

Classes

08/10/2024

Danilo Ardagna

Politecnico di Milano

danilo.ardagna@polimi.it



POLITECNICO
DI MILANO

Content

- Classes and Structs
- this
- Operators



Classes

- User-defined type, specifies a blueprint for objects
- Consists of a set of members
 - data members
 - member functions (methods)
- Member functions can define the meaning of creation (constructor), initialization, assignment, copy, and cleanup (destruction)



Classes – C++ general syntax

```
class X { // this class' name is X
public:// public members -- that's the interface to users
    // (accessible by all)
    // functions
    // types
    // data (often best kept private!)
private:// private members -- that's the implementation details
    // (accessible by members of this class only)
    // functions
    // types
    // data
};
```



Classes

- Members are accessed using . (dot) for objects and -> (arrow) for pointers
- Operators, such as +, !, and [], can be defined
- The **public** members provide the class interface and the **private** members provide implementation details



Classes

Class members are private by default:

```
class X {
    int mf();
    // ...
};
```

is equivalent to

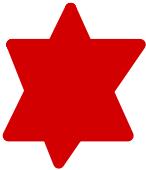
```
class X {
private:
    int mf();
    // ...
};
```

So

```
X x;           // variable x of type X
X *px = &x;    // pointer to type X
```

The syntax is wrong,
but the method
is private

```
int y = x.mf(); // error: mf is private (i.e., inaccessible)
int w = (*px).mf(); // error: mf is private (i.e., inaccessible)
int z = px->mf(); // error: mf is private (i.e., inaccessible)
```



Structs vs. Classes

A struct in C++ is a class where members are public by default:

```
struct X {  
    int m;  
    // ...  
};
```

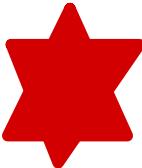
is equivalent to



```
class X {  
public:  
    int m;  
    // ...  
};
```

Question to answer today

- Which are the benefits of public/private?
- When shall we use a Class and when a Struct?
- Running example: let's implement a *Date* data structure



Public/private benefits

- Why bother with the public/private distinction?
- Why not make everything public?
 - To provide a clean interface
 - Data and messy functions can be made private
 - To allow a change of representation
 - You need only to change a fixed set of functions
 - You don't really know who is using a public member

you change internally
the code (i.e. private)

If internal representation is hidden (**information hiding principle**)

- It is easier to support code evolution
- We can change the internals without changing the remaining code

cont'd



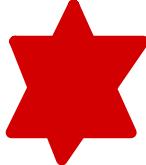
Public/private benefits

- Why bother with the public/private distinction?
- Why not make everything public?
 - To provide a clean interface
 - Data and messy functions can be made private
 - **To allow a change of representation**
 - You need only to change a fixed set of functions
 - You don't really know who is using a public member
 - **To ease debugging**
 - (known as the “round up the usual suspects” technique)
 - **To maintain an invariant**



Invariants → A rule allowing to tell if an object is valid or not.

- Example of Date:
 - The notion of a “valid Date” is an important special case of the idea of a valid value
- We try to design our types so that values are guaranteed to be valid
 - or we have to check for validity all the time
- A rule for what constitutes a valid value is called an “**invariant**”
 - The invariant for Date (“a Date must represent a date in the past, present, or future”) is unusually hard to state precisely
 - Remember February 28, leap years, etc.



Invariants

As a principle, the teacher
✓ never uses struct.

- If we can't think of a good invariant, we are probably dealing with plain data
 - • if so, use a struct

- Try hard to think of good invariants and use classes
 - that saves you from poor buggy code

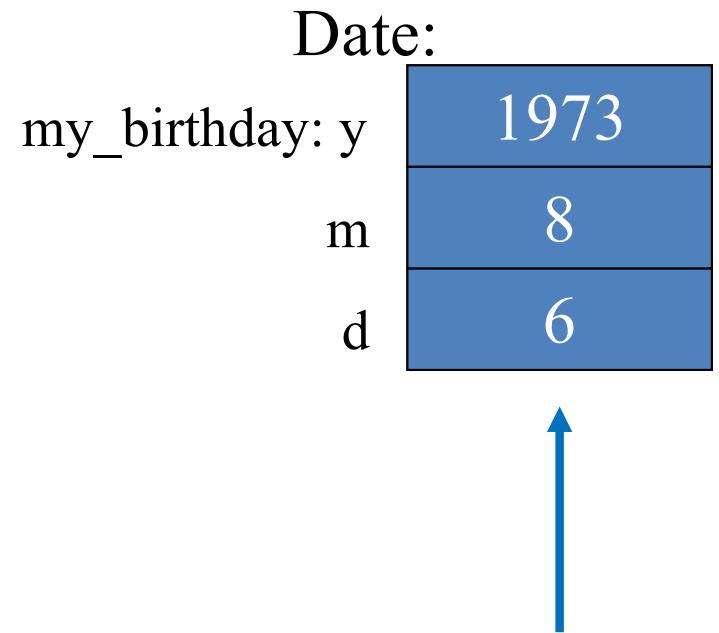
Structs

// simplest Date (just data)

```
struct Date {
    int y, m, d; // year, month, day
};
```

Date my_birthday; // a Date variable (object)

```
my_birthday.y = 1973;
my_birthday.m = 6;
my_birthday.d = 8;
```



This is the state
of my_birthday!!!

Structs

// simplest Date (just data)

```
struct Date {
    int y, m, d; // year, month, day
};
```

Date my_birthday; // a Date variable (object)

```
my_birthday.y = 6;
my_birthday.m = 8;
my_birthday.d = 1973;
```

my_birthday: y

m

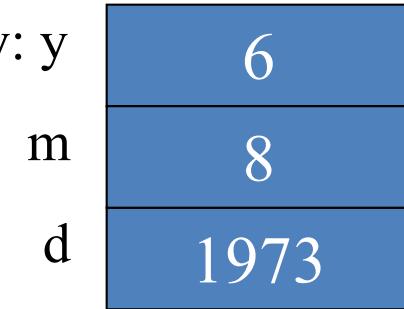
d

6

8

1973

Date:



*// oops! (no day 1973 in month 8)
// later in the program, we'll have a
// problem*

*we can create
a method to
check the
validity
of dates.*

Here we are clearly breaking the invariant ...

Structs – C version

// simple Date (with a few helper functions for convenience)

```
struct Date {  
    int y, m, d; // year, month, day  
};
```

Date my_birthday; // a Date variable (object)

// helper functions:

```
void init_day(Date& dd, int y, int m, int d);  
    // check for valid date and Initialize  
    // Note: this y, m, and d are local
```

```
void add_day(Date& dd, int n);  
    // increase the Date by n days
```

init_day(my_birthday, 6, 8, 1973); // run time error: no day 1973 in month 8

Structs – C version

// simple Date (with a few helper functions for convenience)

```
struct Date {
    int y, m, d; // year, month, day
};
```

Date my_birthday; // a Date variable (object)

// helper functions:

```
void init_day(Date& dd, int y, int m, int d);
    // check for valid date and Initialize
    // Note: this y, m, and d are local
```

```
void add_day(Date& dd, int n);
    // increase the Date by n days
```

init_day(my_birthday, 1973, 6, 8);

Date:

my_birthday: y

1973

m

8

d

6

Structs – C version

// simple Date (with a few helper functions for convenience)

```
struct Date {
    int y, m, d; // year, month, day
};
```

Date my_birthday; // a Date variable (object)

// helper functions:

```
void init_day(Date& dd, int y, int m, int d);
    // check for valid date and Initialize
    // Note: this y, m, and d are local
```

```
void add_day(Date& dd, int n);
    // increase the Date by n days
```

init_day(my_birthday, 1973, 6, 8);

Date:

my_birthday: y

1973

m

8

d

6

{ check validity

Structs – C++ v. 0.1

```

// simple Date
//     guarantee initialization with constructor
//     provide some notational convenience

struct Date {
    int y, m, d; // year, month, day
    Date(int y, int m, int d); // constructor: check for valid
                               // date and initialize
    void add_day(int n); // increase the Date by n days
};

// ...
Date my_birthday(8, 6, 1973); // oops! Runtime error
Date my_day(1973, 6, 8); // ok

```

Date:

my_day: y	1973
m	6
d	8



Structs – C++ v. 0.1

```

// simple Date
//     guarantee initialization with constructor
//     provide some notational convenience

struct Date {
    int y, m, d; // year, month, day
    Date(int y, int m, int d); // constructor: check for valid
                               // date and initialize
    void add_day(int n); // increase the Date by n days
};

// ...

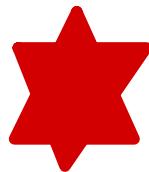
Date my_birthday(8, 6, 1973); // oops! Runtime error
Date my_day(1973, 6, 8);      // ok
my_day.add_day(2);           // June 10, 1973

```

Date:

my_day: y	1973
m	6
d	10

Structs – C++ v. 0.1



Date:

my_day: y

1973

m

14

d

10

```

// simple Date
// guarantee initialization with constructor
// provide some notational convenience

struct Date {
    int y, m, d; // year, month, day
    Date(int y, int m, int d); // constructor: check for valid
                               // date and initialize
    void add_day(int n); // increase the Date by n days
};

// ...

Date my_birthday(8, 6, 1973); // oops! Runtime error
Date my_day(1973, 6, 8); // ok
my_day.add_day(2); // June 10, 1973
my_day.m = 14; // ouch! (now my_day is a
                // bad date)

```

*Let's use
classes instead;*



*Why
structs
are bad*

We can write this



SINCE WE HAVE ONLY PUBLIC METHODS !!

Classes – C++ v. 0.2



Date:

	1973
m	6
d	8

// simple Date (control access)

```
class Date {
    int y, m, d;           // year, month, day
public:
    Date(int y, int m, int d); // constructor: check for valid date
                                // and initialize
    void add_day(int n);      // increase the Date by n days
    int month() const { return m; } // access functions:
    int day() const { return d; }
    int year() const { return y; }
};

// ...
Date my_birthday(1973, 6, 8); // ok
cout << my_birthday.month() << endl; // we can read
my_birthday.m = 14; // error: Date::m is private
```

Classes

```
// simple Date (some people prefer implementation details last)
class Date {
public:
    Date(int yy, int mm, int dd); // constructor: check
                                // for valid date and initialize
    void add_day(int n);        // increase the Date by n days
    int month() const;
    // ...
private:
    int y,m,d;                // year, month, day
};

Date::Date(int yy, int mm, int dd) // definition; note :: member of"
    :y(yy), m(mm), d(dd) /* ... */; // note: member initializers
void Date::add_day(int n) /* ... */; // definition
```

Classes



Date.h (or Date.hh or Date.hpp)

```
// simple Date (some people prefer implementation details last)
class Date {
public:
    Date(int yy, int mm, int dd); // constructor: check
                                // for valid date and initialize
    void add_day(int n);      // increase the Date by n days
    int month() const;
    // ...
private:
    int y,m,d;              // year, month, day
};
```

```
Date::Date(int yy, int mm, int dd) // definition; note :: member of "
:y(yy), m(mm), d(dd) { /* ... */ }; // note: member initializers
void Date::add_day(int n) { /* ... */ }; // definition
```

Classes



```
// simple Date (some people prefer implementation details last)
class Date {
public:
    Date(int yy, int mm, int dd); // constructor: check
                                // for valid date and initialize
    void add_day(int n);      // increase the Date by n days
    int month() const;
    // ...
private:
    int y,m,d;              // year, month, day
};
```

```
Date::Date(int yy, int mm, int dd) // definition; note :: member of""
:y(yy), m(mm), d(dd) /* ... */;// note: member initializers
void Date::add_day(int n) { /* ... */}; // definition
```

Classes



```
// simple Date (some people prefer implementation details last)
class Date {
public:
    Date(int yy, int mm, int dd); // constructor: check
                                // for valid date and initialize
    void add_day(int n);         // increase the Date by n days
    int month() const;
    // ...
private:
    int y,m,d;                // year, month, day
};
```



```
int month() { return m; } // error: forgot Date::
// this month() will be seen as a global function not the member function, so can't access
// members
int Date::season() { /*...*/ } // error: no member season
```

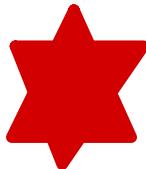
See the exercise notes for
the full implementation of Date class.

season is not declared within class.

Date.cpp

this

this



```
Date my_birthday (1973, 6, 8);  
int m1 = my_birthday.month();
```

- Here we use the dot operator to run `month()` on the object named `my_birthday`
- With the exception of static members, **when we call a member function we do so on behalf of an object**
- When `month()` refers to members of `Date` (e.g., `m`), it is referring implicitly to the members of the object on which the function was called. In this call, when `month()` returns `m`, it is implicitly returning `my_birthday.m`

this



- Member functions access the object on which they were called through an extra, implicit parameter named **this**
- When we call a member function, **this is initialized with the address of the object on which the function was invoked**. For example, when we call

```
my_birthday.month();
```

- the compiler passes the address of `my_birthday` to the implicit **this** parameter. It is as if the compiler rewrites this call as

// *pseudo-code illustration of how a call to a member function is translated*
int Date::month(**Date * this**) *That's what happens implicitly.*

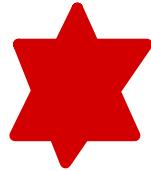
```
Date::month(&my_birthday)
```

- which calls the `month` member of `Date` passing the address of `my_birthday`

this

- Inside a member function, we can refer directly to the members of the object on which the function was called
- Any direct use of a member of the class is assumed to be an implicit reference through `this`. That is, **when `month()` uses `m`, it is implicitly using the member to which `this` points.** It is as if we had **written `this->m`**

this



- The `this` parameter **is defined for us implicitly**:
 - It is illegal for us to define a parameter or variable named `this`
 - Inside the body of a member function, we can use `this`
 - It would be legal, although unnecessary, to define `month()` as

```
int month() { return this->m; }
```

because `this` is intended to always refer to “this” object

- `this` is a **const pointer**, we cannot change the address that `this` holds

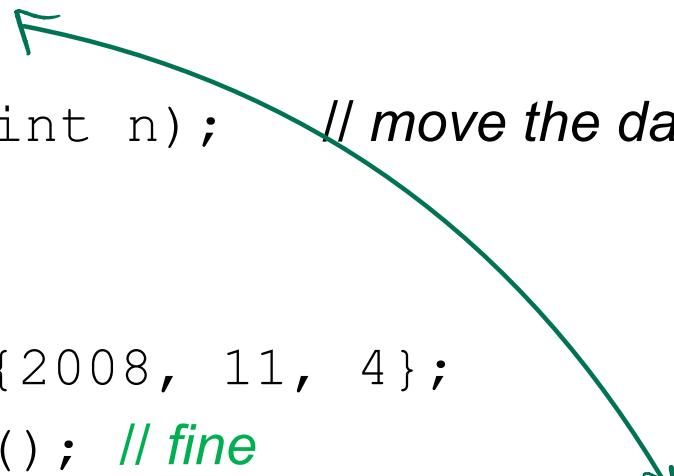
DEMO

More on const

Const member functions

```
// Distinguish between functions that can modify (mutate) objects
// and those that cannot ("const member functions")
class Date {
public:
    // ...
    int day() const; // get (a copy of) the day
    // ...
    void add_day(int n); // move the date n days forward
    // ...
};

const Date dx {2008, 11, 4};
int d = dx.day(); // fine
dx.add_day(4); // error: can't modify constant (immutable) date
```



Const member functions

```
Date d (2004, 1, 7);           // a variable
const Date d2 (2004, 2, 28);   // a constant
d2 = d;                      // error: d2 is const
d2.add(1);                   // error: d2 is const
d = d2;                      // fine
d.add(1);                   // fine
```



Classes: What makes a good interface?

- Minimal
 - As small as possible
- Complete
 - And no smaller
- Invariant preserving
 - Invariants hold from object creation (i.e., constructors!) and for every operation performed (non-const methods!)
- const correct



Interfaces and “helper functions”

- Keep a class interface (the set of public functions) minimal
 - Simplifies understanding
 - Simplifies debugging
 - Simplifies maintenance
- When we keep the class interface simple and minimal, we need extra “helper functions” outside the class (non-member functions)
 - `next_weekday()`, `next_Sunday()`
 - `==` (equality), `!=` (inequality)



Helper functions

```
Date next_Sunday(const Date& d)
{
    // access d using d.day(), d.month(), and d.year()
    // make new Date to return
}
```

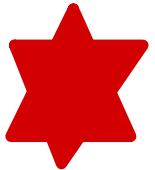
```
Date next_weekday(const Date& d) { /* ... */ }
```

```
bool operator==(const Date& a, const Date& b)
{
    return a.year() == b.year() && a.month() == b.month() && a.day() == b.day();
}
```

```
bool operator!=(const Date& a, const Date& b) {
    return !(a==b); }
```

"operator overloading": we redefine operator "==" for Date datatype.

year is private so we need *.year()*.



Helper functions

```
Date next_Sunday(const Date& d)
{
    // access d using d.day(), d.month(), and d.year()
    // make new Date to return
}
```

```
Date next_weekday(const Date& d) { /* ... */ }
```

- b
- Declare helper functions in the Class header
 - Define helper function in the Class source (.cpp) file

```
&& d.day() == b.day();  
}
```

```
bool operator!=(const Date& a, const Date& b) {
    return !(a==b); }
```

Operators



Operator overloading



- You can define only existing operators
 - E.g., + - = += * / % [] () ^ ! & < <= > >=
- You can define operators only with their conventional number of operands
 - E.g., no unary <= (less than or equal) and no binary ! (not)
- An overloaded operator must have at least one user-defined type as operand
 - `int operator+ (int, int);` // error: you can't overload built-in +
 - `Vector operator+ (const Vector&, const Vector &);` // ok



Advices

- Overload operators only with their conventional meaning
 - + should be addition, * be multiplication, [] be access, () be call, etc.
- Don't overload unless you really have to
- Don't overload , * && || !
 - Operand-evaluation are not preserved
 - Short circuit does not work anymore

Class example: MatlabVector

- We want to implement Matlab like vectors in C++
 - Implement row vectors of double type
 - Elements indexing follows the C++ convention, i.e., the first element has index 0! **(-2 points at exams otherwise!)**
 - Vectors can grow as in Matlab
 - Simplified version: a read of an element which does not exist do not return error

	v[0]	v[1]	v[2]	v[3]
v:	0.33	22.0	27.2	54.2

$$v[6] = 4.1$$

	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]	v[6]
v:	0.33	22.0	27.2	54.2	0.0	0.0	4.1

Class example: MatlabVector

- **Goals:**
 - Provide operator +
 - Implement the product with a scalar: operator *
 - Provide operator [] to access individual elements
 - Neglect, in the beginning, errors (e.g., vectors size do not match)

MatlabVector

```
class MatlabVector {  
  
    vector<double> elem;  
  
public:  
    double get(size_t n); // access: read  
    void set(size_t n, double v); // access: write  
    size_t size() const; // return number of elements  
  
    MatlabVector operator*(double scalar) const;  
};  
  
MatlabVector operator+(MatlabVector& v1, MatlabVector& v2);
```

DEMO

MatlabVector

```
size_t MatlabVector::size() const {  
    return elem.size();  
}  
  
double MatlabVector::get(size_t n) {  
    while (elem.size() < n+1)  
        elem.push_back(0.);  
  
    return elem[n];  
}
```

MatlabVector

```
void MatlabVector::set(size_t n, double v) {  
  
    while (elem.size() < n+1)  
        elem.push_back(0.);  
  
    elem[n] = v;  
}
```

```
MatlabVector MatlabVector::operator*(double scalar) const{  
    MatlabVector result;  
  
    for (size_t i=0; i<elem.size(); ++i)  
        result.set(i, scalar * elem[i]);  
        // alternatively result.elem[i]=scalar * elem[i];  
  
    return result;  
}
```

MatlabVector

```
MatlabVector operator+(MatlabVector& v1, MatlabVector& v2)
{
    MatlabVector result;
    for (size_t i=0; i<v1.size(); ++i)
        result.set(i, v1.get(i) + v2.get(i));

    return result;
}
```

MatlabVector (primitive access)

```
MatlabVector v;  
for (size_t i=0; i<10; ++i) {// pretty ugly:  
    v.set(i,i);  
    cout << v.get(i);  
}  
  
for (size_t i=0; i<10; ++i) {// we're used to this:  
    v[i]=i;  
    cout << v[i];  
}
```

MatlabVector (we use references for access)

```
class MatlabVector {  
    vector<double> elem;  
public:  
    double & operator[](size_t n); // access: return reference  
    size_t size() const; // return number of elements  
  
    MatlabVector operator*(double scalar) const;  
};  
MatlabVector operator+(MatlabVector& v1, MatlabVector& v2);  
  
MatlabVector v;  
for (size_t i=0; i<10; ++i) { // works and looks right!  
    v[i] = i; // v[i] returns a reference to the ith element  
    cout << v[i];  
}
```

MatlabVector

```
double & MatlabVector::operator[](size_t int n) {  
  
    while (elem.size() < n+1)  
        elem.push_back(0.);  
  
    return elem[n];  
  
}
```

Operators member functions

- First operand (left hand) is bounded to this
 - They have one less explicit operator

```
class MatlabVector {  
  
    vector<double> elem;  
  
public:  
    double & operator[](size_t n); // access: return reference  
    size_t size() const; // return number of elements  
  
  
    MatlabVector operator*(double scalar) const;  
    MatlabVector operator+(const MatlabVector& rhs) const;  
};
```

Operators member functions

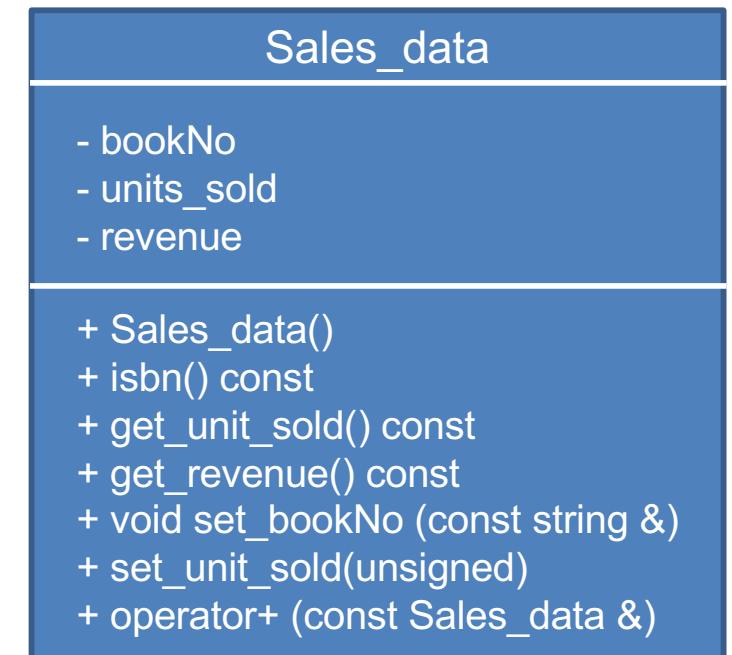
- First operand (left hand) is bounded to this
 - They have one less explicit operator

```
MatlabVector operator+(const MatlabVector& rhs) const
{
    MatlabVector result;
    for (size_t i=0; i< elem.size(); ++i)
        result[i] = elem[i] + rhs.elem[i];

    return result;
}
```

Class example: Sales_data

```
class Sales_data {  
private:  
    string bookNo;  
    unsigned units_sold;  
    double revenue;  
  
public:  
    Sales_data() :  
        bookNo("") ,  
        units_sold(0) ,  
        revenue(0.0)  
    {}
```



Class example: Sales_data

```
/* Getters and Setters */  
string isbn() const;  
unsigned get_unit_sold() const;  
double get_revenue () const;  
void set_bookNo (const string & bn);  
void set_unit_sold(unsigned u);  
void set_revenue (double r);  
};
```



Operators member functions

- First operand (left hand) is bounded to this
 - They have one less explicit operator

```
class Sales_data {
    //Other code
public:
    Sales_data operator+(const Sales_data &rhs) const;
}
```

```
Sales_data Sales_data::operator+(const
const {
    Sales_data ret;
    ret.bookNo = bookNo;
    ret.units_sold = units_sold + rhs
    ret.revenue = revenue + rhs.revenue
    return ret;
}
```



Operators member functions

- First operand (left hand) is bounded to this
 - They have one less explicit operator

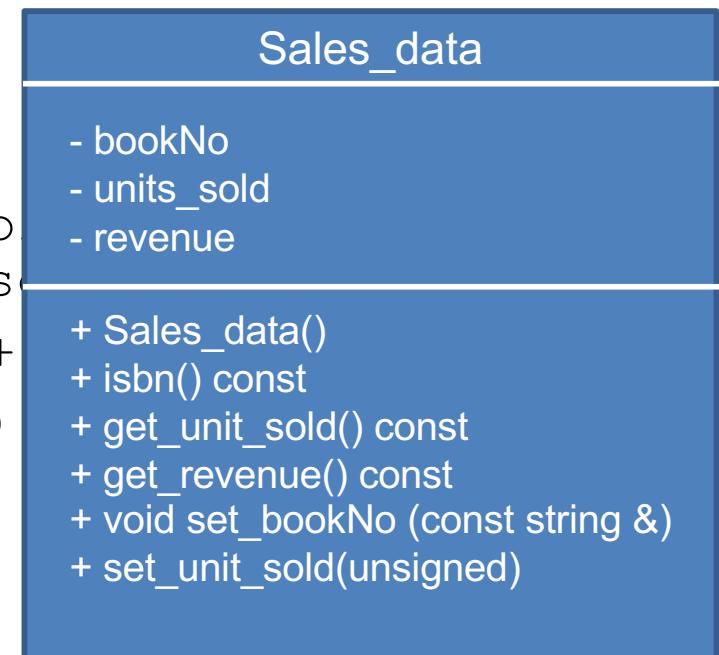
```
class Sales_data {  
    //Other code  
public:  
    Sales_data operator+(const Sales_data &rhs) const;  
}
```

```
Sales_data Sales_data::operator+(const & Sales_data rhs)  
const {  
    Sales_data ret;  
    ret.bookNo = bookNo;  
    ret.units_sold = units_sold + rhs.units_sold;  
    ret.revenue = revenue + rhs.revenue;  
    return ret;  
}
```

Operators non-member functions

- Same number of parameters as the operator
- They need to access to all data members of type
 - Typically declared as friend, we will see how in the next class

```
Sales_data operator+(const Sales_data & lhs,
                      const Sales_data & rhs)
{
    Sales_data ret;
    ret.set_bookNo(lhs.isbn());
    ret.set_units_sold(lhs.get_units_sold() +
                       rhs.get_units_sold());
    ret.set_revenue(lhs.get_revenue() +
                    rhs.get_revenue());
    return ret;
}
```



Operators non-member functions

- Same number of parameters as the operator
- They need to access to all data members of type
 - Typically declared as friend, we will see how in the next class

```
Sales_data operator+(const Sales_data & lhs,  
                      const Sales_data & rhs)  
{  
    Sales_data ret;  
    ret.set_bookNo(lhs.isbn());  
    ret.set_units_sold(lhs.get_units_sold() +  
                       rhs.get_units_sold());  
    ret.set_revenue(lhs.get_revenue() +  
                   rhs.get_revenue());  
    return ret;  
}
```

Operators non-member functions

- Same number of parameters as the operator
- They need to access to all data members of type
 - Typically declared as friend, we will see how in the next class

```
Sales_data operator+(const Sales_data & lhs,  
                      const Sales_data & rhs)  
{  
    Sales_data ret;  
    ret.set_bookNo(lhs.isbn());  
    ret.set_units_sold(lhs.get_units_sold() +  
                       rhs.get_units_sold());  
    ret.set_revenue(lhs.get_revenue() +  
                   rhs.get_revenue());  
    return ret;  
}
```

Member or non-member?

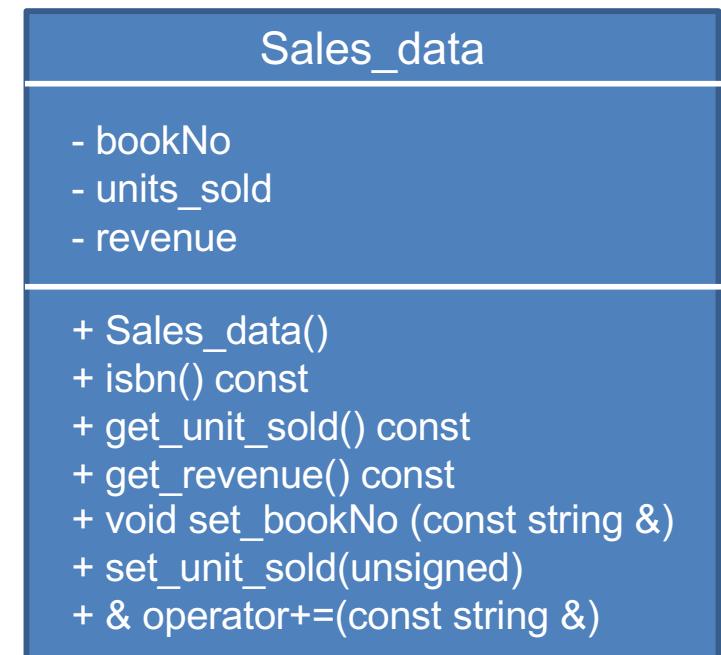
- Must be member
 - = [] () ->
- Should be member
 - Compound assignments += -= /= %= ^= &= |= *= <<= >>=
 - Modify operators ++ -- *
- Better non-member
 - Arithmetic operators + - * %
 - Bitwise operators ^ & |
 - Equality operators < > <= => != ==
 - Relational operators ! && ||

Member or non-member?

- Better not overloaded
 - * && || !
- Cannot be overloaded
 - :: . * . ? :

Defining a Function to Return “this” Object

```
class Sales_data {  
public:  
    std::string isbn() const { return bookNo; }  
    Sales_data& operator+=(const Sales_data&);  
    double get_revenue() const;  
  
private:  
    std::string bookNo;  
    unsigned units_sold;  
    double revenue;  
};
```



Why return a reference

- `+ =` etc. must return a reference
 - mimic built-in operators
- `a = b = c`
 - works also even with copies
- `(a = b) = c`
 - Return copy: `a` takes value of `b`
 - Return reference: `a` takes value of `c`

DEMO

Defining a Function to Return “this” Object

```
Sales_data trans;  
/* modify trans */  
Sales_data total;  
/* modify total */  
total += trans; // total.combine(trans); in the Lipman's book
```

Defining a Function to Return “this” Object

- The object on which this operator is called represents the left-hand operand of the assignment. The right-hand operand is passed as an explicit argument

```
Sales_data& Sales_data::operator+=(const Sales_data &rhs)
{
    units_sold += rhs.units_sold;// add the members of rhs
    revenue += rhs.revenue;           // into the members of "this"
                                    // object
    return *this; // return the object on which the function was
                  // called
}
```

Defining a Function to Return “this” Object

```
total += (trans); // update the running total
```

- the address of total is bound to the implicit this parameter and rhs is bound to trans
- Thus, when += executes:

```
    units_sold += rhs.units_sold;
```

- the effect is to add total.units_sold **and** trans.units_sold, storing the result back into total.units_sold
- the same happens for revenues

Defining a Function to Return “this” Object

- The interesting part about this operator is its return type and the return statement
- When we define an operator, it should mimic the behavior of the built-in operator
 - The built-in assignment operators return their left-hand operand as an lvalue
 - To return an lvalue, our operator **must** return a reference, because the left-hand operand is a `Sales_data` object, the return type is `Sales_data&`

Defining a Function to Return “this” Object

- As we have seen, we do not need to use the implicit `this` pointer to access the members of the object on which a member function is executing. However, we do need to use `this` to access the object as a whole:

```
return *this; // return the object on which the function  
              // was called
```

- Here the return statement dereferences `this` to obtain the object on which the operator is executing

References

- Lippman Chapters 1, 7, 14

Credits

- Bjarne Stroustrup. www.stroustrup.com/Programming