Classes

PIC 10A, UCLA

Classes

A class provides a nice packaging for related data and functions. We have thus far studied how to use classes, should they be provided. Here, we will look at writing our own classes to accomplish various tasks.

Using Classes

A **class interface** defines a class, telling us what class consists of.

It can include the declarations (or even definitions) of its member functions.

A class typically begins with either the **class** or **struct** keyword, followed by the class name, then the contents of the class are placed inside of braces, followed by a semicolon.

```
class nameOfClass {
public:
   // stuff
private:
   // other stuff
};
```

Using Classes

Everything following a **public access modifier**, the **public:**, until another access modifier arises is publicly accessible, i.e., all users of the class can use/access any member that is **public**.

Everything following a **private access modifier**, the **private:**, until another access modifier arises is part of the **private interface**, i.e. by default, nothing outside of the class can access or modify these private members.

Using Classes

A member function or member variable can be accessed (if accessible) through the **. member access operator**. Consider the class interface:

```
class Foo {
public:
    Foo();
    int x;
    int bar() const;

private:
    Foo(int a, int b);
    int y;
    void baz() const;
};
```

Using Classes

As users of this class, we can do all of:

```
Foo f; // default constructor Foo() is public
f.x = 7; // change value of x within f to 7: x is public
cout << f.bar(); // will print whatever bar ouputs: bar is public
```

But we cannot do any of:

```
Foo g(3,4); // ERROR: foo(int, int) is a private constructor
cout << f.y; // ERROR: y is private, cannot access it
f.baz(); // ERROR: baz is private, cannot use it
```

class vs struct

The **class** and **struct** keywords are both used to create classes. The only difference between the two is their default level of access.

By default, until an access modifier is listed, all members (variables and functions) of a **class** are **private** up to that point. If there is no access modifier listed, then all members are **private**.

By default, unless an access modifier is listed, all members (variables and functions) of a **struct** are **public** up to that point. If there is no access modifier listed, then all members are **public**.

class vs struct

```
class Bar {
    Bar();
    int b;
}; // this is the same as

class Bar {
private:
    Bar();
    int b;
}; // this, which is even the same as ...

struct Bar {
private:
    Bar();
    int b;
}; // this!
```

class vs struct

```
struct Baz {
    Baz(); // will be public
private:
    int b;
}; // this is the same as

struct Baz {
public:
    Baz();
private:
    int b;
}; // this, which is even the same as ...

class Baz {
    int b; // will be private
public:
    Baz();
}; // this!
```

Planning Classes

We will consider a **Chemical** class, used to store the name and quantity (volume) of a chemical in storage in litres, along with its shipment date in format **dd/mm/yy**.

In creating a class, we need to be mindful of what data types / variables could be relevant for the class (these will be its member variables) and the sort of functionality we would like it to have (these will be its member functions).

Planning Classes: Member Variables

We begin by considering **member variables**: these are variables that each instance of a class should contain.

To represent the type of chemical, we choose to use a variable of type **std::string** and we will call it **chemicalName**.

To represent the quantity of fluid, we choose a variable called **volumeInLitres**, of type **double** to track its volume in L (litres).

To represent the date, we will use three **unsigned char**s named **day**, **month**, and **year**.

Planning Classes: Member Variables

For example, we could have an instance of **Chemical** with

- **chemicalName** of “Pylakor Blue Dye”
- **volumeInLitres** of 0.05 L
- **day** of 1 [for 1st of month]
- **month** of 7 [for July]
- **year** of 10 [for 2010]

Planning Classes: Accessor Member Functions

We now consider **accessor functions**: these are functions that give information about an object of type **Chemical** but do not modify it in any way.

We consider a few possible functions (of many that could be useful).

Planning Classes: Accessor Member Functions

A function **showLabel** that prints the chemical name, volume, and shipment date to the console window. For example, if **bottle** is an instance of the **Chemical** class with Pylakor Blue Dye (back 2 slides), we could call

```
bottle.showLabel();
```

and obtain the output

```
Pylakor Blue Dye, 0.05 L, shipped 1/7/10
```

Planning Classes: Accessor Member Functions

A function **getName** that returns the name of the chemical as a **std::string**.

```
string name = bottle.getName();

cout << name;
```

```
Pylakor Blue Dye
```

Planning Classes: Accessor Member Functions

A function **containsAtLeast** that accepts a volume in litres and returns **true** if the volume of fluid in the container is at least as large as the input volume and otherwise returns **false**.

```
bool contains1Litre = bottle.containsAtLeast(1); // will be false
```

Planning Classes: Mutator Member Functions

We now consider a **mutator function**. A mutator function may change the member variables of a class.

We consider a mutator function, **extractVolume**, that accepts a volume in litres of a chemical to extract. It then displays a message saying some chemical was used and reduces the volume of chemical stored.

```
bottle.extractVolume(0.01); // extracts 0.01 L
```

```
Extracted 0.01 L.
```

Planning Classes: Constructors

We now consider how we would like to **construct** instances of the class, i.e., we now determine what **constructors** are appropriate.
We first consider a constructor that accepts 5 arguments: a name for the chemical, a volume in litres; and the day, month, and year and creates the object from those arguments.

```
Chemical bigBottle( "Polydimethylsiloxane", 20, 30, 1, 17 );
```

This should create a **Chemical** variable called **bigBottle** with

- **chemicalName == "Polydimethylsiloxane"**,
- **volumeInLitres == 20**,
- **day == 30**,
- **month == 1**, and
- **year == 17**.

Planning Classes: Constructors

Now we consider a constructor that accepts a single argument representing the volume of chemical, with the chemical being set to “Water” and the day/month/year being all set to 0.

```
Chemical bigBottle2 ( 20 );
```

This should create a **Chemical** variable called **bigBottle2** with

- **chemicalName == "Water"**,
- **volumeInLitres == 20**,
- **day == 0**,
- **month == 0**, and
- **year == 0**.

Planning Classes: Constructors

Now we consider a default constructor, taking no arguments, and setting the chemical variable to 1 litre of water, with 0/0/0 for the day/month/year.

```
Chemical bigBottle3;
```

This should create a **Chemical** variable called **bigBottle3** with

- **chemicalName == "Water"**,
- **volumeInLitres == 1**,
- **day == 0**,
- **month == 0**, and
- **year == 0**.

Defining Classes

In designing a class, we wish to provide the user of the class with all the functions they would reasonable need, while at the same time protecting the data from abuse.

Defining Classes I

We want the constructors and member functions to be **public** and the member variables to be **private**. We can define the class by its interface below:

```
class Chemical {

private:

    std::string chemicalName; // stores the name of the chemical

    double volumeInLitres; // volume of the chemical in litres

    unsigned char day, month, year; // the day/month/year of shipment
```

Defining Classes II

```
public:

    /**
    Class constructor: creates a Chemical with a name, volume (in L), day,
    month, and year of shipment

    @param name the name of the chemical
    @param volume the volume of the chemical in litres
    @param shipDay the day of shipment 1-31
    @param shipMonth the month of shipment 1-12
    @param shipYear the year of shipment 0-99
    */
    Chemical( const std::string& name, double volume,
       unsigned char shipDay, unsigned char shipMonth,
       unsigned char shipYear );
```

Defining Classes III

```
/**
Class constructor: creates a Chemical storing "Water", with shipment
day/month/year of 0/0/0, with a given input volume

@param volume the volume of water in litres
*/
Chemical ( double volume );

/**
Class constructor: default constructor creates a Chemical storing 1 L
of “Water”, with shipment day/month/year of 0/0/0
*/
Chemical();
```

Defining Classes IV

```
/**
Displays the chemical information to the console in the format
[CHEMICAL NAME], [VOLUME] L, shipped [DAY]/[MONTH]/[YEAR]
*/
void showLabel() const;

/**
Returns the name of the chemical

@return the chemical name
*/
const std::string& getName() const;
```

Defining Classes V

```
/**
Determines whether there is as much chemical as the input volume
specified

@param minVolume the minimum volume required
@return whether the volume of chemical is at least as much as
minVolume, true if so, false otherwise
*/
bool containsAtLeast( double minVolume ) const;
```

Defining Classes VI

```
    /**
    Extracts a given volume of chemical and displays a message with how
    much has been extracted

    @param toExtract the volume of chemical to extract
    */
    void extractVolume ( double toExtract );

};
```

Defining Classes

Remarks:

- Note the **public:** and **private:** access specifiers. If these were to be absent, for classes defined by the **class** keyword, all members (variables and functions) would be private.

- Note that the accessor functions have the extra **const** keyword! This is necessary for **const correctness**.

- Note that the class interface terminate with a semicolon **;**

Defining Classes

- A **class is defined** by being given an interface, even if its members are only declared.
- There are some rumors out there that all C++ classes should have “getters” and “setters” for every member variable, i.e. functions that return the member variable as accessors and functions that allow each member variable to be modified. This is poor coding practice.

Defining Classes

Unless a function actually adds to the intended functionality of a class, it should not be included. We do not ever just return the date, for example, but it is sufficient to have the **showLabel** function.

Of course if we ever needed a function to return the date, we could write one.

Having “setters” for every member variable violates **encapsulation**, the practice of protecting member data from inappropriate modification.

Defining Member Functions and Constructors Inside/Outside the Class

There are two ways that classes and their member functions can be defined.

- The first option is to define the class and only provide the declarations of its member functions (as we have done with **Chemical**), and then to define those member functions outside of the class interface.

- The second option is to define the class and within its body, to define the member functions.

It's also possible to mix-and-match, providing some member function definitions within the class and others outside.

Defining Member Functions and Constructors Inside/Outside the Class

```
class Option1 {
public:
    void foo() const; // only declared
    // other stuff like constructors
    // ...
}; // now Option1 has been defined, but foo is only declared
```

To define the member outside the class, we must be clear we are talking about the **foo** of **Option1**, by writing **Option1::foo** instead of **foo** alone:

```
// defining foo
void Option1::foo() const {
    cout << "foo";
}
```

Defining Member Functions and Constructors Inside/Outside the Class

Note that had we just defined

```
void foo() const {
    cout << "foo";
}
```

This would be a compiler error: the compiler thinks we are defining a function **void foo()** instead of the member function of **Option1**; in addition, functions that are non-member functions cannot have the **const** keyword appearing immediately before the body because they are not member functions and we are not promising to keep any member variables constant...

Defining Member Functions and Constructors Inside/Outside the Class

```
class Option2 {
public:
    void foo() const {
        cout << "foo";
    } // Option2::foo() is defined here (has body!)
    // other stuff like constructors
    // ...
}; // Option2 is defined
```

Defining Member Functions

We can now look at defining the member functions of the class as we have only declared these functions.

To define a function outside of the class, we need to specify that what we define is part of the class scope and make use of the **scoping operator ::**.

Function/constructor names are prefixed by **Chemical::** (or more generally *className::*).

Within a member function, whether defined inside the class or outside the class, we can refer to all member variables and other member functions directly by name.

Defining Member Functions

```
void Chemical::showLabel() const {

    // display the chemical information

    // cast the unsigned chars to ints to see numbers
    cout << chemicalName << ", "<< volumeInLitres << " L, shipped "
        << static_cast<int>(day) << "/"<< static_cast<int>(month) <<
        "/"<< static_cast<int>(year);

}
```

Note how we can speak of **chemicalName** as though it is a local variable. All member variables are within scope for member functions.

While **chemicalName** is private, the class is able to access/modify its own member variables even if they are private.

Defining Member Functions

```
/* will return a reference to const string, referencing the chemical name.
   In doing so, no copy is made. */
const std::string& Chemical::getName() const {
   return chemicalName; // return the name
}
```

Note how the output of **getName** is a **const std::string&**: we are returning a reference to the local member variable and not a copy of it.

A reference to an object takes up less space and is more efficient than a copy of an object.

Defining Member Functions

```
void Chemical::extractVolume( double toExtract ) {
    if ( containsAtLeast(toExtract) ) { // if there is enough chemical
        volumeInLitres -= toExtract; // remove that much volume

        cout << "Extracted "<< toExtract << "L.";
    }
    else { // if not enough chemical
        cout << "Insufficient chemical."; // warn not enough
    }
}
```

Note how we can refer to **containsAtLeast** as any ordinary function.

This function does not have the const keyword because it can modify **volumeInLitres**.

Defining Constructors

A constructor sets its member variables through a **constructor initializer list**: following the constructor name, this is a list of the member variables it is supposed to set, with their construction parameters in parentheses, separated by commas. Following this, we can add additional instructions inside a set of braces or leave the braces empty.

[with scope if defined outside class::]className
(comma separated list of input arguments) :
firstMember (comma separated list of construction parameters),
secondMember (comma separated list of construction parameters),
...,
lastMember (comma separated list of construction parameters) {
*   a body;*
}

Defining Constructors

This is best seen by example:

```
Chemical::Chemical( double volume ) : chemicalName( "Water"),
volumeInLitres ( volume ), day(0), month(0), year(0) { }
```

The variable **chemicalName** is set to "Water"; the variable **volumeInLitres** is set to whatever input volume is given as an argument, and **day**, **month**, and **year** are all set to 0.

Defining Constructors

For the default constructor:

```
// give a default set of values for Chemical
Chemical::Chemical() : chemicalName( "Water"), volumeInLitres ( 1 ),
day(0), month(0), year(0) { }
```

Defining Constructors

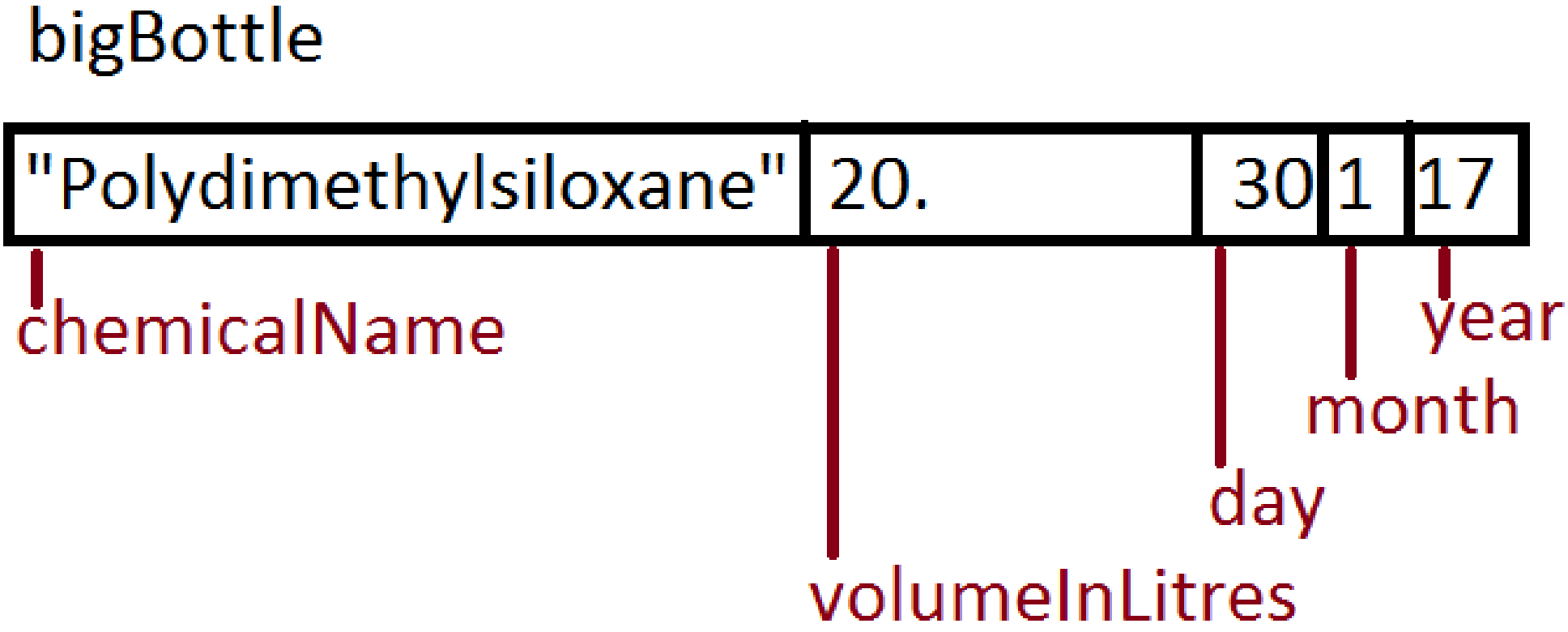
For the other constructor:

```
// use all parameters given to construct Chemical given its properties
Chemical::Chemical( const std::string& name, double volume,
      unsigned char shipDay, unsigned char shipMonth,
      unsigned char shipYear ) :
      chemicalName( name ), volumeInLitres ( volume ), day ( shipDay),
      month ( shipMonth ), year ( shipYear ) { }
```

We use all the arguments passed to the constructor to set the member variables. We construct **chemicalName** by telling the compiler to copy the value of the **name** parameter, etc.

Defining Constructors

A class stores all of its data together, in the order the member variables are declared within the class. Thus, we could visualize a **Chemical** variable as something like:



Defining Constructors

It is important to initialize all member variables of a class during its construction to a sensible value.

Without initializing a numeric type member variable, the compiler will **default initialize** it. Recall:

- **default initialization** means the value is whatever pattern of bits happen to be in memory where the variable is created - the value could be anything!

- **value initialization** means setting the value to **0**. For example, std::vector<double> v(14); // will store 14 0's

Defining Constructors

Without initializing a class member variable, it will be **default constructed**. This means that it will be set according to its default constructor.

If default construction is required for a member variable and a member variable does not have a default constructor, this will be a compiler error.

Defining Constructors

Consider

```
struct Employee {
    Employee( const std::string& _name, double _salary ) :
        name ( _name ), salary ( _salary ) { }

    std::string name;
    double salary;
    bool shouldBeFired;
};

// Alice might just have a really awful first day at work!
Employee alice( "Alice", 50000 );
```

Defining Constructors

Given we didn't set **shouldBeFired**, the bit pattern where **alice.shouldBeFired** lives in memory may be set to **true** and she could be fired on her first day!!!

Catching bugs due to not initializing a variable can be difficult.

Defining Constructors

Here's a better rendition of the class:

```
struct Employee {
    Employee( const std::string& _name, double _salary ) :
        name ( _name ), salary ( _salary ), shouldBeFired( false ) { }

    std::string name;
    double salary;
    bool shouldBeFired;
};
```

Here we have initialized the **shouldBeFired** variable.

Defining Constructors

The member variables are initialized in the same order they are declared inside the class, regardless of the order their initializations are given within the constructor initializer list!

Before the constructor body is read, all the variable are initialized.

Defining Constructors

Consider the class

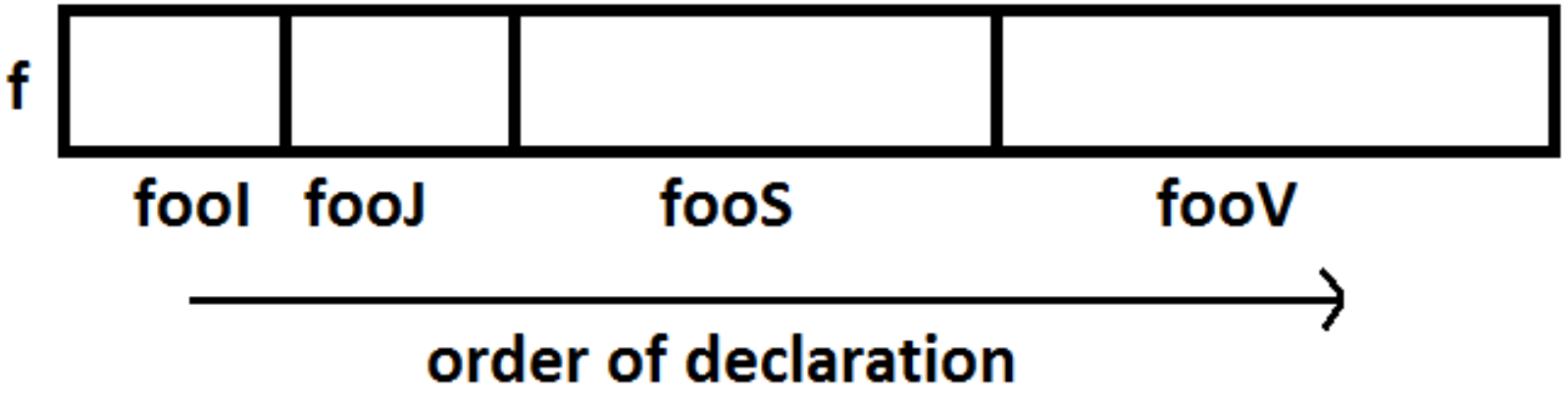
```
class Foo {
public:
    Foo( const string& s, int a, size_t b, char c ) :
        fooS(b, c), fooJ(fooI), fooI(a*a), fooV( b, s ) {
        fooI = 119; // modifies fooI inside the body
    }

private:
    int fooI, fooJ;
    string fooS;
    vector<string> fooV;
};
```

with the construction call

```
Foo f("bar", 3, 2, '!');
```

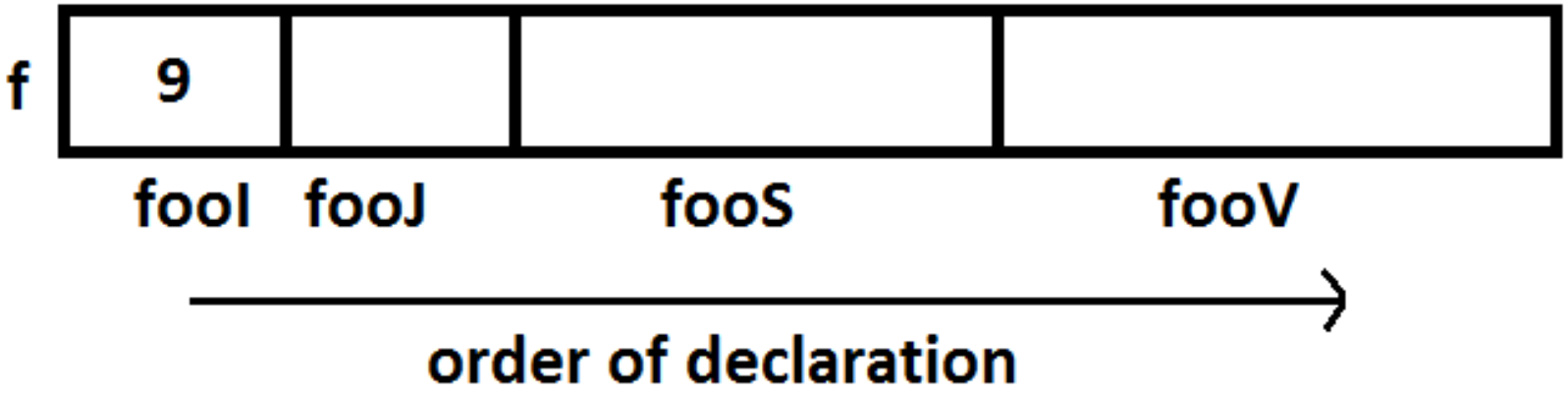
Defining Constructors




```
foo f("bar", 3, 2, '!');
```

s a b c

Defining Constructors

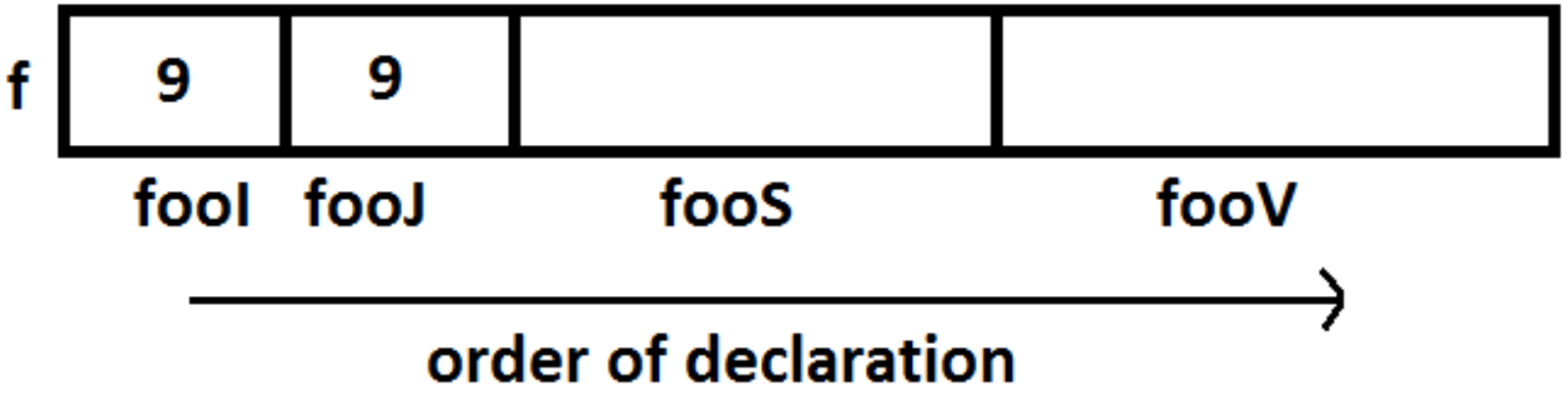



```
foo f("bar", 3, 2, '!');
```

s a b c

```
fooI(a*a)
```

Defining Constructors

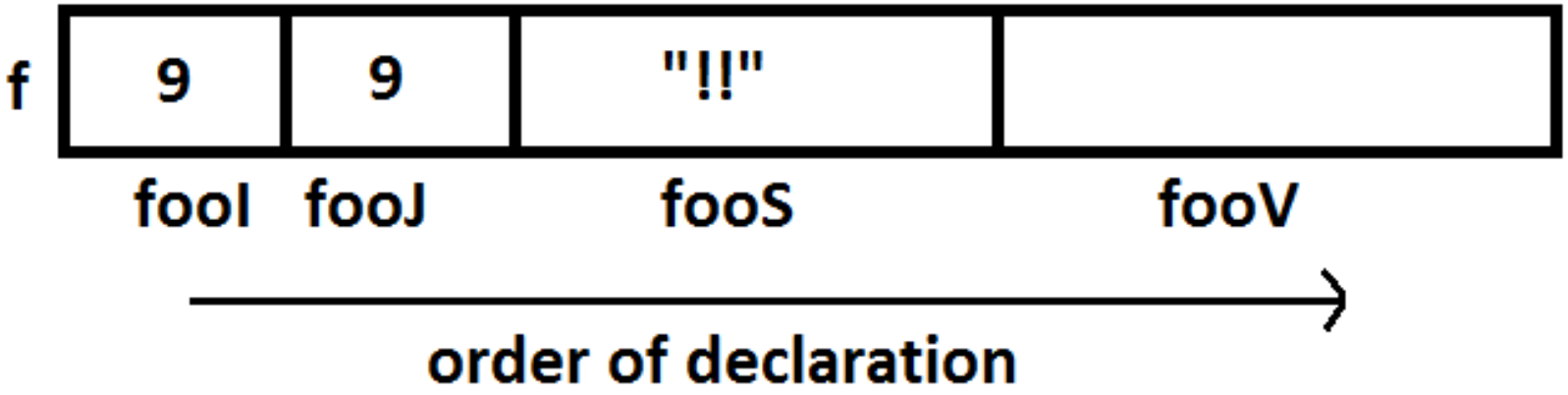



```
foo f("bar", 3, 2, '!');
```

s a b c

```
fooJ( fooI )
```

Defining Constructors



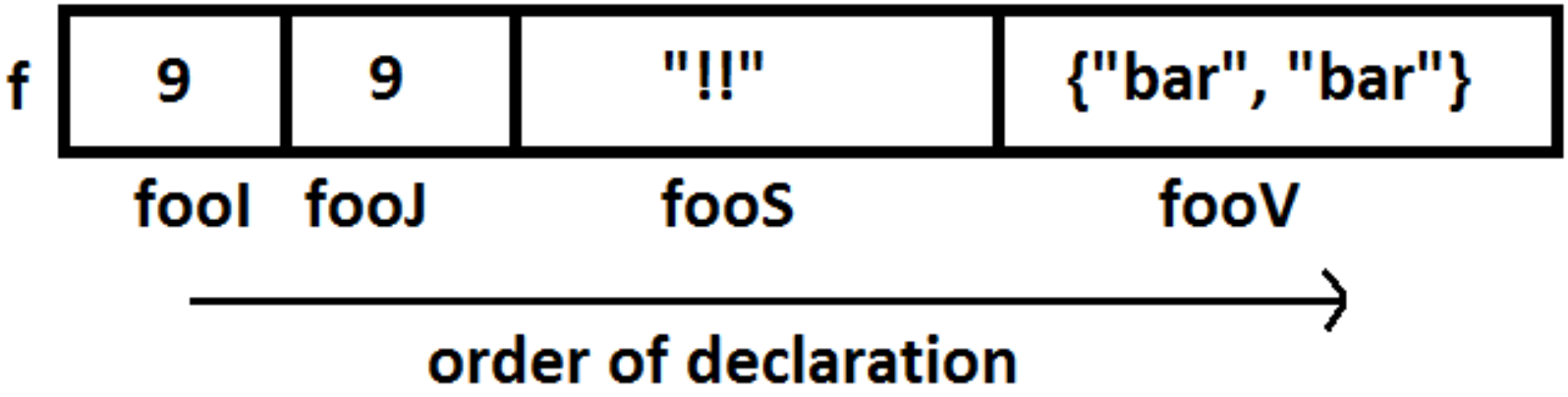

```
foo f("bar", 3, 2, '!');
```

s a b c

fooS(b, c)

i.e. string(2,'!')

Defining Constructors



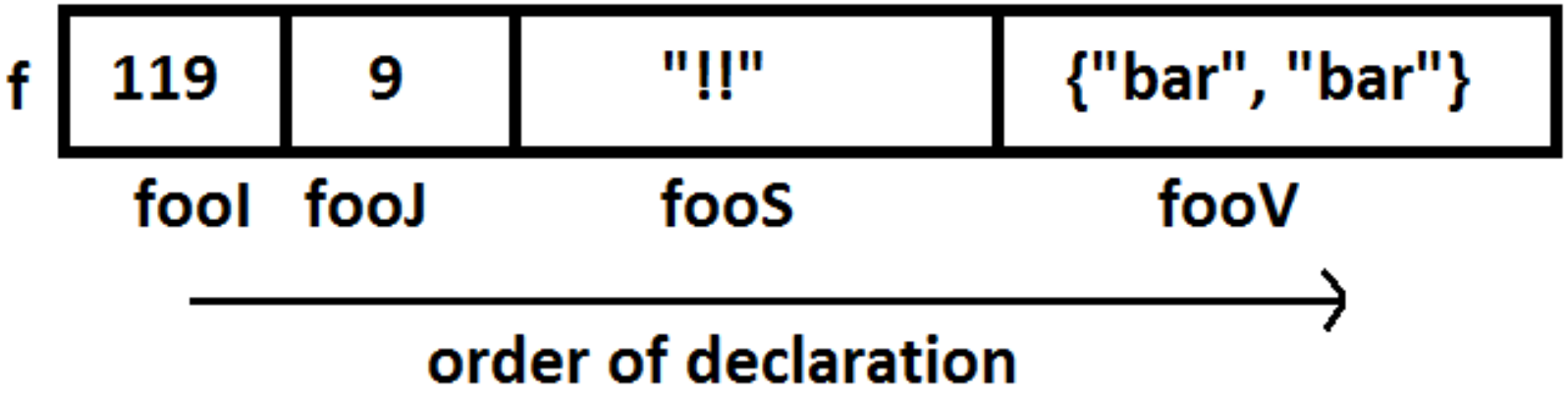

```
foo f("bar", 3, 2, '!');
```

s a b c

```
fooV(b,s)
```

i.e. vector<string>(2,"bar")

Defining Constructors




```
foo f("bar", 3, 2, '!');
```

s a b c

```
{
  fooI = 119;
}
```

Defining Constructors

The member variables are declared in the order: **fooI**, **fooJ**, **fooS**, and **fooV**. Thus, despite the order provided in the initialization list, this must be the order the variables are set!

- First, **fooI** is set to **a*a == 3*3**;
- second, **fooJ** is set to the value of **fooI == 9**;
- third, **fooS** is set to **string(2, '!') == "!!"**; and
- fourth, **fooV** is set to **vector<string>(2, "bar") == { "bar", "bar"}**.
- Finally, the body is entered and **fooI** is set to **119**.

Constructor Initializer Lists vs Setting Variables In-Body

Some sources may suggest setting the member variables inside of the constructor body.

```
class BadExample {
public:
    BadExample( const std::string& in ) {
        s = in;
    }

    void print() const {
        std::cout << s;
    }

private:
    std::string s;
};
```

Constructor Initializer Lists vs Setting Variables In-Body

This is not recommend in terms of efficiency.

All member variables have to be initialized before the constructor body anyway! All class member variables, if not initialized by the programmer are default constructed before the constructor body.

In the case of **BadExample**, the member **s** will first be default initialized before the body; then, this value will be overwritten with **in**.

It would be more efficient to set **s** to **in** directly before the body.

Sometimes code must invariably be placed inside the constructor body, but that should be kept to a minimum.

Constructor Initializer Lists vs Setting Variables In-Body

```
class MoreEfficient {
public:
    MoreEfficient( const std::string& in ) : s(in) { }

    void print() const {
        std::cout << s;
    }

private:
    std::string s;
};
```

Default Constructors

A **default constructor** is a constructor that takes no input arguments and initializes the members of a class.

If we **do not write any constructors at all** for a class, the **compiler will generate a public default constructor for us**; it will **default initialize all numeric types** (they will have unspecified values) and it will **initialize all class member variables by calling their default constructors**.

Default Constructors

If we **write any constructor at all, even if it is not a default constructor**, the **compiler will not generate any default constructor for us** and our class may fail to have a default constructor unless we write one ourselves!

If the compiler attempts to generate a default constructor for us but one of the class member variables lacks a default constructor, this will generate a compiler error!

Placement of Class Member Declaration and Definitions

There are different styles for where to place the class member definitions.

We will consider four options here.

If we define all member functions inside the class interface we could then

1. place the class interface in a single header file
2. place the class interface above **int main()**.

On the other hand, we could declare some or all member functions within the class interface and define them outside. Then we could

3. place the class interface in a header file and define the member functions in a separate **.cpp** file
4. place the class interface above **int main()** and define the member functions below the main routine.

Placement of Class Member Declaration and Definitions

Foo.h

```
#ifndef __FOO__
#define __FOO__
struct Foo { void bar() const { } }; // everything defined here
#endif
```

main.cpp

```
#include "Foo.h"
int main() {
    Foo().bar(); return 0;
}
```

Placement of Class Member Declaration and Definitions

main.cpp

```
struct Foo { void bar() const { } }; // everything defined here

int main() {
    Foo().bar(); return 0;
}
```

Placement of Class Member Declaration and Definitions

Foo.h

```
#ifndef __FOO__
#define __FOO__
struct Foo { void bar() const; }; // only declared member
#endif
```

Foo.cpp

```
#include "Foo.h"
void Foo::bar() const { } // define member here
```

main.cpp

```
#include "Foo.h"
int main() {
    Foo().bar(); return 0;
}
```

Placement of Class Member Declaration and Definitions

main.cpp

```
struct Foo { void bar() const; }; // member only declared

int main() {
    Foo().bar(); return 0;
}

void Foo::bar() const { } // member defined here
```

Const Correctness

Whenever a member function of a class does not modify the values of its member variables, that member must be listed as **const**.

As with const correctness for ordinary functions, being const correct can make tracing bugs and errors much easier because one knows what functions can modify their arguments, etc.

Const Correctness

For ordinary functions, if an input argument is taken by reference (not const reference), the compiler assumes is that the function will modify its input argument.

For member functions, if they are not listed as **const**, the compiler assumes the member variables may be changed.

Coders adhering to const correctness cannot work with code from those who do not follow this practice.

Const Correctness

```
/* programmer did not pay attention to detail: FAILED to follow const
correctness */
struct NotConstCorrect {
    int i;
    // this function SHOULD be listed as const!
    void printTheInt() {
        std::cout << i;
    }
};
```

Another programmer writes a function that invokes printTheInt, knowing that the function does not modify its argument and only prints the value: they make the argument const...

```
void displayInt( const NotConstCorrect& n) {
    /* ERROR: cannot call member function because n is const and
        the member function is not const! */
    n.printTheInt();
}
```

Const Correctness

The programmer following const correctness is unable to use the member function, even though they know it should not modify its argument.

On the other hand, for a more complicated member function, a user of the class might avoid using that member function entirely because they believe it could modify their variable in a way they do not want.

It is useful to glance at an interface and recognize which member functions may modify the class member variables and which may not.

Const Correctness

When writing a member function, **always ask: can/does this function modify the member variables? If the answer is no, then make sure it is const!**

```
/* now const correct */
struct ConstCorrect {
    int i;
    // this function IS listed as const!
    void printTheInt() const {
        std::cout << i;
    }
};
```

Writing a Header File

A header file should always begin with:

```
#ifndef SOME_NAME
#define SOME_NAME
```

and end with

```
#endif
```

Recall we pick **SOME_NAME** to be a variable name that is unlikely to ever be used in the program. It usually has something to do with the classes or functions we are declaring.

This protects against multiple definitions of a class in a single **.cpp** file, which would be a compiler error.

Writing a Header File

Recall that in a header file, we should not specify any namespaces.

Any **.cpp** files that are supposed to define class member functions or **.cpp** files that need to know of their existence should **#include** the header file.

Only header files should be **#include**d; a **.cpp** file should not be.

Direct vs Indirect Initialization

We are now going to look at different syntaxes for defining and initializing class variables. While in many cases the end results can be the same, they are not necessarily equivalent.

Consider something familiar in creating **std::string**s. Note that a **std::string** has a constructor that accepts a **const char*** argument (string literal).

```
string s1("cat");
string s2 = string("cat");
string s3 = "cat";
```

These are all a little different. The first is **direct initialization**; the latter two are **copy initialization**, recognized by the presence of the **=** sign.

Direct vs Indirect Initialization

```
string s1("cat");
```

The code above **directly** invokes **std::string::string(const char *)** and **s1** is directly constructed through this call to the constructor.

Direct initialization is efficient.

Direct vs Indirect Initialization


```
string s1("cat");
```

Direct vs Indirect Initialization

```
string s2 = string("cat");
```

The code above uses **copy initialization**. It

- (i) requests that a temporary **std::string** object (an R-value) be constructed, **string("cat")**.
- (ii) Then it checks if it "has permission" to use this temporary object to make a new string variable.
- (iii) If it has permission, the data from this temporary object are to be harvested and used to construct **s2**.
- (iv) Finally, the temporary **std::string** object is destroyed.

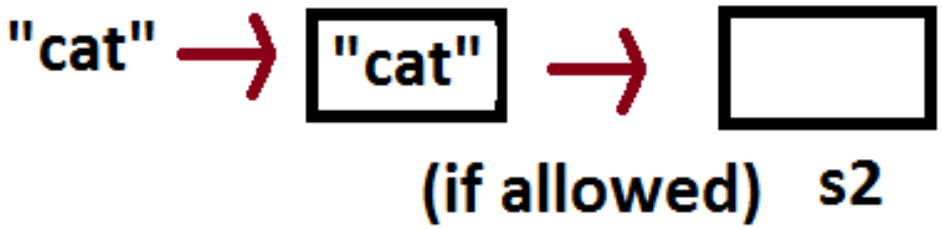
This is inefficient!

Remark: recall that a constructor returns an object, so **string("cat")** returns a **std::string** object.

Direct vs Indirect Initializaton


```
string s2 = string("cat");
```

Direct vs Indirect Initializaton




```
string s2 = string("cat");
```

Direct vs Indirect Initializaton


```
string s2 = string("cat");
```

Direct vs Indirect Initialization

```
string s3 = "cat";
```

The code above also uses **copy initialization**. It

- (i) determines if it “has permission” to convert the string literal into a temporary string object, **string("cat")**.
- (ii) If it does, it constructs the temporary object **string("cat")**.
- (iii) Then it checks if it “has permission” to use this temporary object to make a new **std::string** variable.
- (iv) If it has permission, the data from this temporary object are to be harvested and used to construct **s3**.
- (v) Finally, the temporary **std::string** object is destroyed.

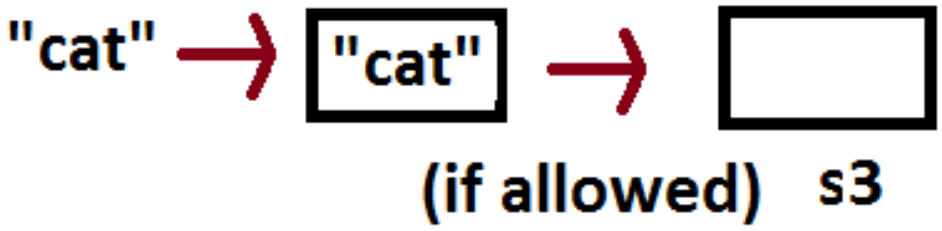
This is also inefficient!

Direct vs Indirect Initializaton

(if allowed)

```
string s3 = "cat";
```

Direct vs Indirect Initializaton




```
string s3 = "cat";
```

Direct vs Indirect Initializaton


```
string s3 = "cat";
```

Direct vs Indirect Initialization

Despite the inefficiency of what the code directly means, as part of the new C++ standard, provided the compiler has permission to perform various conversions, **copy elision** can lead to these extra objects and steps being optimized away.

Fundamentally, though, the three statements for creating a **std::string** variable are all different and it is incorrect to say they are all the same.

Direct vs Indirect Initialization

The same principles apply to all class objects of the C++ Standard and those we write.

Just because the resulting object is the same at the end does not mean the process of creating it is the same.

If some conversions are not allowed in the copy initialization cases, direct initialization may be the only valid way to construct an object!

Masking of Variable Names and this

One should beware that constructor parameter names mask the member variable names within the constructor body; likewise, creating a local variable in a constructor (or other member function) with the same name as a member variable masks that member variable.

In general, one should avoid naming local variables the same as member variables.

Masking of Variable Names and this

```
struct Foo {
    Foo(int i) {
        i = i; // does nothing: input parameter i assigned to itself
    }
    int i;
};
```

Besides the inefficiency above in not using a constructor initializer list, the member variable **i** is not properly initialized because **i** in the constructor and its body only refers to the local variable!

Foo.i is not initialized.

Masking of Variable Names and this

```
struct Bar {
    Bar() {
        int i = 44; // local int i, not Bar.i
    }
    int i;
};
```

Above, **Bar.i** is not set to 44; we created a local variable **i**, setting it equal to 44. After the constructor body exits, that local **i** is destroyed.

Masking of Variable Names and this

Recall: if **ptr** is a pointer to a class object then both

```
ptr->memberName; // arrow access syntax
(*ptr).memberName; // dereference and access member
```

refer to the member **memberName** of the object **ptr** points to.

Masking of Variable Names and this

Every class has a special pointer called **this** that points to that class in memory.

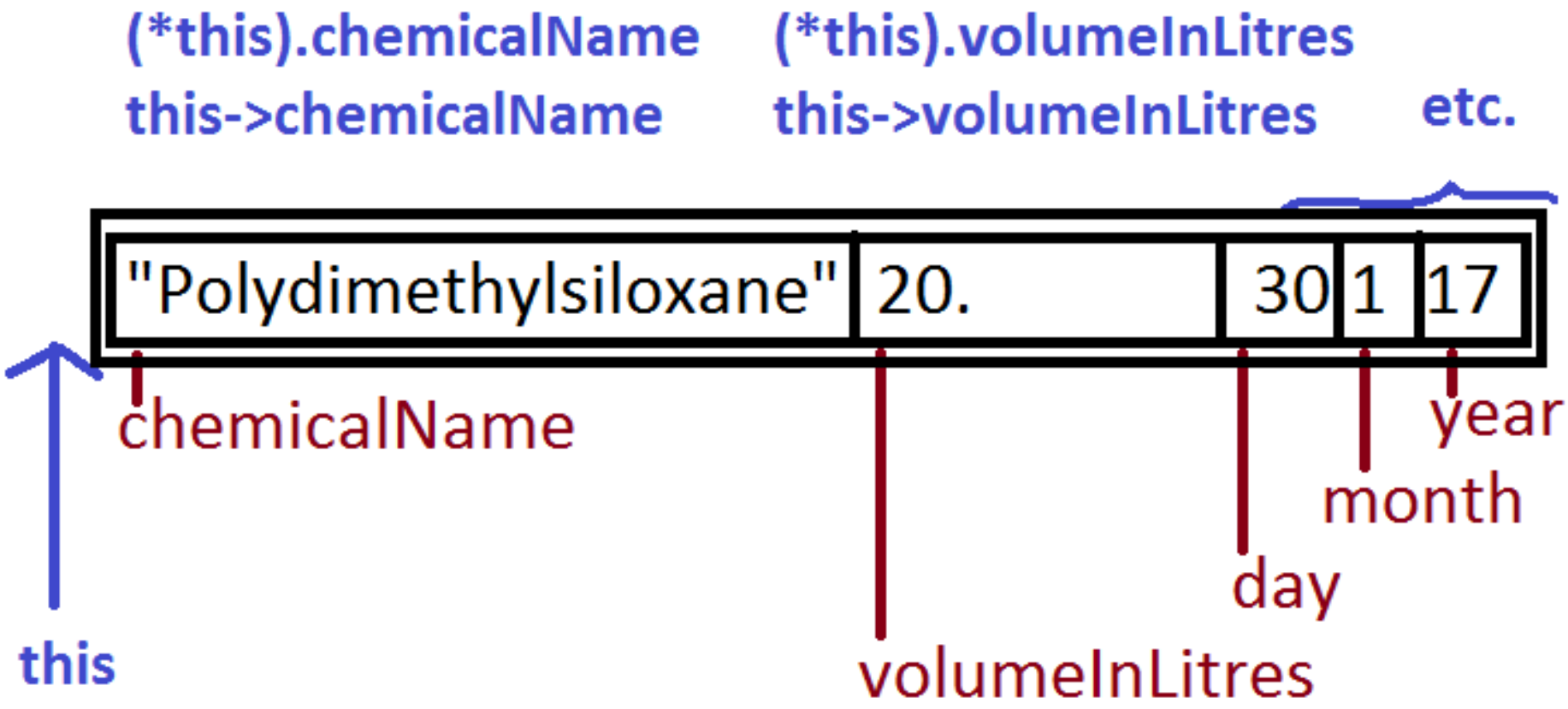
We can then write **this->memberName** or **(*this).memberName** to refer explicitly to the member variable called **memberName** of a given class, without the fear of masking. For example,

```
this->i = i;
```

would have set the member **Foo.i** to the local variable **i**, despite masking.

It is still better to choose a different local variable name than the member names!

Masking of Variable Names and this



Masking of Variable Names and this

When writing constructor initializers, masking is not a concern for the parts appearing outside of the constructor body; however, we should still avoid choosing the same names if possible.

```
struct Foo {
   Foo(int i) : i(i) { }
   int i;
};
```

This is confusing, but in general the variables named outside the parentheses refer to the member variables of the class and the variables appearing within the parentheses refer to the constructor parameters.

Masking of Variable Names and this

It would be better to have chosen different names.

```
struct Foo {
    Foo(int _i) : i(_i) { }
    int i;
};
```

Notice the input parameter is called **_i** while the local variable is called **i**.

More on Default Values

In C++11 and beyond, it is possible to give member variables a default value.

This default value will be assigned to the variable during the constructor initialization phase if a value is not otherwise provided.

More on Default Values

```
struct Boo {
    int i = 111; // default value for i is 111 if not specified
    int j = 1024;
    Boo(int input) : j (input) {
        cout << i << j;
    }
};
```

Consider the code and output below

```
Boo x(3);
```

```
111 3
```

We will have **x.i == 111** because there is no initialization value given in the construtor initializer but 111 is a default value, and **x.j == 3** as it was set explicitly and the default value is ignored.

Operator Overloading

We will take a brief look into a very useful and powerful feature of the C++ language: **operator overloading**.

Operator overloading is the technique of giving new meanings to C++ operators that were not present before.

Operator overloading can only be done when one of its operands (input arguments) is of class type. We cannot, for example, redefine the meaning of + between two **int**s, etc.

Operator Overloading

operator<: this operator by its natural definition, only compares two numeric types.

```
int a = 3, b = 4;
a < b; // true because 3 < 4
```

With the **std::string** class, two **std::string**s can be compared lexicographically:

```
string x("x"), y("y");
x < y; // true
```

Using **<** to compare two **std::string**s is an example of operator overloading: someone decided that would be a reasonable thing to define, beyond its normal comparison of numeric types.

Operator Overloading

operator==: this operator by its natural definition, only compares two numeric types (including pointers) to determine if they are equal.

```
int x = 4, y = 11;
int *xp = &x;
int *yp = &y;
xp == yp; // false: they point to different addresses
```

Two **std::vector**s storing the same data type can be compared as well to see if they store the same data.

```
vector<bool> vbool1 = { true, true, false };
vector<bool> vbool2 = { true, true, true };
vbool1 == vbool2; // false
```

Someone gave extra meaning to compare two **std::vector**s and as a result of this operator overload, we can compare two vectors with **==**.

Operator Overloading

operator<<: the left bit shift operator. In its natural form, it shifts the binary bits left for an integer type (with 0's moving in from the right):

```
int x = 7; // in binary: ... 000111
// <<is a left bit-shift operator, shifting all the bits by a given amount
x = (x <<2); // now in binary it is 011100 and x is 28
```

Of course, we also use **operator<<** in the context of printing stuff to the console:

```
cout << "Hello!";
```

Operator Overloading I

Let's consider a **Book** class:

```
class Book {
private:
   std::string title; // book title
   double price; // price in dollars

public:
   /**
   Constructor: creates a Book from a given title and price
   @param _title the title of the book
   @param _price the book's price
   */
   Book( const std::string& _title, double _price) : title(_title),
      price(_price) { }
```

Operator Overloading II

```
/**
Returns the title of the book
@return the book title
*/
const std::string& getTitle() const {
    return title;
}

/**
Returns the price of the book
@return the price in dollars
*/
double getPrice() const {
    return price;
}
```

Operator Overloading III

```
    /**
    Allows the book price to be updated
    @param newPrice the updated price in dollars for the book
    */
    void setPrice(double newPrice) {
        price = newPrice;
    }

};
```

Note all member functions and constructors (in this case just 1) were defined within the class.

Operator Overloading

It might be nice to add our own functionality to how we use **Book** objects...

```
Book cppBook("C++11 Primer", 69.99); // great C++11 reference book
// classic PDE book
Book mathBook("Partial Differential Equations", 70.06);
Book fluidsBook("Fluid Mechanics", 55.36); // intro fluids book

// be able to "cout" a book!
cout << cppBook <<endl << mathBook << fluidsBook << endl;
```

```
C++11 Primer, $69.99
Partial Differential Equations, $70.06
Fluid Mechanics, $55.36
```

Operator Overloading

We may wish to have **operator<** defined for **Book**s to they can be sorted by title (alphabetically)

```
// add the Books to a vector
vector<Book> bookList = { cppBook, mathBook, fluidsBook };

// sort the vector of books by title: the sort function requires a comparison operator!
sort( bookList.begin(), bookList.end() );

for ( const auto& book : bookList ) {
    cout << book << endl;
}
```

```
C++11 Primer, $69.99
Fluid Mechanics, $55.36
Partial Differential Equations, $70.06
```

Operator Overloading

We may wish to have **operator==** defined so we can compare two **Book**s, where equality requires the same title and price...

```
// check if the second book in the vector is the fluid book
if ( bookList[1] == fluidsBook ) {
    cout <<"same!"<< endl;
}
```

```
same!
```

Operator Overloading

Although it may seem magical, an operator is just a function! We will consider implementations for **==**, **<**, and **<<** that we would like to have for **Book** objects.

Basically, we wish to define functions with appropriate inputs and outputs with the names **operator==**, **operator<**, and **operator<<**.

Operator Overloading

Operator overloading may require granting special access privileges to private members of a class or making the operators member functions; in this introductory overview, however, this will not be necessary.

Operator Overloading

The purpose of **operator==** is to tell us whether two **Book**s are equal in both title and price, hence the output should be **bool**.

The operator needs to compare two **Book** objects, neither of which should be changed, and hence the arguments should both be **const Book&**.

We need to define:

```
/**
Operator compares two Books to determine if they are equal
@param book1 the first book
@param book2 the second book
@return whether they are equal in both price and title
*/
bool operator==(const Book& book1, const Book& book2);
```

Operator Overloading

If **b1** and **b2** are **Book** objects, by writing

```
b1==b2
```

we are actually invoking

```
operator==(b1, b2);
```

Thus, **b1** is the first argument and **b2** is the second argument.

Operator Overloading

We define

```
bool operator==(const Book& book1, const Book& book2) {
    // both titles and prices must be the same
    return ( book1.getTitle() == book2.getTitle() ) &&
        ( book1.getPrice() == book2.getPrice() );
}
```

Operator Overloading

We can likewise define

```
bool operator<(const Book& book1, const Book& book2) {
    // comparison is by title
    return book1.getTitle() < book2.getTitle();
}
```

Similar to **operator==**, the lines of code below are equivalent:

```
b1 < b2;
operator<(b1, b2);
```

Operator Overloading

Defining **operator<<** is a little special because we want to preserve the ability of chaining **std::cout** statements together.
There are a few items to be aware of.

- **std::cout** is an **std::ostream** object;
- **std::ostream** objects cannot be copied and must always be passed and returned by reference;
- the process of displaying information to the console modifies the internal state of **std::cout**;
- to properly chain statements together, the output of every **<<** operation must be **std::cout** itself.

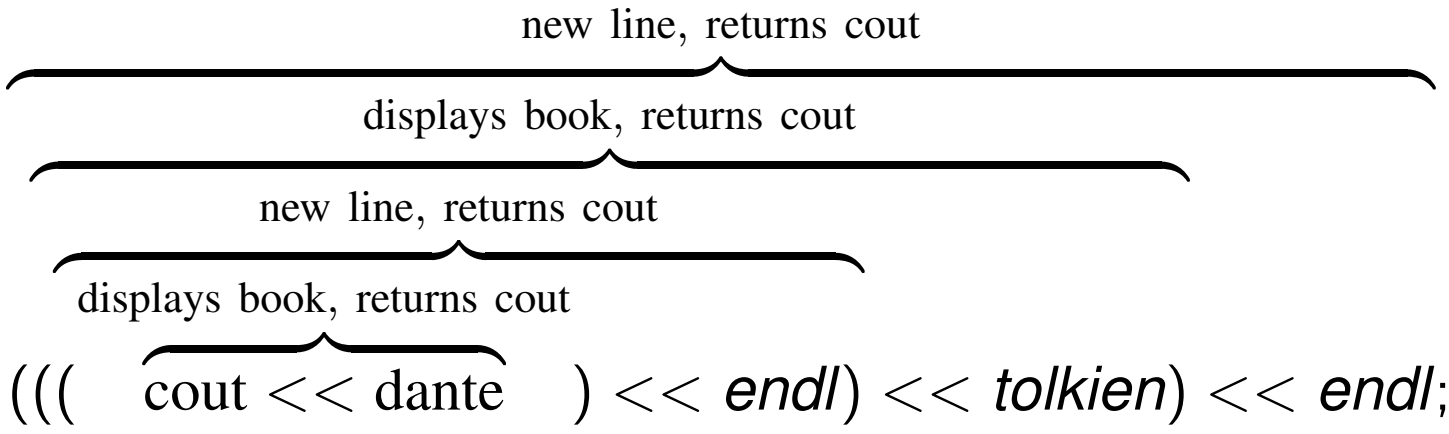
Operator Overloading

The secret behind chaining **std::cout** statements...

```
Book dante("The Inferno", 5);
Book tolkien("The Silmarillion", 11.09);

cout << dante << endl << tolkien << endl;
```

Operations with **operator<<** are read from left to right as:



Operator Overloading

- First, **cout << dante** is processed: this displays the book information to the screen and returns **cout**;
- second, **cout <<endl** is processed: this produces a new line and returns **cout**;
- third, **cout << tolkien** is processed: this displays the book information to the screen and returns **cout**;
- fourth, **cout << endl** is processed; this produces a new line and returns **cout**;
- finally **cout** is left alone on the line and nothing more needs to happen.

Operator Overloading

To implement the operator, we have

```
ostream& operator<<( ostream& out, const Book& toPrint ) {
   // locally we refer to the stream as out

   // write the title, add price formatting, and display the price
   out << toPrint.getTitle() << ", $"<< toPrint.getPrice();

   // return the stream
   return out;
}
```

The **std::ostream** enters by reference and is returned by reference! The **Book** object does not change, so it is passed as a reference to const.

Operator Overloading

Once again: an operator is just a function. In lieu of

```
cout << dante;
```

We could have written

```
operator<<(cout, dante);
```

Operator Overloading

The same chaining mechanism applies to reading in data through a process like **cin >>...**

std::cin is a **std::istream** object, which also must always be passed and returned by reference and which is modified internally by the reading process.

Summary

- A **class** allows for related data and functions to be grouped and encapsulated.
- A class interface can have **public** and **private** sections; the public sections are accessible by all users of the class with the **.**; the private section can generally not be accessed outside of the class.
- A **class** and **struct** are the same except for their default level of access.
- Member variables belong to a class and are stored next to each other forming the class' memory layout.
- Member functions may require a **const** modifier if they are accessors.
- A constructor creates a class object; all members are initialized during the constructor initializer and then modified further in the constructor body.
- In class member functions, member functions and variables can be referred to by name, unless **masking** applies.

Summary

- In general, the most direct way to construct a class is through **direct construction** as opposed to **copy initialization**.
- Constructor/function parameters and local variables can make member variable names.
- Operator overloading allows us to give new meaning to operators by defining functions with suitable arguments called **operator<<**, **operator<**, **operator>**, **operator>>**, etc.