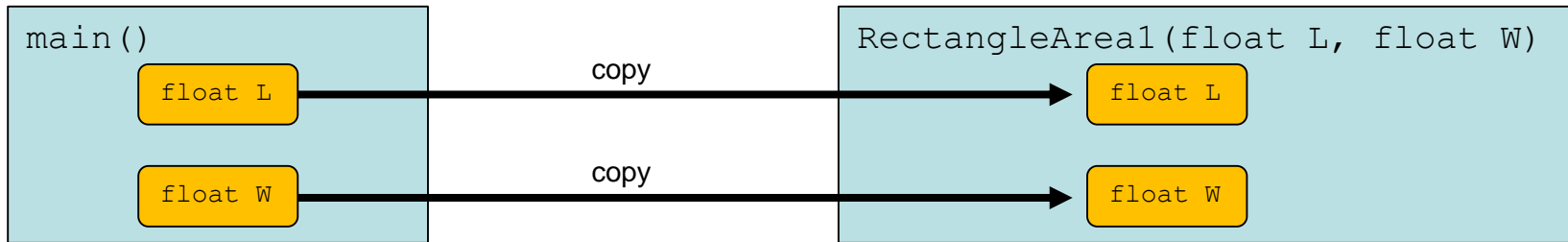


Tutorial Outline

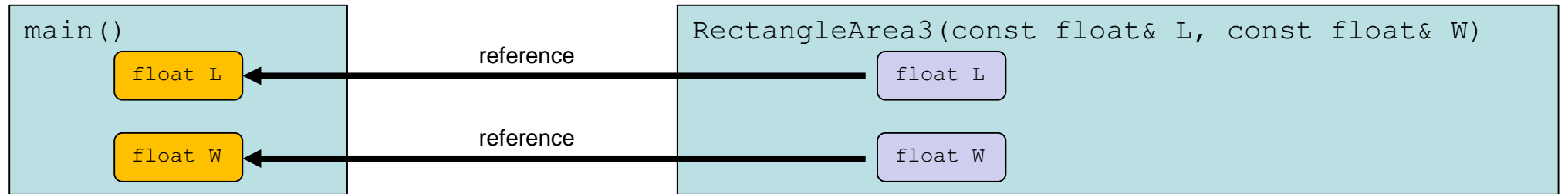
- Pass by value and references
- C++ Classes
- Constructor and copy constructor
- Encapsulation
- Exercises

Pass by Value



- C++ defaults to *pass by value* behavior when calling a function.
- The function arguments are **copied** when used in the function.
- Changing the value of L or W in the RectangleArea1 function does **not** effect their original values in the main() function
- When passing objects as function arguments it is important to be aware that potentially large data structures are automatically copied!

Pass by Reference



- *Pass by reference* behavior is triggered when the `&` character is used to modify the type of the argument.
- Pass by reference function arguments are **NOT** copied. Instead the compiler sends a *pointer* to the function that references the memory location of the original variable. The syntax of using the argument in the function does not change.
- Pass by reference arguments almost always act just like a pass by value argument when writing code **EXCEPT** that changing their value changes the value of the original variable!!
- The *const* modifier can be used to prevent changes to the original variable in `main()`.

void does not return a value.



```
void RectangleArea4(const float& L, const float& W, float& area) {  
    area= L*W ;  
}
```

- In RectangleArea4 the pass by reference behavior is used as a way to return the result without the function returning a value.
- The value of the *area* argument is modified in the main() routine by the function.
- This can be a useful way for a function to return multiple values in the calling routine.

- In C++ arguments to functions can be objects...which can contain any quantity of data you've defined!
 - Example: Consider a string variable containing 1 million characters (approx. 1 MB of RAM).
 - Pass by value requires a copy – 1 MB.
 - Pass by reference requires 8 bytes!
- Pass by value could potentially mean the accidental copying of large amounts of memory which can greatly impact program memory usage and performance.
- When passing by reference, use the *const* modifier whenever appropriate to protect yourself from coding errors.
 - Generally speaking – use *const* anytime you don't want to modify function arguments in a function.

“C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.” – Bjarne Stroustrup

A first C++ class

- You can start a new project in C::B or just modify the Hello World! code.
- In the main.cpp, we'll define a class called BasicRectangle
- First, just the basics: length and width
- Enter the code on the right before the main() function in the main.cpp file (copy & paste is fine) and create a BasicRectangle object in main.cpp:

```
#include <iostream>

using namespace std;

class BasicRectangle
{
public:
    // width ;
    float W ;
    // length
    float L ;
};

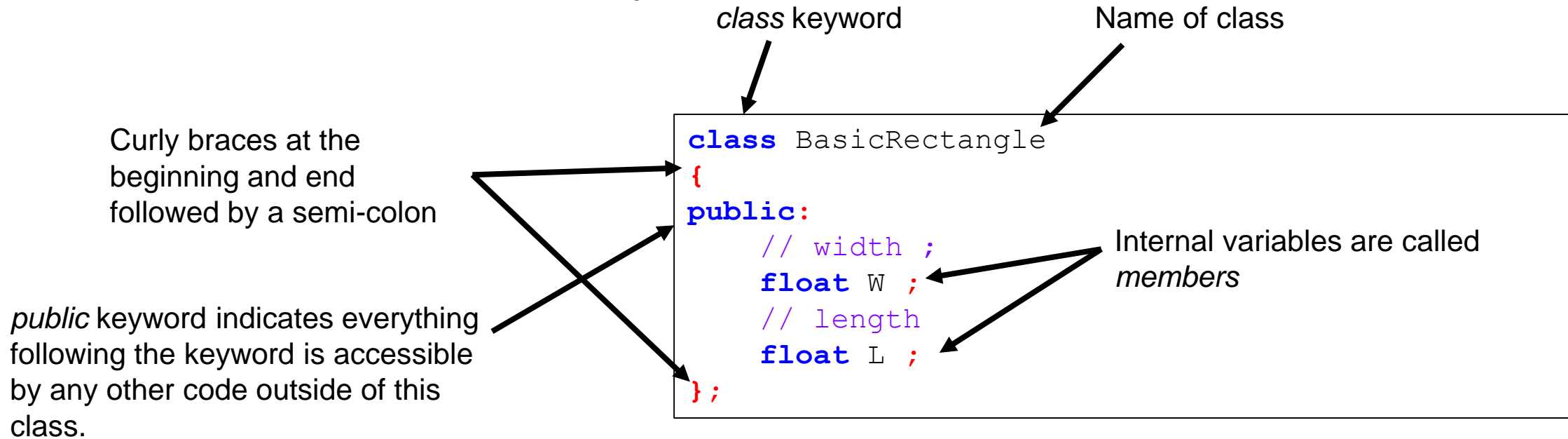
int main()
{
    cout << "Hello world!" << endl;

    BasicRectangle rectangle ;
    rectangle.W = 1.0 ;
    rectangle.L = 2.0 ;

    return 0;
}
```



Basic C++ Class Syntax



The class can now be used to declare an object named *rectangle*. The width and length of the rectangle can be set.

```
BasicRectangle rectangle ;
rectangle.W = 1.0 ;
rectangle.L = 2.0 ;
```

Accessing data in the class

- Public members in an object can be accessed (for reading or writing) with the syntax:

object.member



```
int main()  
{  
    cout << "Hello world!" << endl;  
  
    BasicRectangle rectangle ;  
    rectangle.W = 1.0 ;  
    rectangle.L = 2.0 ;  
  
    return 0;  
}
```

- Next let's add a function inside the object (called a *method*) to calculate the area.

method Area does not take any arguments, it just returns the calculation based on the object members.

```
class BasicRectangle
{
public:
    // width ;
    float W ;
    // length
    float L ;
    float Area() {
        return W * L ;
    }
};

int main()
{
    cout << "Hello world!" << endl;

    BasicRectangle rectangle ;
    rectangle.W = 21.0 ;
    rectangle.L = 2.0 ;

    cout << rectangle.Area() << endl ;

    return 0;
}
```

Methods are accessed just like members:
object.method(arguments)

Now for a “real” class

- Defining a class in the main.cpp file is not typical.
- Two parts to a C++ class:
 - Header file (my_class.h)
 - Contains the interface (definition) of the class – members, methods, etc.
 - The interface is used by the compiler for type checking, enforcing access to private or protected data, and so on.
 - Also useful for programmers when *using* a class – no need to read the source code, just rely on the interface.
 - Source file (my_class.cc)
 - Compiled by the compiler.
 - Contains implementation of methods, initialization of members.
 - In some circumstances there is no source file to go with a header file.

rectangle.h

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
    public:
        Rectangle();
        virtual ~Rectangle();

    protected:

    private:
};

#endif // RECTANGLE_H
```

rectangle.cpp

```
#include "rectangle.h"

Rectangle::Rectangle()
{
    //ctor
}

Rectangle::~~Rectangle()
{
    //dtor
}
```

- 2 files are automatically generated: rectangle.h and rectangle.h.cpp

Modify rectangle.h

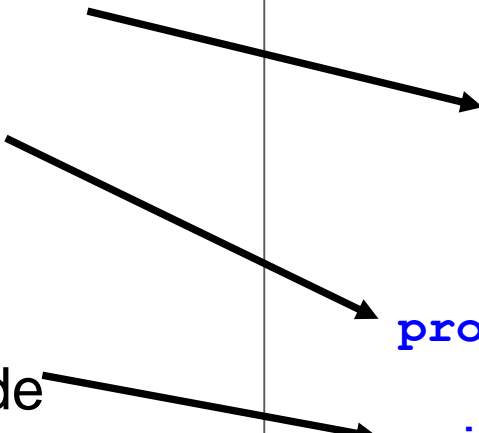
- As in the sample *BasicRectangle*, add storage for the length and width to the header file. Add a *declaration* for the Area method.
- The *protected* keyword will be discussed later.
- The *private* keyword declares anything following it (members, methods) to be visible only to code **in this class**.

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
    public:
        Rectangle();
        virtual ~Rectangle();

        float m_length ;
        float m_width ;

        float Area() ;
    protected:
    private:
};
#endif // RECTANGLE_H
```

A diagram consisting of three black arrows pointing from the list items on the left to specific parts of the C++ code on the right. The first arrow points from the first list item to the `float m_length ;` and `float m_width ;` lines. The second arrow points from the second list item to the `protected:` keyword. The third arrow points from the third list item to the `private:` keyword.

rectangle.cpp

- The Area() method now has a basic definition added.
- The syntax:

class::method


tells the compiler that this is the code for the Area() method declared in rectangle.h

- Now take a few minutes to fill in the code for Area().
 - Hint – look at the code used in BasicRectangle...

```
#include "rectangle.h"
```

```
Rectangle::Rectangle()  
{  
    //ctor  
}
```

```
Rectangle::~~Rectangle()  
{  
    //dctor  
}
```

```
float Rectangle::Area()  
{  
      
}
```

Copy constructor

- A copy constructor is a member function which initializes an object using another object of the same class. A copy constructor has the following general function prototype:

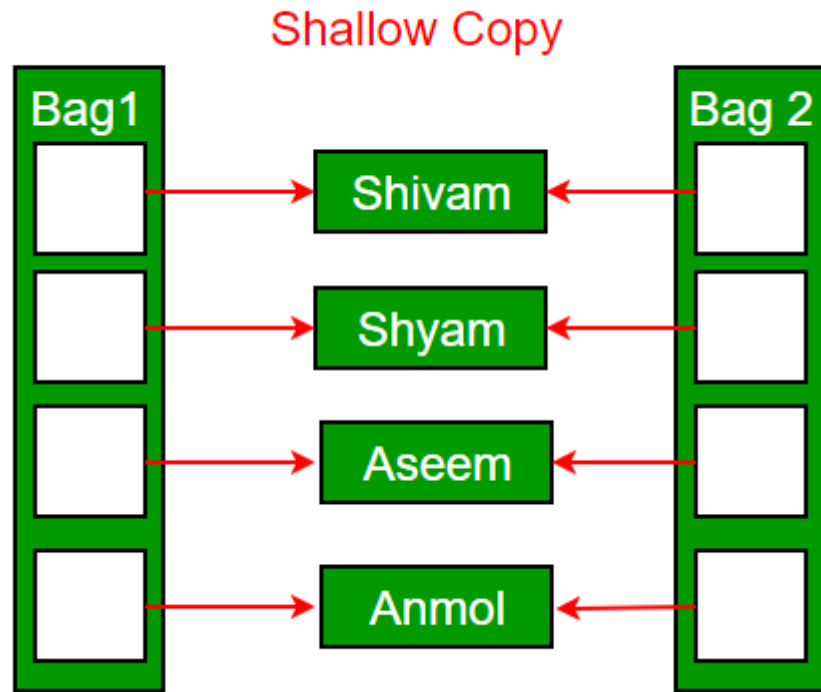
```
ClassName (const ClassName &old_obj);
```

- The **copy constructor** is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. The copy constructor is used to –
 - Initialize one object from another of the same type.
 - Copy an object to pass it as an argument to a function.
 - Copy an object to return it from a function.
- If a copy constructor is not defined in a class, the compiler itself defines one. If the class has pointer variables and has some dynamic memory allocations, then it is a must to have a copy constructor.

When is copy constructor called?

- In C++, a Copy Constructor may be called in following cases:
 1. When an object of the class is returned by value.
 2. When an object of the class is passed (to a function) by value as an argument.
 3. When an object is constructed based on another object of the same class.
 4. When the compiler generates a temporary object.*
- It is, however, not guaranteed that a copy constructor will be called in all these cases, because the C++ Standard allows the compiler to optimize the copy away in certain cases.

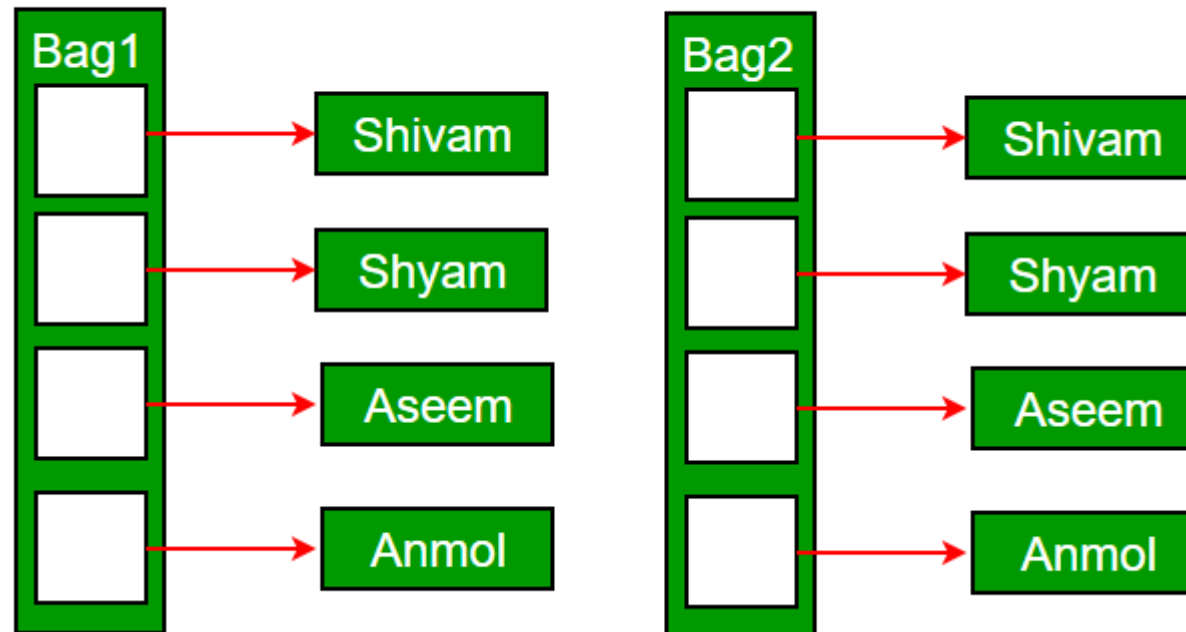
Default constructor does only shallow copy.



Deep copy is possible only with user defined copy constructor.

- In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.

Deep Copy



Shallow vs Deep Copies

- A *shallow copy* of an object copies all of the member field values. This works well if the fields are values, but may not be what you want for fields that point to dynamically allocated memory. The pointer will be copied, but the memory it points to will not be copied -- the field in both the original object and the copy will then point to the same dynamically allocated memory, which is not usually what you want. The default copy constructor and assignment operator make shallow copies.
- A *deep copy* copies all fields, *and* makes copies of dynamically allocated memory pointed to by the fields. To make a deep copy, you must write a copy constructor and overload the assignment operator, otherwise the copy will point to the original, with disastrous consequences.

Exercise 1

```
01. #include<iostream>
02. using namespace std;
03.
04. class Point
05. {
06. private:
07.     int x, y;
08. public:
09.     Point(int x1, int y1) { x = x1; y = y1; }
10.
11.     // Copy constructor
12.     Point(Point& p) { x = p.x; y = p.y; }
13.
14.     int getX()          { return x; }
15.     int getY()          { return y; }
16. };
17.
18. int main()
19. {
20.     Point p1(10, 15); // Normal constructor is called here
21.     Point p2(p1);     // Copy constructor is called here
22.
23.     // Let us access values assigned by constructors
24.     cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
25.     cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
26.
27.     return 0;
28. }
```

Exercise 2

```
01. #include <iostream>
02.
03. using namespace std;
04.
05. class Line {
06.
07.     public:
08.         int getLength( void );
09.         Line( int len );           // simple constructor
10.         Line( const Line &obj);    // copy constructor
11.         ~Line();                  // destructor
12.
13.     private:
14.         int *ptr;
15. };
16.
17. // Member functions definitions including constructor
18. Line::Line(int len) {
19.     cout << "Normal constructor allocating ptr" << endl;
20.
21.     // allocate memory for the pointer;
22.     ptr = new int;
23.     *ptr = len;
24. }
25.
26. Line::Line(const Line &obj) {
27.     cout << "Copy constructor allocating ptr." << endl;
28.     ptr = new int;
29.     *ptr = *obj.ptr; // copy the value
30. }
31.
32. Line::~~Line(void) {
33.     cout << "Freeing memory!" << endl;
34.     delete ptr;
35. }
36.
37. int Line::getLength( void ) {
38.     return *ptr;
39. }
40.
41. void display(Line obj) {
42.     cout << "Length of line : " << obj.getLength() << endl;
43. }
44.
45. // Main function for the program
46. int main() {
47.     Line line(10);
48.
49.     display(line);
50.
51.     return 0;
52. }
```

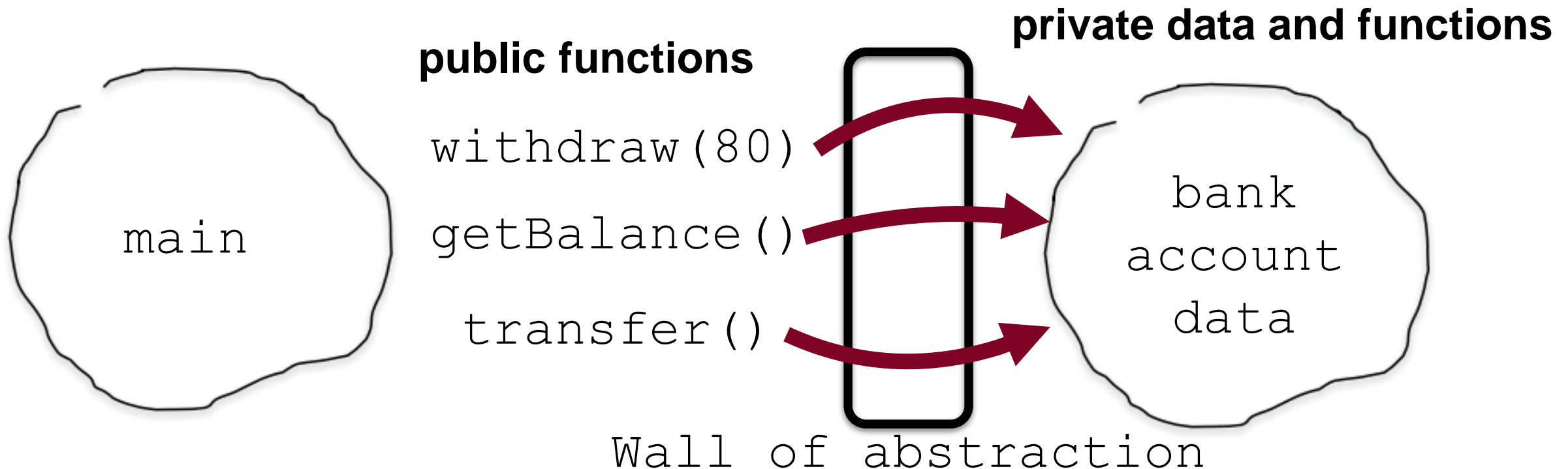
The Need for New Types

- A C++ "class" is simply a very-slightly modified struct (details to follow).
- As with structs, we sometimes want new types:
 - A calendar program might want to store information about dates, but C++ does not have a Date type.
 - A student registration system needs to store info about students, but C++ has no Student type.
 - A music synthesizer app might want to store information about users' accounts, but C++ has no Instrument type.



Classes: Encapsulation

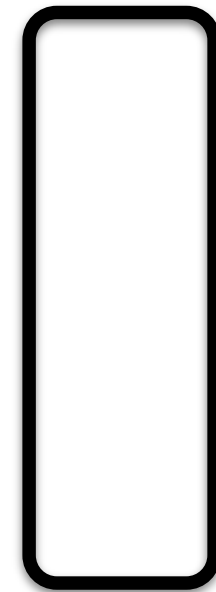
- The *only* difference between a struct and a class is the notion of defaults regarding *encapsulation*. A struct defaults to "public" members, and a class defaults to "private" members.
- "Encapsulation" allows the designer of a class to build a "wall of abstraction" around her data:



Classes: Encapsulation

- The *only* difference between a struct and a class is the notion of defaults regarding *encapsulation*. A struct defaults to "public" members, and a class defaults to "private" members.
- "Encapsulation" allows the designer of a class to build a "wall of abstraction" around her data:

```
int main() {  
    BankAccount checking("Bob", 42);  
    checking.withdraw(80);  
    cout << checking.getBalance() << endl;  
}
```



private data and functions



bank
account
data

Wall of abstraction

Classes: Encapsulation

- The *only* difference between a struct and a class is the notion of defaults regarding *encapsulation*. A struct defaults to "public" members, and a class defaults to "private" members.
- "Encapsulation" allows the designer of a class to build a "wall of abstraction" around her data:

```
int main() {  
    BankAccount checking("Bob", 42);  
    checking.withdraw(80);  
    cout << checking.getBalance() << endl;  
}
```

`withdraw(80)`

`error: not enough funds!`

private data and functions



Wall of abstraction

Classes: Encapsulation

- The *only* difference between a struct and a class is the notion of defaults regarding *encapsulation*. A struct defaults to "public" members, and a class defaults to "private" members.
- "Encapsulation" allows the designer of a class to build a "wall of abstraction" around her data:

```
int main() {  
    BankAccount checking("Bob", 42);  
    checking.withdraw(80);  
    cout << checking.getBalance() << endl;  
}
```

getBalance()

42

private data and functions

bank
account
data

Wall of abstraction



Classes: Encapsulation

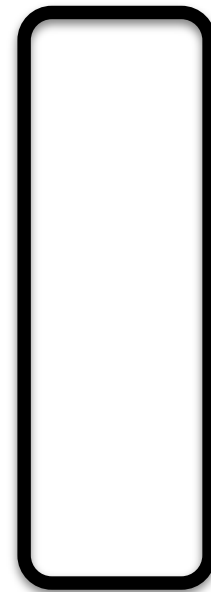
- The reason we want encapsulation is so that the end user of our class does not have direct access to the data -- we, as the class designer, control the data completely:

```
int main() {  
    BankAccount checking("Bob", 42);  
    checking.withdraw(80);  
    cout << checking.getBalance() << endl;  
    checking.balance = 81.2345;  
}
```

If we allowed this, our internal class data might not be in a state we can handle (e.g., too many decimal places for a monetary value)



private data and functions



Wall of abstraction

Classes: Encapsulation

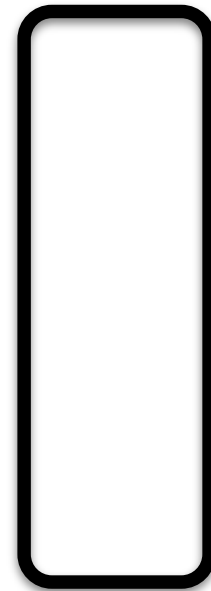
- So, we block the ability for someone using our class to directly touch the data, and we force them to go through our own functions:

```
int main() {  
    BankAccount checking("Bob", 42);  
    checking.withdraw(80);  
    cout << checking.getBalance() << endl;  
    checking.setBalance(81.2345);  
}
```



Because we control the function, we can do a check on this, and enforce the "only two decimals" limit.

private data and functions



Wall of abstraction

Elements of a Class

- **member variables:** State inside each object.
 - Also called "instance variables" or "fields"
 - Declared as private
 - Each object created has a copy of each field.
- **member functions:** Behavior that executes inside each object.
 - Also called "methods"
 - Each object created has a copy of each method.
 - The method can interact with the data inside that object.
- **constructor:** Initializes new objects as they are created.
 - Sets the initial state of each new object.
 - Often accepts parameters for the initial state of the fields.

Class Interface Divide

Interface

name.h

Client reads

Shows methods and
states instance
variables

Source

name.cpp

Implementer writes

Implements methods

Structure of a .h file

```
// classname.h
#pragma once

class ClassName {
    // class definition
};
```

This basically says, "if you see this file more than once while compiling, ignore it after the first time"
(so the compiler doesn't think you're trying to define things more than once)

Structure of a .h file

```
// classname.h
#pragma once

class ClassName {
    // class definition
};
```

This basically says, "if you see this file more than once while compiling, ignore it after the first time" (so the compiler doesn't think you're trying to define things more than once)

Older format, not as nice:

```
// classname.h
#ifndef _CLASSNAME_H
#define _CLASSNAME_H

class ClassName {
    // class definition
};

#endif
```

Structure of a .h file

```
// in ClassName.h
```

```
class ClassName {
```

```
public:
```

```
    ClassName(parameters);    // constructor
```

```
    returnType func1(parameters); // member functions
```

```
    returnType func2(parameters); // (behavior inside
```

```
    returnType func3(parameters); // each object)
```

```
private:
```

```
    type var1; // member variables
```

```
    type var2; // (data inside each object)
```

```
    type func4(); // (private function)
```

```
};
```


Encapsulation defined in .h

```
// in MyClass.h
```

```
class MyClass {
```

```
public:
```

```
    MyClass(parameters);      // constructor  
    returnType func1(parameters); // member functions  
    returnType func2(parameters); // (behavior inside  
    returnType func3(parameters); // each object)
```

```
private:
```

```
    type var1; // member variables  
    type var2; // (data inside each object)  
    type func4(); // (private function)
```

```
};
```

Any class *instance* can directly use anything defined as public (but you **never** directly call a constructor):

```
MyClass a;  
a.func1(arguments)
```

Encapsulation defined in .h

```
// in MyClass.h
class MyClass {
public:
    MyClass(parameters);    // constructor
    returnType func1(parameters); // member functions
    returnType func2(parameters); // (behavior inside
    returnType func3(parameters); // each object)
```

```
private:
    type var1; // member variables
    type var2; // (data inside each object)
    type func4(); // (private function)
};
```

Class *instances* can **not** directly use anything defined as private:

```
MyClass a;
a.var1 = 2; // error!
```

Constructors and (eventually) Destructors

```
// in MyClass.h
class MyClass {
public:
    MyClass(); // default constructor
    MyClass(parameters); // constructor
    ...
};
```

When a class instance is created, we say that it is "constructed":

```
string s1; // uses default constructor
```

```
string s2("I'm a string"); // uses a constructor
                        // that takes 1 string parameter
```

```
string s3 = "I'm a string"; // different! (we'll get to that)
```

The Implicit Parameter

implicit parameter:

The object on which a member function is called.

- During the call `chris.withdraw(...)`,
the object named `chris` is the implicit parameter.
- During the call `aaron.withdraw(...)`,
the object named `aaron` is the implicit parameter.
- The member function can refer to that object's member variables.
 - We say that it executes in the context of a particular object.
 - The function can refer to the data of the object it was called on.
 - It behaves as if each object has its own copy of the member functions.

Let's Start an Example: The Fraction Class



- As an example of a class, we're going to define a **Fraction** class that can deal with rational numbers directly, without decimals.
- We are going to walk through the class one step at a time, demonstrating the various parts of a class as we go.

The Fraction Class

- Questions we must answer about the Fraction class:

What data should the class hold?

What kinds of functions (public / private) should our class have?

What constructors could we have?

What is a good value for a default fraction?

$$\frac{3}{8} + \frac{6}{4}$$

The Fraction Class

Class outline

```
class Fraction {
```

```
public:
```

Things we want class users to see

```
private:
```

Things we want to keep hidden
from class users

```
};
```

The Fraction Class

public:

Class outline

```
private:
    int num; // the
    numerator
    int denom; // the
    denominator
```

What data would a Fraction class have?

Why is it private?

The Fraction Class

```
class Fraction {  
public:  
    void add(Fraction f);  
    void mult(Fraction f);  
    double decimal();  
    int getNum();  
    int getDenom();  
    friend ostream& operator<<  
        (ostream& out, Fraction &frac);  
  
private:  
    int num; // the numerator  
    int denom; // the denominator  
  
};
```

What functions
should a fraction
class be able to do?

Why are they public?

What is this???

The Fraction Class

```
class Fraction {  
public:  
    void add(Fraction f);  
    void mult(Fraction f);  
    double decimal();  
    int getNum();  
    int getDenom();  
    friend ostream& operator<<  
        (ostream& out, Fraction &frac);  
  
private:  
    int num; // the numerator  
    int denom; // the denominator  
};
```

What is this???

This defines an operator "overload" to make it possible to use the "<<" operator with cout.

We will write this function in a few minutes.

The Fraction Class

```
class Fraction {  
public:  
    Fraction();  
    Fraction(int num, int denom);  
    void add(Fraction f);  
    void mult(Fraction f);  
    double decimal();  
    int getNum();  
    int getDenom();  
    friend ostream& operator<<  
        (ostream& out, Fraction &frac);  
  
private:  
    int num; // the numerator  
    int denom; // the denominator  
};
```

We need to *construct* the class when it is called.

What should a "default" fraction look like?

1 / 1 probably makes the most sense (why not 0/0?)

Should we let the user create an initial fraction, e.g., 3/4?

The Fraction Class

```
class Fraction {  
public:  
    Fraction();  
    Fraction(int num,int denom);  
    void add(Fraction f);  
    void mult(Fraction f);  
    double decimal();  
    int getNum();  
    int getDenom();  
    friend ostream& operator<<  
        (ostream& out, Fraction &frac);  
  
private:  
    int num; // the numerator  
    int denom; // the denominator  
    void reduce(); // reduce the fraction  
  
    int gcd(int u, int v);  
};
```

Any other functions?

What about reduce?
(necessary for multiplication)

Reduce needs gcd()...

The Fraction Class

```
#pragma once
#include<ostream>
using namespace std;

class Fraction {
public:
    Fraction();
    Fraction(int num,int denom);
    void add(Fraction f);
    void mult(Fraction f);
    double decimal();
    int getNum();
    int getDenom();
    friend ostream& operator<<
        (ostream& out, Fraction &frac);
private:
    int num; // the numerator
    int denom; // the denominator
    void reduce(); // reduce the fraction
    int gcd(int u, int v);
};
```

Last, but not least...

The Fraction Class

Let's start writing our functions. We do this in our fraction.cpp file, and we have to define the class that each function belongs to. We also cannot forget to include our header file!

```
#include "fraction.h"
```

The **default constructor** is used when someone wants to just create a default fraction:

```
Fraction frac;
```

```
// purpose: the default constructor
// to create a fraction of 1 / 1
// arguments: none
// return value: none
// (constructors don't return anything)

Fraction::Fraction()
{

}
}
```

The Fraction Class

Let's start writing our functions. We do this in our fraction.cpp file, and we have to define the class that each function belongs to. We also cannot forget to include our header file!

```
#include "fraction.h"
```

The **default constructor** is used when someone wants to just create a default fraction:

```
Fraction frac;
```

```
// purpose: the default constructor  
// to create a fraction of 1 / 1  
// arguments: none  
// return value: none  
// (constructors don't return anything)
```

```
Fraction::Fraction()  
{
```

This tells the compiler what class we are creating. The double-colon is called the "scope resolution operator" because it helps the compiler resolve the scope of the function.

```
}
```

The Fraction Class

Let's start writing our functions. We do this in our fraction.cpp file, and we have to define the class that each function belongs to. We also cannot forget to include our header file!

```
#include "fraction.h"
```

The **default constructor** is used when someone wants to just create a default fraction:

```
Fraction frac;
```

```
// purpose: the default constructor
// to create a fraction of 1 / 1
// arguments: none
// return value: none
// (constructors don't return anything)

Fraction::Fraction()
{

    num  = 1;
    denom = 1;

}
```

Pretty simple! We are just setting our two class variables to default values.

The Fraction Class

We also have an *overloaded* constructor that takes in two values that the user sets. It is called as follows:

```
// create a
// 1/2 fraction
Fraction fracA(1,2);
```

```
// create a
// 4/6 fraction
Fraction fracB(4,6);
```

```
// purpose: an overloaded constructor
//         to create a custom fraction
//         that immediately gets reduced
// arguments: an int numerator
//            and an int denominator
Fraction::Fraction(int num,int denom)
{

```

The Fraction Class

We also have an *overloaded* constructor that takes in two values that the user sets. It is called as follows:

```
// create a  
// 1/2 fraction  
Fraction fracA(1,2);
```

```
// create a  
// 4/6 fraction  
Fraction fracB(4,6);
```

```
// purpose: an overloaded constructor  
//         to create a custom fraction  
//         that immediately gets reduced  
// arguments: an int numerator  
//            and an int denominator  
Fraction::Fraction(int num, int denom)  
{  
  
    this->num = num;  
    this->denom = denom;  
  
    // reduce in case we were given  
    // an unreduced fraction  
    reduce();  
}
```

The Fraction Class

Let's write some more functions...

```
// create two fractions
Fraction fracA(1,2);
Fraction fracB(2,3);

fracA.mult(fracB);
// fracA now holds 1/3
```

```
// purpose: to multiply another fraction // with this one
// with the result being
// stored in this fraction
// arguments: another Fraction
// return value: none
void Fraction::mult(Fraction other)
{

}
```

The Fraction Class

Let's write some more functions...

```
// create two fractions
Fraction fracA(1,2);
Fraction fracB(2,3);

fracA.mult(fracB);
// fracA now holds 1/3
```

```
// purpose: to multiply another fraction // with this one
// with the result being
// stored in this fraction
// arguments: another Fraction
// return value: none
void Fraction::mult(Fraction other)
{
    // multiplies a Fraction
    // with this Fraction
    num *= other.num;
    denom *= other.denom;

    // reduce the fraction
    reduce();
}
```

The Fraction Class

Let's write some more functions...

```
// get the decimal value
Fraction fracA(1,2);
double f = fracA.decimal();
cout << f << endl;
```

output:
0.5

```
// purpose: To return a decimal
// value of our fraction
// arguments: None
// return value: the decimal
//             value of this fraction
double Fraction::decimal()
{

}
```

The Fraction Class

Let's write some more functions...

```
// get the decimal value
Fraction fracA(1,2);
double f = fracA.decimal();
cout << f << endl;
```

output:
0.5

```
// purpose: To return a decimal
// value of our fraction
// arguments: None
// return value: the decimal
//             value of this fraction
double Fraction::decimal()
{
    // returns the decimal
    // value of our fraction
    return (double)num / denom;
}
```

The Fraction Class: reduce()

```
void Fraction::reduce() {  
    // reduce the fraction to lowest terms  
    // find the greatest common divisor  
    int frac_gcd = gcd(num,denom);  
  
    // reduce by dividing num and denom  
    // by the gcd  
    num = num / frac_gcd;  
    denom = denom / frac_gcd;  
}
```

The Fraction Class: gcd() — nice recursive function

```
int Fraction::gcd(int u, int v) {  
    if (v != 0) {  
        return gcd(v, u%v);  
    }  
    else {  
        return u;  
    }  
}
```


Exercise 3

- Write a function named *NonEmpty* that has one parameter, a vector of strings *V*, and that returns another vector of strings that contains just the non-empty strings in *V*. For example, if parameter *V* contains the 6 strings: "hello", "", "bye", "", "", "!" then function *NonEmpty* should create and return a vector that contains the 3 strings: "hello", "bye", "!"

Solution

- You should have come up with something like this:

```
01. vector <string> NonEmpty(vector <string> V) {  
02.     // count the number of non-empty strings in V  
03.     int num = 0;  
04.     for (int k=0; k<V.size(); k++) {  
05.         if (V[k] != "") num++;  
06.     }  
07.  
08.     // declare a vector of the right size and fill it in  
09.     vector <string> newV(num);  
10.     int index = 0;  
11.     for (int k=0; k<V.size(); k++) {  
12.         if (V[k] != "") {  
13.             newV[index] = V[k];  
14.             index++;  
15.         }  
16.     }  
17.  
18.     // return result  
19.     return newV;  
20. }
```

Exercise 4

Write a function named `Expand` that has one parameter, a vector of ints `V`. `Expand` should change `V` so that it is double its original size, and contains (in its first half) the values that were originally in `V`.

Solution

- You should have come up with something like this:

When you run this program, the output should be:

```
vector size before calling Expand: 1
vector size after calling Expand: 2
vector size before calling Expand: 2
vector size after calling Expand: 4
vector size before calling Expand: 4
vector size after calling Expand: 8
vector size before calling Expand: 8
vector size after calling Expand: 16
[ 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 ]
```

```
01. void Expand(vector<int> &V) {
02.     V.resize(2*V.size());
03. }
```

```
01. #include <iostream>
02. #include <vector>
03.
04. int main() {
05.     vector v(1);
06.
07.     for (int k = 1; k <= 16; k++) {
08.         if (v.size() < k) {
09.             cout << "vector size before calling Expand: " << v.size() << endl;
10.             Expand(v);
11.             cout << "vector size after calling Expand: " << v.size() << endl;
12.         }
13.         v[k-1] = k;
14.     }
15.     cout << "[ ";
16.     for (int k = 0; k < v.size(); k++) {
17.         cout << v[k] << ' ';
18.     }
19.     cout << "]\n";
20.     return 0;
21. }
```