
};
```

- Goal:
 - resize only needs size of **MyVectorB** **that** to reset its own size.
 - It takes a reference as we would not want to pass a copy of the entire vector

Using const

- Let us assume to have a class called **MyVectorA** with a resize function

```
class MyVectorA {  
public:  
    ...  
    void resize( MyVectorB & that );  
};
```

- Problem:
 - **resize** has a reference to **that**, and can therefore change the original
 - Can we formally prevent this? Can we protect **that** from changes?

Using const

- Solution: Pass as a const reference!

```
class MyVectorA {  
public:  
    ...  
    void resize( const MyVectorB & that );  
};
```

- Now **resize** is no longer allowed to change **that**!

Using const

- Let us furthermore assume that MyVectorB has a function to get the size

```
class MyVectorB {  
public:  
    ...  
    int size() {  
        return m_size;  
    }  
};
```

Using const

- MyVectorA uses this function inside its resize method

```
class MyVectorA {  
public:  
    ...  
    void resize( const MyVectorB & that ) {  
        this->m_data.resize( that.size() );  
    }  
};
```

- Does not work!
- Class methods can change the member variables!
- Calling **size()** is not safe, it may change **that**!

Using const

- Solution:
 - Mark function as constant!

```
class MyVectorB {  
    public:  
        ...  
        int size() {  
            return m_size;  
        }  
};
```

Using const

- Solution:
 - Mark function as constant!

```
class MyVectorB {  
    public:  
        ...  
        int size() const {  
            return m_size;  
        }  
};
```

- Adding **const** behind function means this function is not allowed to change member variables!

Using const

- What about functions that are not allowed to change either member variables or parameters?

```
class MyVectorA {  
public:  
    ...  
    double innerProd( MyVectorA & that ) {  
        //calculate inner product  
        //...  
        return result;  
    }  
};
```

Using const

- What about functions that are not allowed to change either member variables or parameters?

```
class MyVectorA {  
public:  
    ...  
    double innerProd( const MyVectorA & that ) {  
        //calculate inner product  
        //...  
        return result;  
    }  
};
```


Using const

- What about functions that are not allowed to change either member variables or parameters?

```
class MyVectorA {  
public:  
    ...  
    double innerProd( const MyVectorA & that ) const {  
        //calculate inner product  
        //...  
        return result;  
    }  
};
```

Outline

- references
- const
- namespaces
- pre-processor directives
- More on operators
- smart/shared pointers
- friend

The name conflict problem

- Imagine we want to define our own class **vector**

```
#include <vector>
```

```
template<class T>
```

```
class vector {
```

```
public:
```

```
...
```

```
private:
```

```
    std::vector<T> m_data;
```

```
};
```



- STL vector lives in namespace std
- namespace has to be resolved

The name conflict problem

- For convenience, we want to resolve the namespace **std** generally

```
#include <vector>
```

```
using namespace std;
```

```
template<class T>  
class vector {  
public:  
    ...  
private:  
    vector<T> m_data;  
};
```

- Name conflict!
- Compiler can't decide which vector to use
- Compiler won't allow the name **vector** for a class

The name conflict problem

- Other sources of name conflicts:
 - Variable name used more than once

```
#include <stdlib.h>
```

```
int lengthParameter;
```

```
int lengthParameter;
```

```
//...
```

Global namespace

The name conflict problem

- Other sources of name conflicts:
 - Function name/signature used more than once

```
#include <stdlib.h>
```

```
int computeLength();
```

```
int computeLength();
```

```
//...
```

Global namespace

Introducing your own namespace

- Example:

```
Custom namespace 1 {  
    #include <iostream>  
  
    namespace VarSet1 {  
        int lengthParameter;  
        int computeLength();  
    }  
}  
  
Custom namespace 2 {  
    namespace VarSet2 {  
        int lengthParameter;  
        int computeLength();  
    }  
}  
  
int main() {  
    std::cout << VarSet1::lengthParameter << "\n";  
}
```

Introducing your own namespace

- Namespaces for classes
 - A way to create logical grouping

```
namespace MyMathLibrary {  
    template<class T>  
    class Vector {  
        //...  
    };  
  
    template<class T>  
    class Matrix {  
        //...  
    };  
}
```


Introducing your own namespace

- Namespaces for classes
 - A way to create logical grouping
 - Namespace additions can be in different files

File 1

```
namespace MyMathLibrary {  
    template<class T>  
    class Vector {  
        //...  
    };  
}
```

File 2

```
namespace MyMathLibrary {  
    template<class T>  
    class Matrix {  
        //...  
    };  
}
```

- Custom namespaces can be generally included, too:
`using namespace MyMathLibrary;`

Nested namespaces

- Namespaces can be nested
 - Creates a hierarchy of the functionality

```
namespace MyMathLibrary {  
    namespace Linear {  
        //...  
    }  
    namespace Nonlinear {  
        namespace ClosedForm {  
            //...  
        }  
        namespace Iterative {  
            //...  
        }  
    }  
}
```

- Use: `using namespace MyMathLibrary::Nonlinear::Iterative::...`;

Outline

- references
- const
- namespaces
- **pre-processor directives**
- More on operators
- smart/shared pointers
- friend

Preprocessor directives

- Preprocessor directives are lines in the code preceded by a hash-tag (#)
- They are processed by the "pre-processor"
 - Resolved before actual compilation starts
 - They will be replaced by something else
 - They define code to be replaced by something else
- Example: include-guards!

```
#ifndef MYCLASS_HPP_  
#define MYCLASS_HPP_  
//...  
#endif
```

Defines

- `#define`
- Can be used to define constants
 - Example:

```
#define PI 3.1415
```

```
//...
```

```
int main() {
```

```
//...
```

```
    float circum = 2.0f * radius * PI;
```

```
//...
```

```
}
```

Defines

- `#define`
- Does not need a value
 - Can simply mark a variable as defined

```
#define DEBUG_MODE
```

```
//...
```

```
int main() {  
#ifdef DEBUG_MODE  
    if( !(index < vec.size()) )  
        cout << "Access out of bounds\n";  
#endif  
}
```

Defines

- `#define`
- Can be set through the console
 - Example: `g++ -DDEBUG_MODE problem1.cpp -o main`

```
int main() {  
#ifdef DEBUG_MODE  
    if( !(index < vec.size()) )  
        cout << "Access out of bounds\n";  
#endif  
}
```

Defines

- `#define` vs `static const`
 - `#define` consumes no memory
 - `#define` has no type, so can be assigned flexibly
 - `static const` can be scoped
 - `static const` has a single, clearly defined type, which may also be an advantage
 - `static const` enables pointers

Macros

- `#define` can be used to define functions
- Syntax:
`#define fctName(param1,param2) ([fct using params])`
- Example:

```
#define getrandom(min, max) ((rand()%(int)((max) + 1) - (min)) + (min))
```

typedef

- Sometimes we may have long type-names
- Example:

```
std::vector< std::vector< MyMathLibrary::Types::Matrix<double> > >
```

- We can introduce an alias for this type:

```
typedef std::vector< std::vector< MyMathLibrary::Types::Matrix<double> > >  
    MatrixTable;
```

- Can be in global, class, or function scope
- Not really a pre-processor directive anymore

Outline

- references
- const
- namespaces
- pre-processor directives
- **More on operators**
- smart/shared pointers
- friend

What are operators?

+			-	*	/	%
^		&		~	!	&&
		++	--	<<	>>	,
<		<=	==	!=	>	>=
=		+=	-=	*=	/=	%=
&=	=	^=	<<=		>>=	
[]		()	->	new		delete

What are operators?

- Almost all operators in C++ can be overloaded with new meanings
- Operators may not look like functions but can hide function invocations
- You cannot overload the meaning of operators if all arguments are primitive data types, nor can you change the precedence or associativity of operators
- Operators can be defined as either functions or member functions

What are operators?

- Example of an operator (global operator)

```
class Box {  
public:  
    Box (int v) : value(v) {}  
    int value;  
};  
  
// define meaning of comparison for boxes  
bool operator< (const Box & left, const Box & right)  
{  
    return left.value < right.value;  
}
```

- Binary comparison operator!

What are operators?

- Example of a member function

```
class Box {  
public:  
    Box (int v) : value(v) {}  
  
    // define meaning of comparison for boxes  
    bool operator< (const Box & right) {  
        return value < right.value;  
    }  
private:  
    int value;  
};
```

The Increment and Decrement Operators

- ... `operator++`
... `operator--`
- If the increment operator is overloaded, you should define both the prefix and postfix forms (`++it`, `it++`)
- Whenever you have a choice, always invoke the prefix form of the increment operator as it is usually simpler

The Increment and Decrement Operators

```
class Box {  
    public:  
        Box (int v) : value(v) { }  
  
        // prefix versions (++someBox)  
        Box& operator++ () { ++value; return *this; }  
        Box& operator-- () { --value; return *this; }  
  
        Box operator++ (int) { // postfix versions (someBox++)  
            Box result = *this; // step 1, save old value  
            ++value;           // step 2, update value  
            return result;     // step 3, return original  
        }  
        Box operator-- (int) {  
            Box result = *this;  
            --value;  
            return result;  
        }  
    private:  
        int value;  
};
```

The Shift Operators

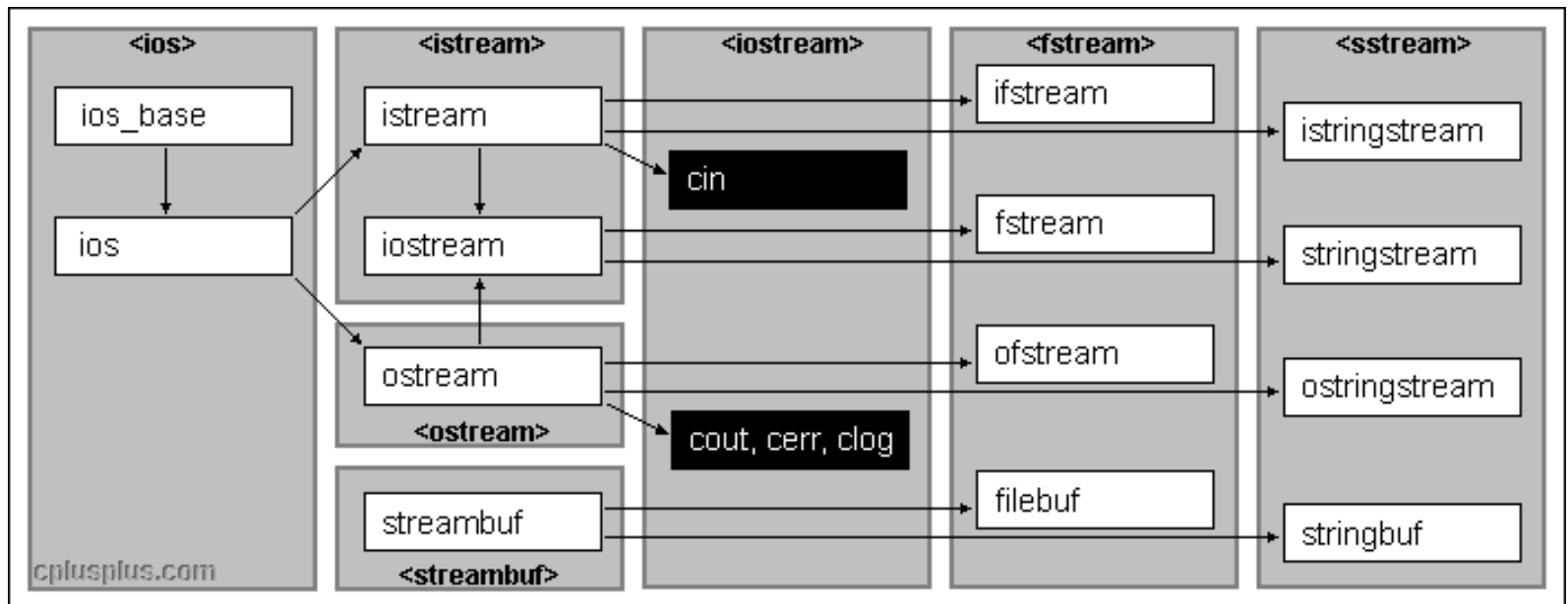
```
cout << "m " << m << " n " << n  
      << " average " << (n+m)/2.0 << '\n';
```

- The shift operators are overloaded in exactly the same fashion as the binary arithmetic operators

```
ostream & operator<<  
    (ostream & out, const RationalNumber & value)  
{  
    // print rational number on an output stream  
    out << value.numerator() << '/' << value.denominator();  
    return out;  
}
```

ios

- Class hierarchy for input/output functionality in STL



The Assignment Operator

- ... **operator=**(...)
- The assignment, comma, and address-of operators will be constructed automatically if the programmer does not specify an alternative.

```
class Box {  
public:  
    Box () { value = 0; }  
    Box (int i) { value = i; }  
    int value;  
};
```

```
Box a(7);  
Box b;  
b = a;
```

The Assignment Operator

- Note 1:
 - Always redefine the assignment operator in classes that include a pointer value
 - Similar to copy constructor: needs deep copy to prevent dangling pointers
 - Example: Box contains vector-pointer `m_ptr`

```
const Box & operator= (Box & left, const Box & right) {  
    if( left.m_ptr != null )  
        delete left.m_ptr;  
    left.m_ptr = new std::vector(*(right.m_ptr));  
    return left;  
}
```

```
Box a, c;  
//... fill box a with elements  
c = a;
```

The Assignment Operator

- Note 2:
 - Despite the use of the assignment symbol, constructors do not use the assignment operator

`Box d = c; // uses copy constructor`

- Note 3:
 - If addition and assignment are both overloaded, then += should be overloaded as well

The Compound Assignment Operators

- Whenever possible, define one operator in terms of another

```
AnObject operator+ ( const AnObject & left, const AnObject & right) {  
    AnObject clone(left); // copy the left argument  
    clone += right;        // combine with right  
    return clone;          // return updated value  
}
```

```
const AnObject & operator+= (AnObject & left, const AnObject & right) {  
    AnObject sum = left + right;  
    left = sum;  
    return left;  
}
```

The Subscript Operator

- Subscript operator is often defined for classes that represent a container abstraction

```
class MyArray {
public:
    MyArray(int s) {
        size = s;
        values = new int[size];
    }
    int & operator[] (unsigned int i) {
        assert(i < size);
        return values[i];
    }
private:
    unsigned int size;
    int * values;
};
```


The Parenthesis Operator

- Function Object: an object that can be used as though it were a function

```
class LargerThan {
public:
    // constructor
    LargerThan (int v) {
        val = v;
    }

    // the function call operator
    bool operator() (int test) {
        return test > val;
    }
private:
    int val;
};
```

The Parenthesis Operator

- Example use:

```
LargerThan tester(12) ;
```

```
int i = ... ;
```

```
if(tester(i)) // true if i is larger than 12
```

- Use inside STL algorithms (for conditional filtering):

```
list<int>::iterator found =  
    find_if( aList.begin(), aList.end(),  
    LargerThan(12) ) ;
```

The Arrow Operator

- Overloading the arrow operator is useful in objects that have a 'pointer-like' behavior
 - Example: iterators, smart pointers
- The arrow operator can only be defined as a member function
- The return type must either be a pointer to a class type or an object for which the member access arrow is defined

The Arrow Operator

- Example:

```
class CountPointer {  
public:  
    CountPointer(Window * w) {  
        count = 0; win = w;  
    }  
    Window * operator->() {  
        count++; return win;  
    }  
private:  
    Window * win;  
    int count;  
};
```

Returns pointer!

```
Window * x = new Window(...); // create the underlying value  
CountPointer p(x);           // create a counting pointer value  
p->setSize(300, 400);         // invoke method in class window
```

Conversion Operators

- For conversions from user types
 - Conversions to user types are defined by constructors

```
operator double( const Rational & val ) {  
    return val.numerator() / (double) val.denominator();  
}
```

```
Rational r(2, 3);
```

```
double d;
```

```
d = 3.14 * double(r); // cast converts fraction to double
```

- In-class definition

```
class Rational {  
    //...  
    operator double() const {  
        return numerator() / (double) denominator();  
    }  
    //...  
};
```

Outline

- references
- const
- Namespaces
- pre-processor directives
- More on operators
- smart/shared pointers
- friend

Raw pointers

- Raw pointers are used to handle memory that was allocated dynamically using **new**
- Raw pointers are unsafe
 - Missing delete causes memory leak

```
if( var == 1 ) {  
    Person* p = new Person("John", 25);  
    p->call();  
    //no assignment, no delete  
}
```

Raw pointers

- Raw pointers are used to handle memory that was allocated dynamically using **new**
- Raw pointers are unsafe
 - Calling delete more than once -> undefined behavior

```
if( var == 1 ) {  
    Person* p = new Person("John", 25);  
    p->call();  
    delete p;  
    delete p;  
}
```


Raw pointers

- Raw pointers are used to handle memory that was allocated dynamically using **new**
- Raw pointers are unsafe
 - Dangling pointers can cause segmentation faults

```
Person* p = new Person[10];  
  
delete[] p;    // p still points to the same memory  
p[0]->call(); // undefined behavior!  
p = NULL;      // ok, now p is not dangling any more
```

Smart pointers

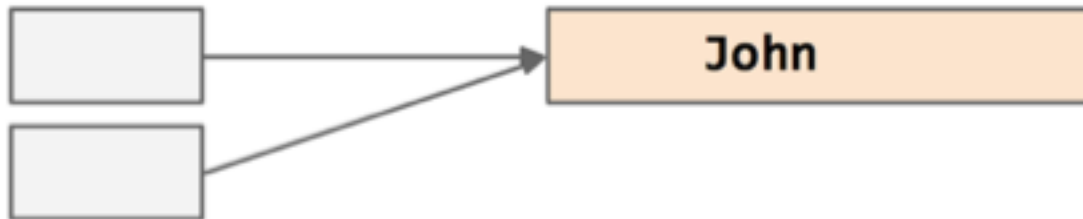
- Smart pointers can avoid these drawbacks
- A smart pointer is a wrapper of a raw pointer providing same functionalities with more safety
 - Same syntax as raw pointers for dereferencing
 - `*p`, `p->val`, and `p[idx]`
 - Automatic management of dynamically allocated memory lifetime
 - Exception-safe destruction
 - Automatically set to NULL, thus avoiding dangling pointers
- RAI – Resource Acquisition Is Initialization
 - A smart pointer takes ownership of the underlying object

Smart pointers

- 4 kinds (all available from `#include <memory>`)
 - **`std::auto_ptr`** (from C++98)
 - First naïve attempt to implement a smart pointer with exclusive ownership.
(Deprecated since C++11, and removed from STL in C++14)
 - **`std::unique_ptr`** (from C++11)
 - Smart pointer used for exclusive ownership that can be copied only with move semantics
 - **`std::shared_ptr`** (from C++11)
 - Smart pointer used for shared ownership with automatic garbage collection based on a reference count
 - **`std::weak_ptr`** (from C++11)
 - Smart pointer used for observing without owning. Similar to **`shared_ptr`**, but does not contribute to the reference count

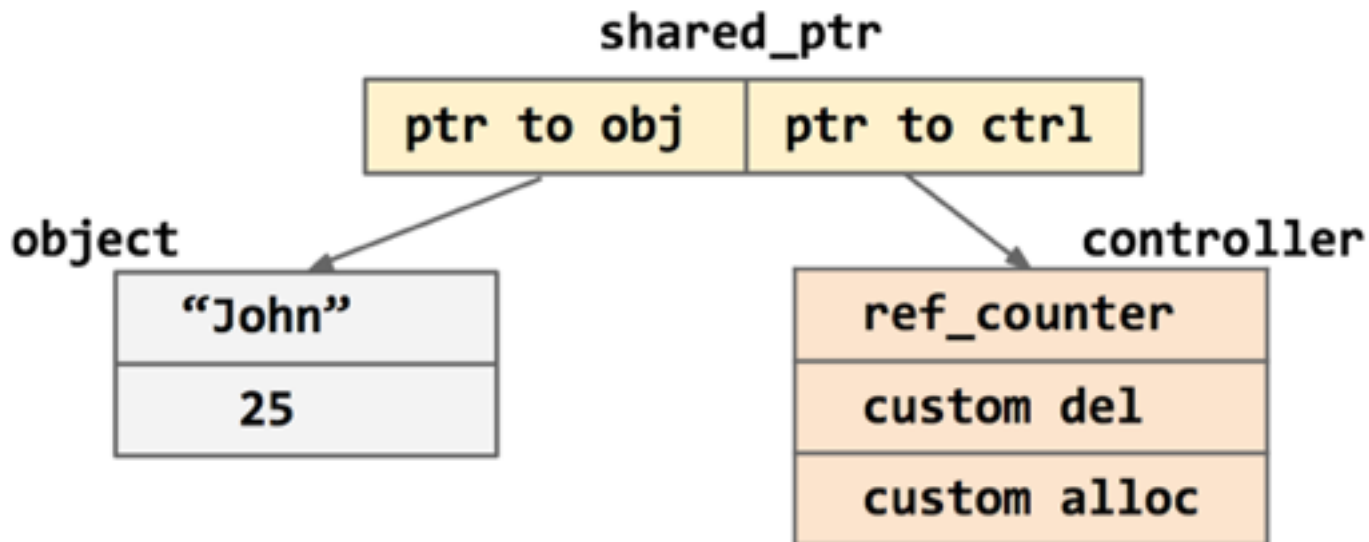
std::shared_ptr

- Shared pointers:
 - Provide shared ownership
 - Many pointers may own the same object
 - The last one to survive is responsible of its disposal through the deleter



std::shared_ptr

- Shared pointers have a garbage collection mechanism based on a reference counter contained in a control block
- Each new shared owner copies the pointer to the control block and increases the count by 1



std::shared_ptr

- Example:

```
std::shared_ptr p(new Person("John",25)); //ref-count = 1
std::shared_ptr q = p;                    //ref-count = 2
p.reset( new Person("Alice",23) );        //ref-count = 1
q = NULL;                                  //ref-count = 0
                                           //call delete!
```

- Important properties:
 - Copy-constructible and copy-assignable, so it can work inside STL containers
 - Non-trivial implementation and memory overhead (2 pointers: object itself, and control block)
 - Operations require atomic update of ref_counter (thread safety!), which may be slow

Do we still need raw pointers?

- Smart pointers implement automatic management of dynamically allocated memory. This comes with very little overhead!
- However, raw pointers are still needed
 - When the pointer really just means an address, not ownership of an object
 - To implement iterators
 - To handle an array decaying into a pointer
 - When working with legacy code (e.g. pre-C++11)
 - When passing arguments by address to modify their content

Outline

- references
- const
- namespaces
- pre-processor directives
- More on operators
- smart/shared pointers
- friend

friend

- Occasionally, we may want to allow a function that is not member of a given class to access its private fields/methods
- Can specify that a given external function gets full access by
 - Placing the signature of the external function inside the class
 - Preceding this signature copy by the keyword **friend**

friend

- Example:

```
class USCurrency {
    friend ostream& operator<< ( ostream &o, const USCurrency &c );
public:
    USCurrency ( const int d, const int c ) : dollars(d), cents(c) {}
private:
    int dollars, cents;
};

ostream& operator<< ( ostream &o, const USCurrency &c ) {
    o << '$' << c.dollars << '.' << c.cents;
    return o;
}
```

friend

- Can do the same with classes
 - Declaring other class as “friend” lets this class directly access private members of the other class

```
class A {  
    friend class B;  
    // More code ...  
};
```