Objects and classes

Last changed 3/25/2022

What we will learn in this set of slides

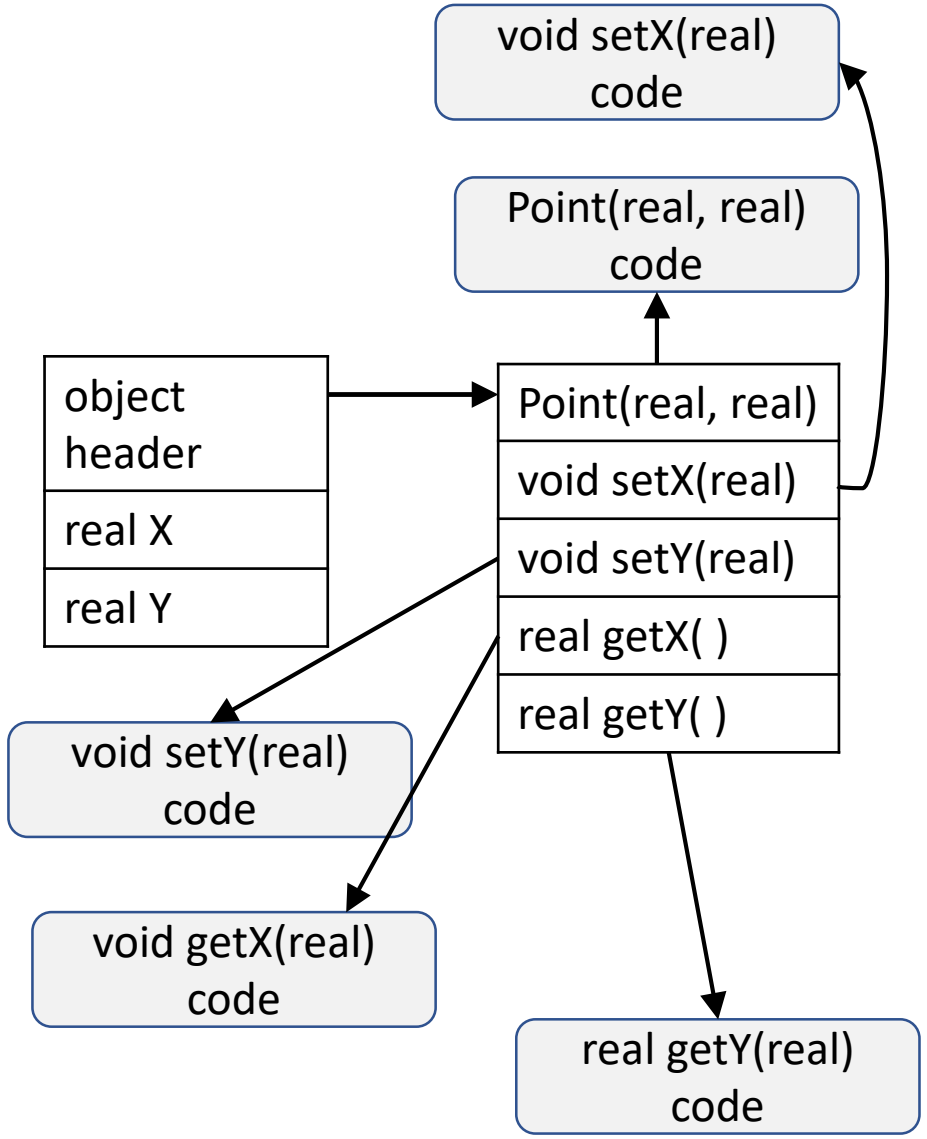
- What an object is, and how they are represented
- What a class is
- A little bit about protection levels
- A little bit about methods and constructors and object allocation
- What we mean by an *interface* and start learning about the right way to make our interfaces
- A tiny bit of Universal Modeling Language (UML)

What is an object?

- An object can be defined in several ways
- Physically, it is
  - A struct that contains the fields, or *attributes*, of the object
  - Functions that operate on the attributes of the object
  - A data structure called the *virtual function table* that allows functions associated with the object to be called
  - Special functions called *constructors* that initialize a newly created objects
  - A *this* reference that points to the struct containing the fields of the object
- Programmatically, it is an *instance* of a *class*. The *class* defines what an object looks like, and its methods, constructors and *destructors*.
  - *Methods* are functions that operate on the object
  - *Constructors* are special functions that initialize the fields or attributes of objects
  - *Destructors* are objects that allow objects to clean up after themselves when they are destroyed
- Stylistically, it is a set of operations that return values, i.e., it is an interface for some set of functionality
  - Stylistically, we should never know what data is held by the object, only that there is an object and operations that return results

Physical representation of an object

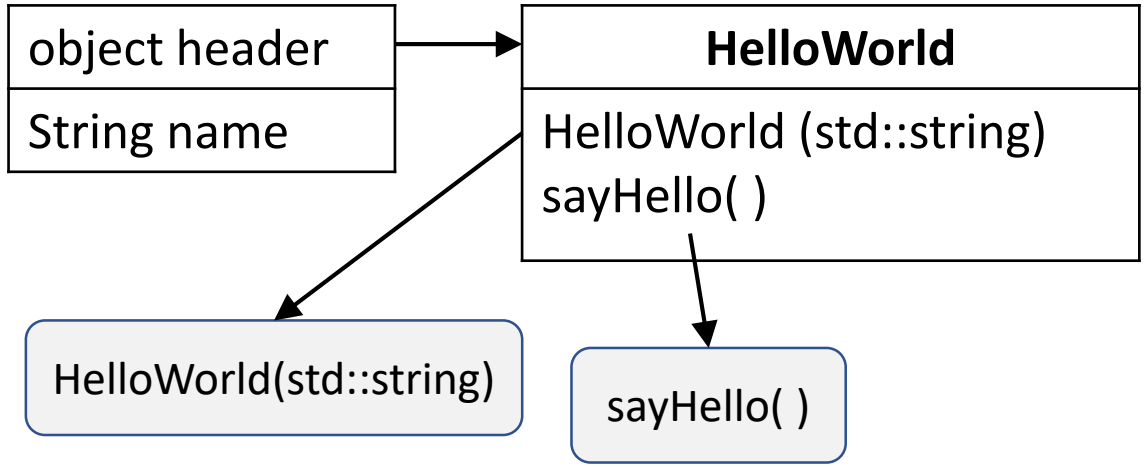
- Consider an object that represents a point in a 2-D cartesian space
  - It has an X and Y coordinate
  - It has functions *setX(real), setY(real), getX( )* and *getY( )* that set and get the X and Y coordinates, respectively
  - It has a *constructor, Point(real,real)* that initializes the data in the struct associated with some Point.
- The rest of this semester will discuss how to define, create, organize and use objects to write programs.



Let's do a HelloWorld in C++ using a class and an object (See Example1 code)

HelloWorld.h

```
#ifndef HelloWorld_H_
#define HelloWorld_H_
#include <string>

class HelloWorld {

private:
   std::string name;

public:
   HelloWorld(
      std::string _name
   );
   virtual void sayHello( );
};
#endif /* HELLOWORLD_H_ */
```

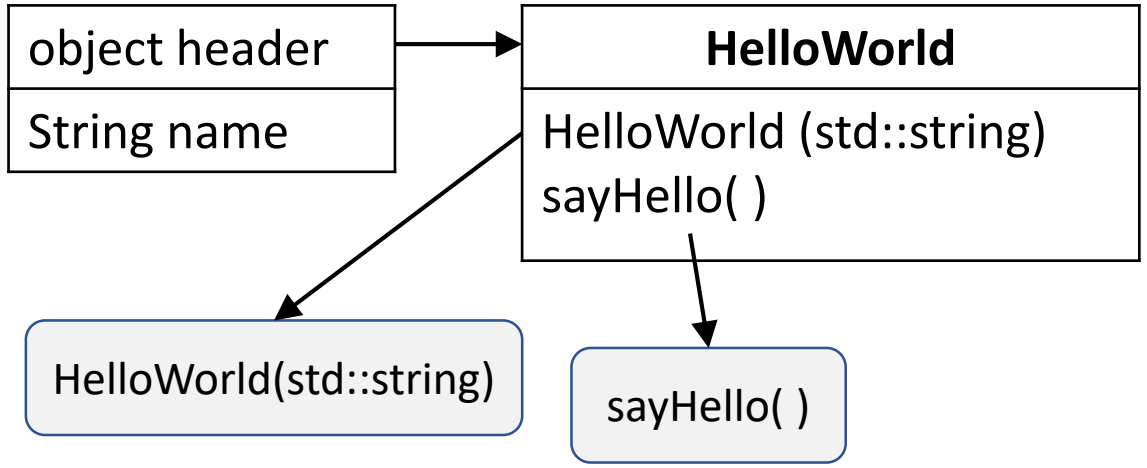


- We use a .h file named the same as the class to *declare* the class
  - The class declares attributes and functions associated with the class. For now, we'll always make functions *virtual*
  - It also declares a special function, called a *constructor*
    - The constructor has the same name as the
    - It is called to initialize the struct associated with the class
    - It has no return value – it simply fills in fields of an object

The .h file is similar in purpose to the .h in a C program – for declarations, in this case declaring the class and its *attributes*.

HelloWorld.h

```
#ifndef HelloWorld_H_
#define HelloWorld_H_
#include <string>

class HelloWorld {

private:
   std::string name;

public:
   HelloWorld(std::string _name);
   virtual void sayHello( );
};
#endif /* HELLOWORLD_H_ */
```

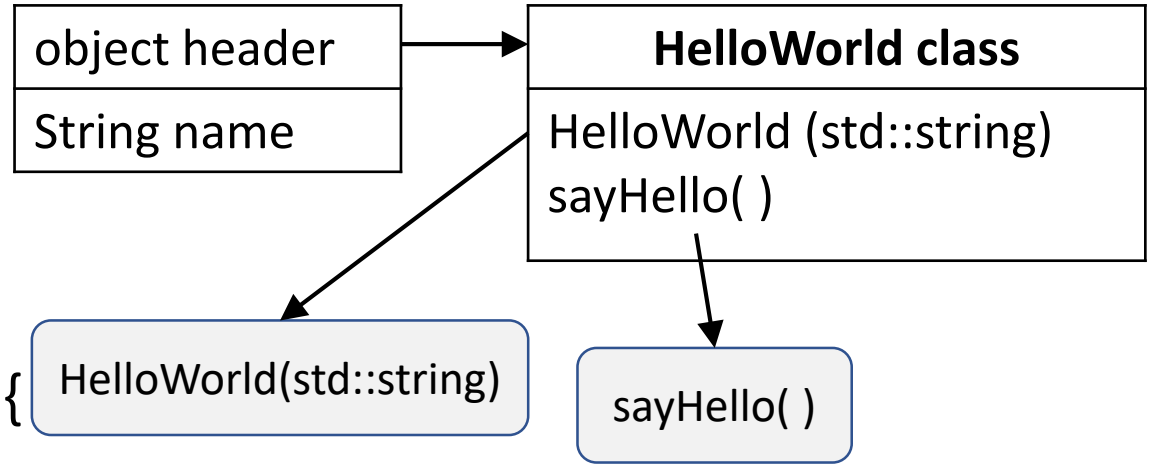


- The .h file declares functions that will operate on the fields (attributes) of the struct that hold the data's object
  - These functions are called *methods* in OO
  - Methods can be *virtual* or non-virtual.
    - For now, we'll declare all of our methods as virtual
    - We'll see what *virtual* means in a lecture or two
- C++ also allows functions that are not part of a class to be declared. *main* is one example of this.

The .cpp file contains *definitions* of what was declared in the .h

HelloWorld.cpp

```
#include <string>
#include <iostream>
#include "HelloWorld.h"

HelloWorld::HelloWorld(std::string _name) {
  name = _name;
}

void HelloWorld::sayHello( ) {
  std::cout << "Hello world and Hello “;
  std::cout << +name+"!" << std::endl;
}
```



- #includes are like with C
- *iostream* allows us to use *cout* to print to the standard console, i.e., to stdout
- *string* allows C++ strings to be used
- Since the declaration and definition are in separate files, the definitions need to specify what class the symbol is in

The main function in C++ is not part of any class

```
#include "HelloWorld.h"

int main(int argc, char* argv[ ]) {
   HelloWorld* helloWorld = new HelloWorld("Sam");
   helloWorld->sayHello( );
}
```

Output:

```
Hello world and Hello Sam!
```

main must be callable before any objects are created. Therefore *main* cannot be a method in a class.

The ***new*** operator allocates space on the heap for a new object, and calls the *constructor* on that space to initialize the object. ***new*** creates object, constructors initialize objects.

new returns a pointer to the object that was just allocated and initialized.

C++ objects and how to access them

```
#include "HelloWorld.h"

int main(int argc, char* argv[ ]) {
  HelloWorld* helloWorld = new HelloWorld("Sam");
  helloWorld->sayHello( );
}
```

C++ allows object to be created on the heap, or as stack allocated variables, like a regular C variable declared in a function

For now, we'll use heap allocated objects accessed using a pointer and the "->" operator
In the future, we'll see how to create and initialize object variables, and how they can be accessed either as a variable, through a pointer, or using a *reference.*

There are usually many objects created from a single class

```
HelloWorld::HelloWorld(std::string _name, HelloWorld* this) {
  name = _name;
}

void HelloWorld::sayHello(HelloWorld* this) {
  std::cout << "Hello world and Hello “;
  std::cout << +name+"!" << std::endl;
}

int main(int argc, char* argv[ ]) {
  HelloWorld* hello1 = new HelloWorld("Sam");
  HelloWorld* hello2 = new HelloWorld("Mary");
  hello1->sayHello( );
  hello2->sayHello( );
  hello1 = hello2;
  hello1->sayHello( );
}
```

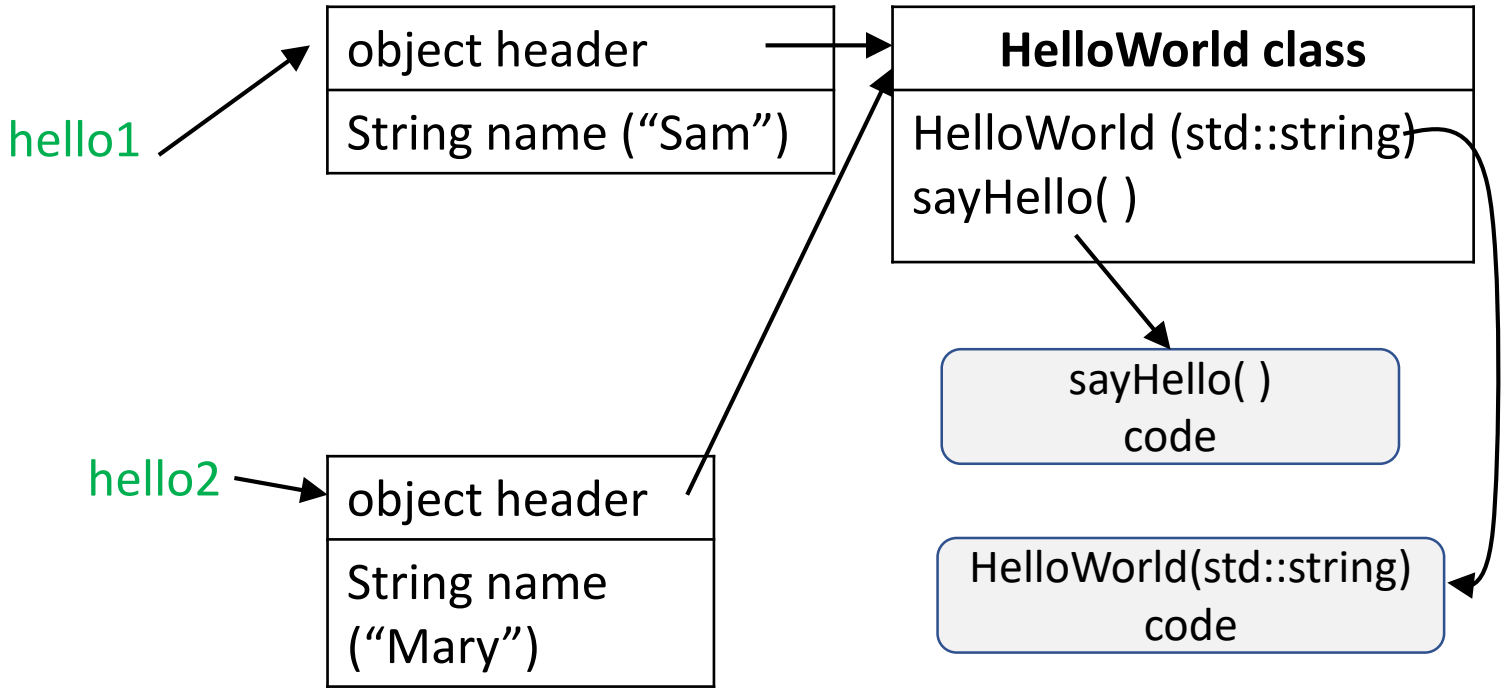
Output:

```
Hello world and Hello Sam!
Hello world and Hello Mary!
Hello world and Hello Mary!
```

Each object is an *instance* of the class

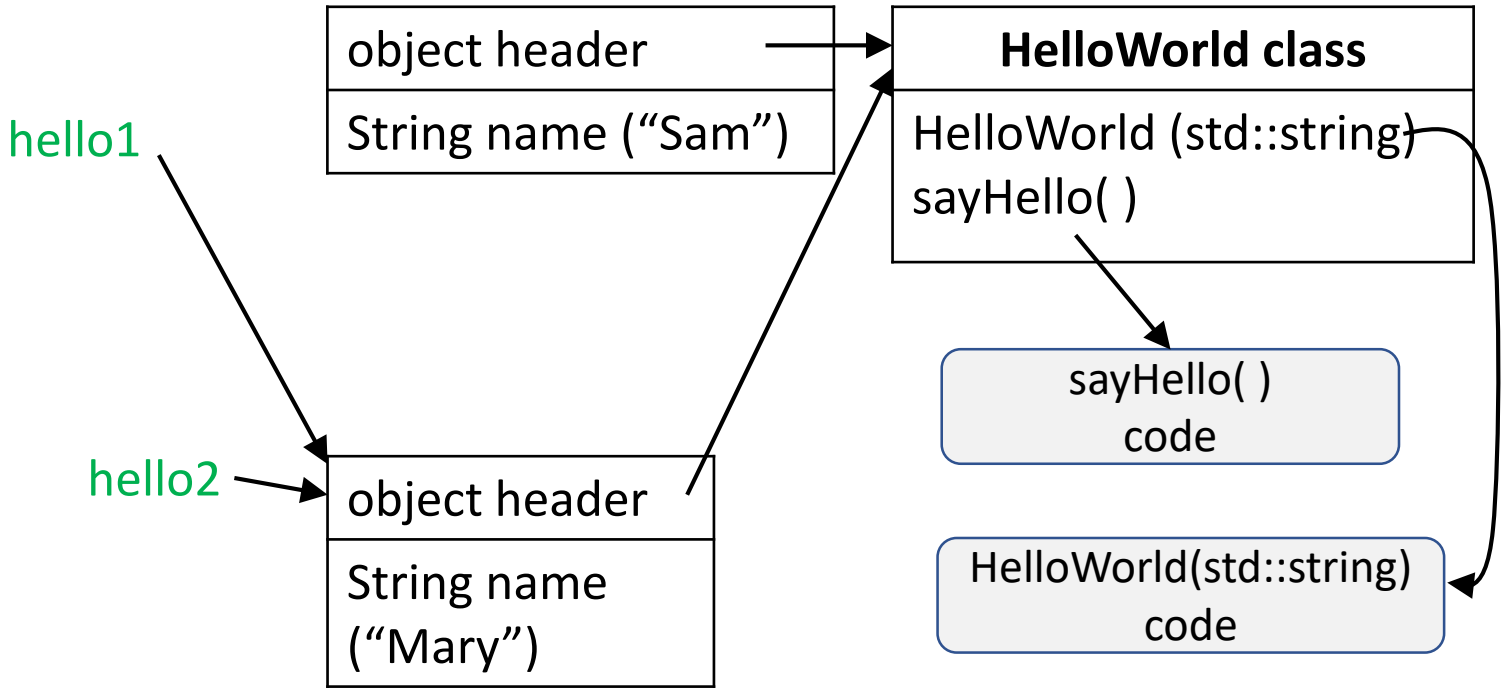
Pointers allow the object to be accessed (See Example2 code)

```
#include "HelloWorld.h"

int main(int argc, char* argv[ ]) {
   HelloWorld* hello1 = new HelloWorld("Sam");
   HelloWorld* hello2 = new HelloWorld("Mary");
   . . .
}
```



Pointers allow the object to be accessed (See Example2 code)

```
#include "HelloWorld.h"

int main(int argc, char* argv[ ]) {
   . . .
   hello1 = hello2;
   hello1->sayHello( );
}
```

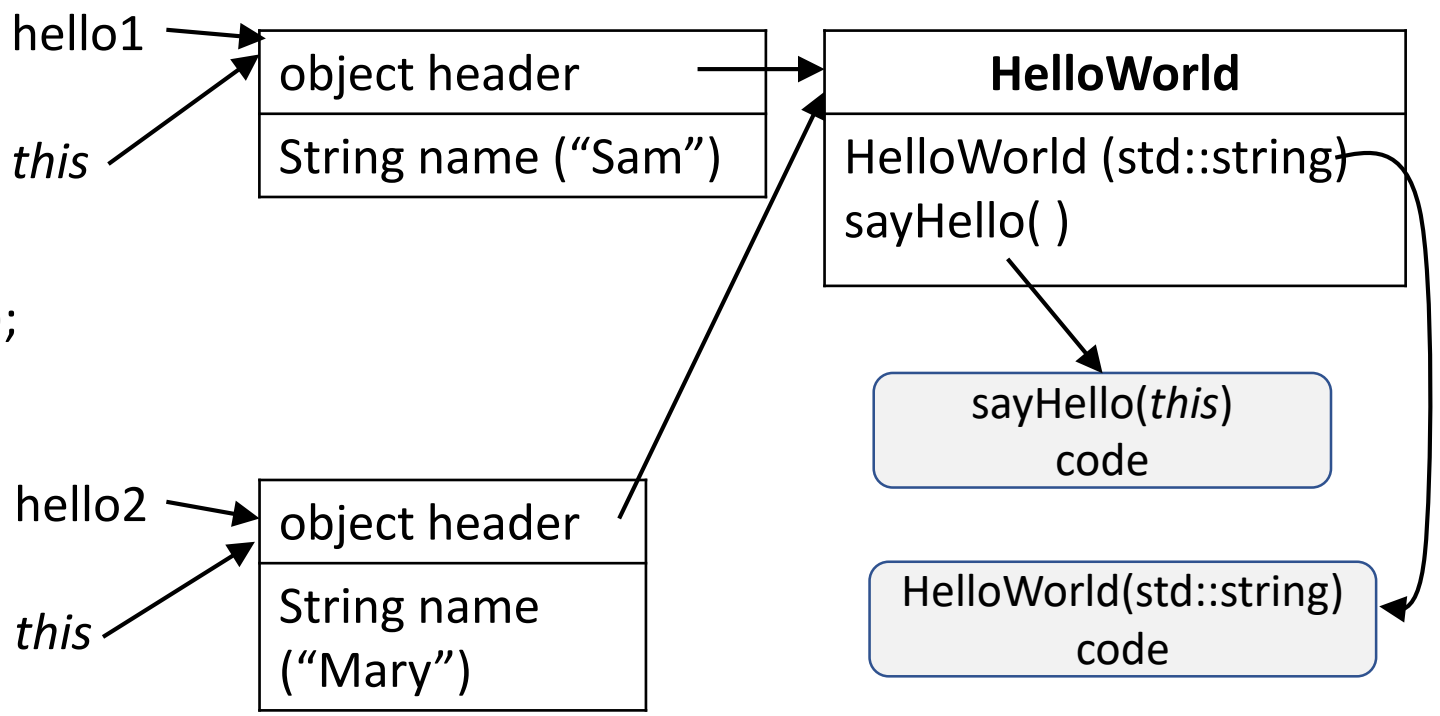


How does *sayHello( )* know which object to get the name from?

- Each method has only one copy of code – there is not a copy for each object
- We have multiple objects
- Where the struct representing the object is located is not known until the object is allocated by *new*
- What object *sayHello( )* is called on is not determined until runtime
- How does the method code know which struct containing the attributes of the object is to be accessed?
  - i.e., how is the invocation of sayHello( ) in the first statement hello1->sayHello( ); able to access the struct with name = “Sam” and not the struct with name = “Mary”?
  - The *this* pointer solves this problem!

this allows a single piece of code to address different objects

```
#include "HelloWorld.h"

int main(int argc, char* argv[ ]) {
   HelloWorld* hello1 = new HelloWorld("Sam");
   HelloWorld* hello2 = new HelloWorld("Mary");
   hello1->sayHello( ); // S1
   hello2->sayHello( ); // S2
   hello1 = hello2;
   hello1->sayHello( ); // S3
}
```

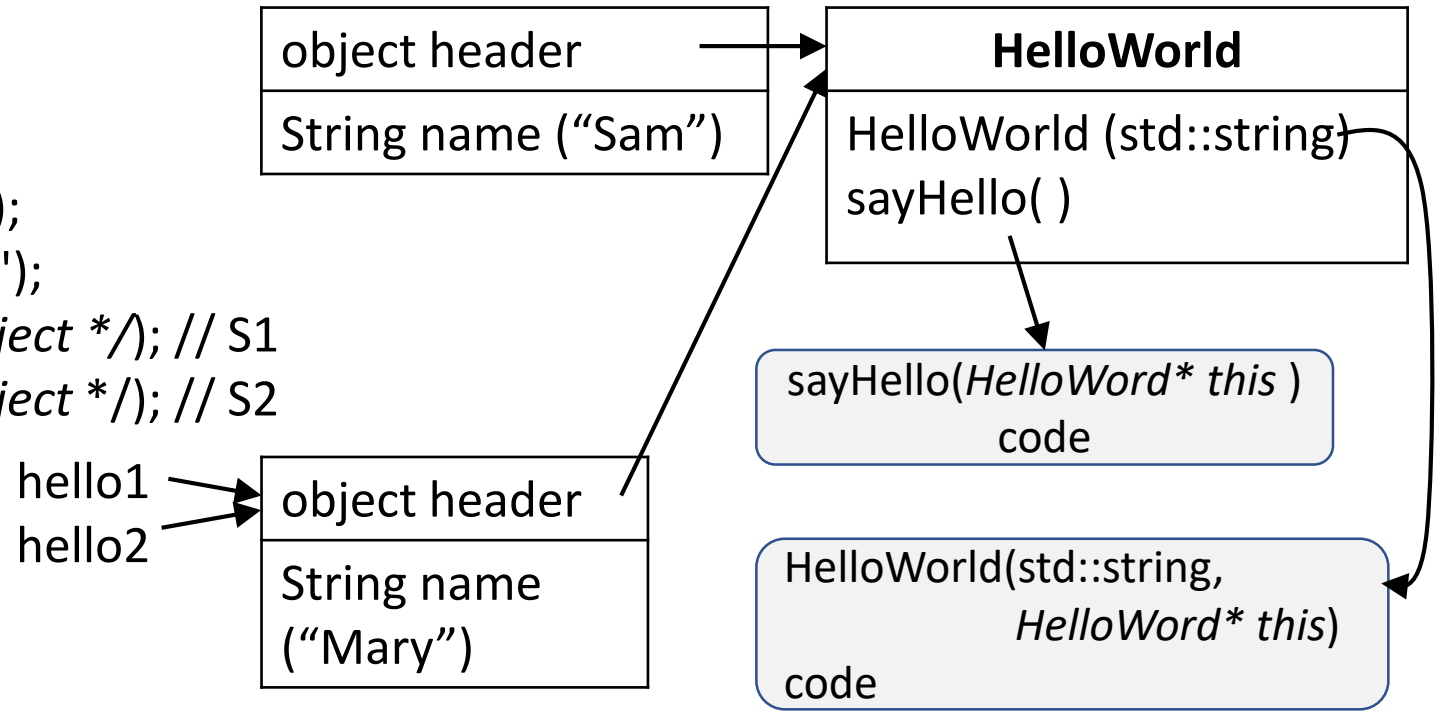


When a method is called on an object, a pointer to the object the method is called on is passed into the function – the passed pointer is the *this* pointer for the function

The *this* pointer is how the call in statement S1 knows to access the object with a name of “Sam”, and the call in statement S2 to access the object with a name of “Mary”.

this allows a single piece of code to address different objects

```
#include "HelloWorld.h"

int main(int argc, char* argv[ ]) {
   HelloWorld* hello1 = new HelloWorld("Sam");
   HelloWorld* hello2 = new HelloWorld("Mary");
   hello1->sayHello(this, /* address of hello1 object */); // S1
   hello2->sayHello(this, /* address of hello2 object */); // S2
   hello1 = hello2; hello1->sayHello( ); // S3
}
```



Each object’s *this* pointer allows the struct for the object (we’ll just say object, from here on out) to be accessed

The *this* pointer is how the calls in S1 and S3 know to access the object with a name of “Sam”, and the call in S2 to access the object with a name of “Mary”.

this is added automatically – we don’t have to add it.

How *this* is passed to methods

```
HelloWorld::HelloWorld(std::string _name, HelloWorld* this) {
  name = _name;
}

void HelloWorld::sayHello(HelloWorld* this) {
  std::cout << "Hello world and Hello “;
  std::cout << +name+"!" << std::endl;
}


int main(int argc, char* argv[ ]) {
  HelloWorld* hello1 = new HelloWorld("Sam");
  HelloWorld* hello2 = new HelloWorld("Mary");
  hello1->sayHello(hello1);
  hello2->sayHello(hello1);
  hello1 = hello2;
  hello1->sayHello(hello1);
}
```

For function definitions, the compiler inserts an extra parameter that receives a pointer to the object’s struct. We do not add this.

For calls, the compiler inserts an extra argument that points to the struct containing the object attributes. We do not add this and it does not appear in our code.

Note that the type of a *this* pointer in functions of class C is of type C.

We can also use the *this* pointer

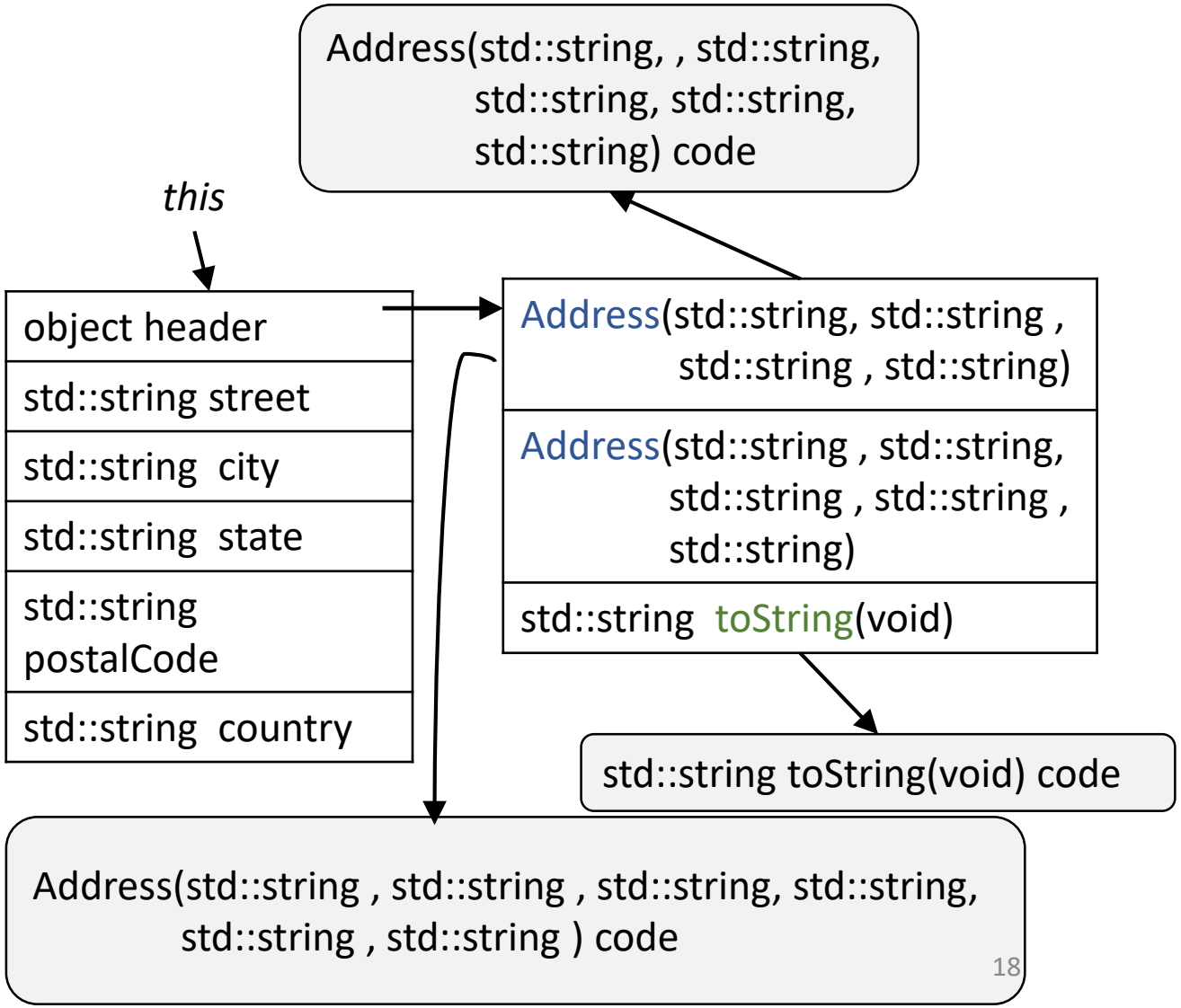
HelloWorld.cpp

```
#include <string>
#include <iostream>
#include "HelloWorld.h"

HelloWorld::HelloWorld(std::string name) {
  this->name = name;
}

void HelloWorld::sayHello( ) {
  std::cout << "Hello world and Hello “;
  std::cout << +name+"!" << std::endl;
}
```

```
#ifndef HelloWorld_H_
#define HelloWorld_H_
#include <string>

class HelloWorld {

private:
  std::string name;

public:
  HelloWorld(std::string _name);
  virtual void sayHello( );
};
#endif /* HELLOWORLD_H_ */
```

Let's consider an Address object (See Example3 code)

- Our address object will have 5 attributes:
  1. Street
  2. City
  3. State
  4. Postal Code
  5. Country
- It will have three functions:
  - Two constructors, one that defaults the country to "USA" and one that allows a country to be set
  - One method, called *toString( ),* that returns a String that represents the object



Let’s create the class that defines an address object

Address.h

```
class Address {
private:
   std::string street;
   std::string city;
   std::string state;
   std::string postalCode;
   std::string country = "USA";
```

```
public:
   Address(std::string _street, std::string _city,
            std::string _state, std::string _postalCode);

   Address(std::string _street, std::string _city,
            std::string _state, std::string _postalCode,
            std::string _country);

   std::string toString( );
};
```

What does the *private* keyword mean?

Address.h

```
class Address {
private:
   std::string street;
   std::string city;
   std::string state;
   std::string postalCode;
   std::string country = "USA";
```

Private says that only methods defined in the Address class have access to these variables. As code maintainers, we know that no code outside of Address.cpp can change the value of street, city, state, postalCode or country.

The constructor definitions from Address.cpp

```
Address::Address(std::string _street,
                 std::string _city,
                 std::string _state,
                 std::string _postalCode) {
  street = _street;
  city = _city;
  state = _state;
  postalCode = _postalCode;
  // country defaults to “USA”
}
```

```
Address::Address(std::string _street,
                 std::string _city,
                 std::string _state,
                 std::string _postalCode,
                 std::string _country) {
  street = _street;
  city = _city;
  state = _state;
  postalCode = _postalCode;
  country = _country;
}
```

A method that returns a string representation of an Address object for printing

```
std::string Address::toString( ) {
   std::string address = street+"\n";
   address += city+"\n";
   address += state+"\n";
   address += country+"\n";
   address += postalCode+"\n";
   return address;
}
```

And the main function, defined in main.cpp

```
int main(int argc, char* argv[ ]) {
  Address* address1 = new Address("465 Northwestern Ave, EE 310,
                                  Purdue University", "West Lafayette", "IN",
                                  "47907");
  std::cout << address1->toString( );

  Address* address2 = new Address("490 Northwestern Ave, MSEE 300, Purdue University",
                                   "West Lafayette", "IN", "47907",
                                   "United States of America");
  std::cout << address2->toString( );
}
```

The compiler passes *this* as a hidden parameter

```
int main(int argc, char* argv[ ]) {
  Address* address1 = new Address(. . .);
  std::cout << address1->toString( );

  Address* address2 = new Address(. . .);
  std::cout << address2->toString( );
}
```

```
int main(int argc, char* argv[ ]) {
  Address* address1 = new Address(. . .);
  std::cout << address1->toString(address1);

  Address* address2 = new Address(. . .);
  std::cout << address2->toString(address2);
}
```

When the C++ code is compiled, an extra argument – the *this* reference -- is inserted for every method that can access object attributes.

The *this* reference can be accessed with functions it is passed to if we want to pass a reference to the object to another function.

How are the private attributes updated? (See Example4 code)

```
class Address {
private:
   std::string street;
   std::string city;
   std::string state;
   std::string postalCode;
   std::string country = "USA";
```

```
std::string Address::getStreet( ) {
   return street;
}

void Address::setStreet(std::string _street) {
   street = _street;
}
```

We use *getter* and *setter* functions as shown in the Example4 code

What if we want to access individual elements?

```
std::string Address::getStreet( ) {
   return street;
}

void Address::setStreet(std::string _street) {
   street = _street;
}
```

- There are 5 attributes, street, city, state, postalCode and country
- Each getter and setter takes 3 lines of code
- 3 x 5 x 2 = 30 lines of code just to read and write the attributes!
- Why not make the attributes public and read them directly?
  - ***This is a terrible idea,*** and leads us to our 3rd definition.

Stylistically, objects are a set of operations that return values

- Classes define an *interface* that are *instantiated* in objects
- The interface is everything in the class that is accessible in code in other classes
- Ideally, only methods are in the interface
- Code making use of the interface of an object should have no idea
  - How the method is implemented
  - What the data (struct) accessed by the method looks like
- This allows everything that is in the class *and is not part of the interface* to be changed without changing code that accesses the interface
- This enables code reuse and other goodness that we will see later

Interfaces are common in many domains

- Chip designers can create 7+ billion transistor chips that actually work because chips are built up from sub-components
  - Sub-components have well defined interfaces
  - If the internals of the sub-component can be changed without changing the interface (and this is the goal) bugs and improvements in the sub-component can be done without affecting the overall design
- Designers treat sub-components as black boxes, and only worry about properly connecting to the interfaces
  - How the sub-component works is unimportant as long as it meets the spec
- Sub-components with well defined interfaces allow chip design to be done in a design and conquer fashion
  - Sub-component internals can be designed by teams that don’t have to communicate with other teams designing the chip

Shipping containers are a form of abstraction

- Not OO languages (although that would be a great selling point for the course)
- “The results are striking. In a set of 22 industrialised countries containerisation explains a 320% rise in bilateral trade over the first five years after adoption and 790% over 20 years. By comparison, a bilateral free-trade agreement raises trade by 45% over 20 years and GATT membership adds 285%.”
- More important than costs are knock-on effects on efficiency. In 1965 dock labour could move only 1.7 tonnes per hour onto a cargo ship; five years later a container crew could load 30 tonnes per hour (see table). This allowed freight lines to use bigger ships and still slash the time spent in port. The journey time from door to door fell by half and became more consistent. The container also upended a rigid labour force. Falling labour demand reduced dockworkers’ bargaining power and cut the number of strikes. And because containers could be packed and sealed at the factory, losses to theft (and insurance rates) plummeted. Over time all this reshaped global trade. Ports became bigger and their number smaller. More types of goods could be traded economically. Speed and reliability of shipping enabled just-in-time production, which in turn allowed firms to grow leaner and more responsive to markets as even distant suppliers could now provide wares quickly and on schedule. International supply chains also grew more intricate and inclusive. This helped accelerate industrialisation in emerging economies such as China, according to Richard Baldwin, an economist at the Graduate Institute of Geneva. Trade links enabled developing economies simply to join existing supply chains rather than build an entire industry from the ground up. But for those connections, the Chinese miracle might have been much less miraculous.
- http://www.economist.com/news/finance-and-economics/21578041-containers-have-been-more-important-globalisation-freer-trade-humble

The UML (Universal Modeling Language) representation of a class

Name of the Class (in bold)
Attributes of the class
Methods and constructors of the class

The UML for Address

Address
private std::string street; private std::string city; private std::string state; private std::string postalCode; private std::string country = "USA";
public Address(std::string _street, std::string _city, std::string _state, std::string _postalCode) public Address(std::string _street, std::string _city, std::string _state, std::string _postalCode, std::string _country) public std::string toString( )

UML gives us a concise way of representing key information about classes, and the relationships between classes

We'll use it to give an overview of classes without necessarily showing all of the code.

The representation of the class can be simplified

Address
street; city; state; postalCode; country
Address(std::string , std::string , std::string , std::string) Address(std::string , std::string, std::string ,std::string , std::string) toString( ) main(std::string [ ] args)

or

Address
- std::string street - std::string city - std::string state - std::string postalCode - std::string country = "USA"
+ Address + Address + String toString( )

What we learned in this set of slides

- What an object is
  - It's a struct, code to manipulate data in the struct, and a *this* pointer to access the struct
  - It's an instance of a class.
  - It's a set of operations that return values
- A class is a description of the data and code associated with an object
  - Objects are instances of classes
  - Classes are themselves can be represented as one or more objects at runtime
- A little about protection levels
  - public methods, constructors and data are accessible by code outside of the class
  - Private methods and constructors and data are NOT accessible by code outside of the class
- A little about methods and constructors and object allocation
  - The *new* operator allocates space for an object
  - The constructor initializes data in the new object
- What we mean by an *interface* and start learning about the right way to make our interfaces
  - The interface is all code and data accessible outside of the class
  - Interfaces should be small to allow as much code and data in the class to be changed as possible, without changing code that makes use of the class