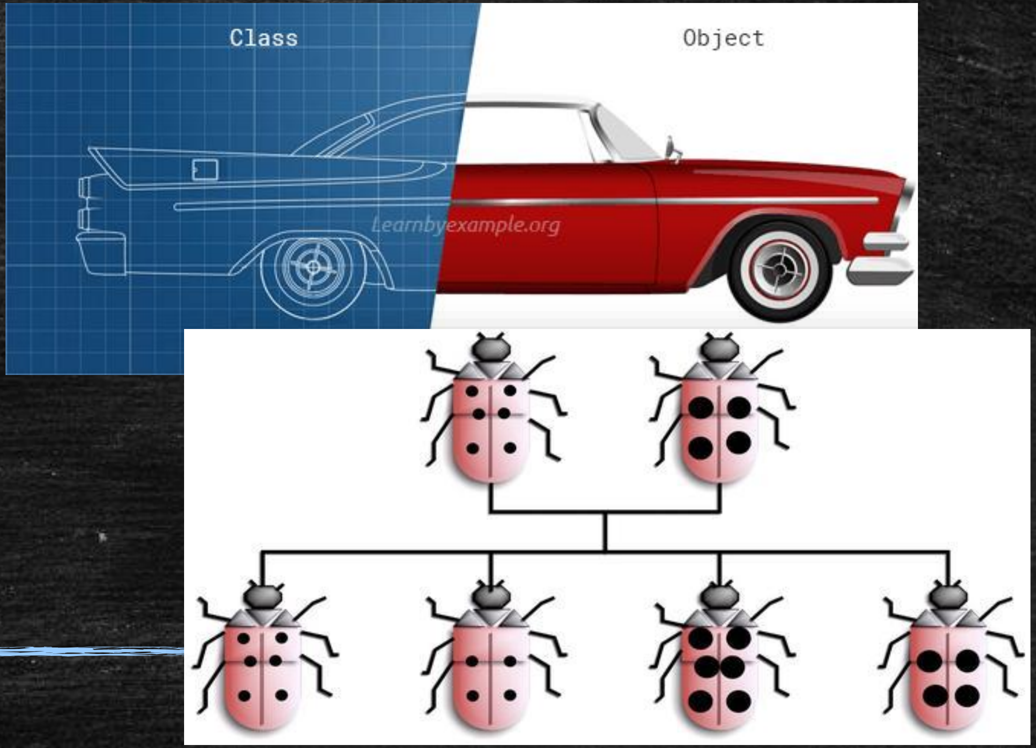
More on Classes & Inheritance



CS 150 – C++ Programming I

Lecture 26

Validating Constructor Parameters

- Constructor must preserve type invariants
  - Enumeration `Month` prevents bad month at compile time
  - What about a bad day? What should we do?
- Days change during different years and months
  - Some years have leap years; months have different valid days
- `mktime(&t)` changes `t` members if date arguments invalid
  - Check arguments against (possibly) changed values
  - `throw` the nested exception type
- Exercise: enhance the working constructor with `throw`

Object Members & Composition

- Data members are often object (class) types
  - Student has two Time members arrive and leave

  -
    ```
    Student(int hourArrive, int hourLeave)
    {    …
         arrive = Time(hourArrive);
         leave = Time(hourLeave);
    }
    ```
- Default constructor called, then the working constructor
  - 4 Time objects are constructed and then destroyed
  - This is very inefficient

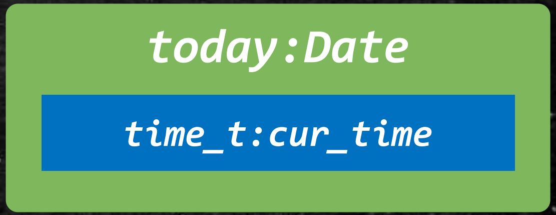
The Initializer List

- Initialize data members before running the constructor

  -
    ```
    Student(int hourArrive, int hourLeave)
      : arrive(hourArrive), leave(hourLeave)
    {
       // nothing in the body
    }
    ```
- The `Time` constructor is called only once for each member
  - Should always use whenever you have data members which are objects, and which should not be default-initialized
  - May use for almost all data members

Accessors and const

- Accessor members tell you about an object ("getters")
  - Accessors never change an object's data members
  - Thus, their signatures always have `const` after them
  - `int Date::day() const { . . . }`
  - `cout << today.day() << endl;`
- Real signature: `int day(const Date* this)`
  - `this` is called the implicit parameter
  - Means you cannot change `cur_time` on `today`
- Exercise: complete day, year, month



Mutators

- Objects which cannot be changed are called immutable
  - Immutable objects have many benefits (Google it)
- Mutators allow you to change an object
  - Need to make sure you don't change invariant
  - `Date& Date::addDays(int days){. . .}`
  - Not `const` because it will change state
- Returns a reference to modified object: `return *this;`
- Exercise: complete `addDays` mutator

Synthesized Constructors

- If you have no constructors the compiler "writes" a default
  - This is called the synthesized default constructor
  - It can be useful if you initialize your data members in place
- What if you only have a working constructor?
  - C++ removes the synthesized default constructor
  - Define explicitly, or add `=default` in the header

```
class Date
{
public:
    Date() = default;
    Date(int d, Month m, int year);
```

Conversion Constructors

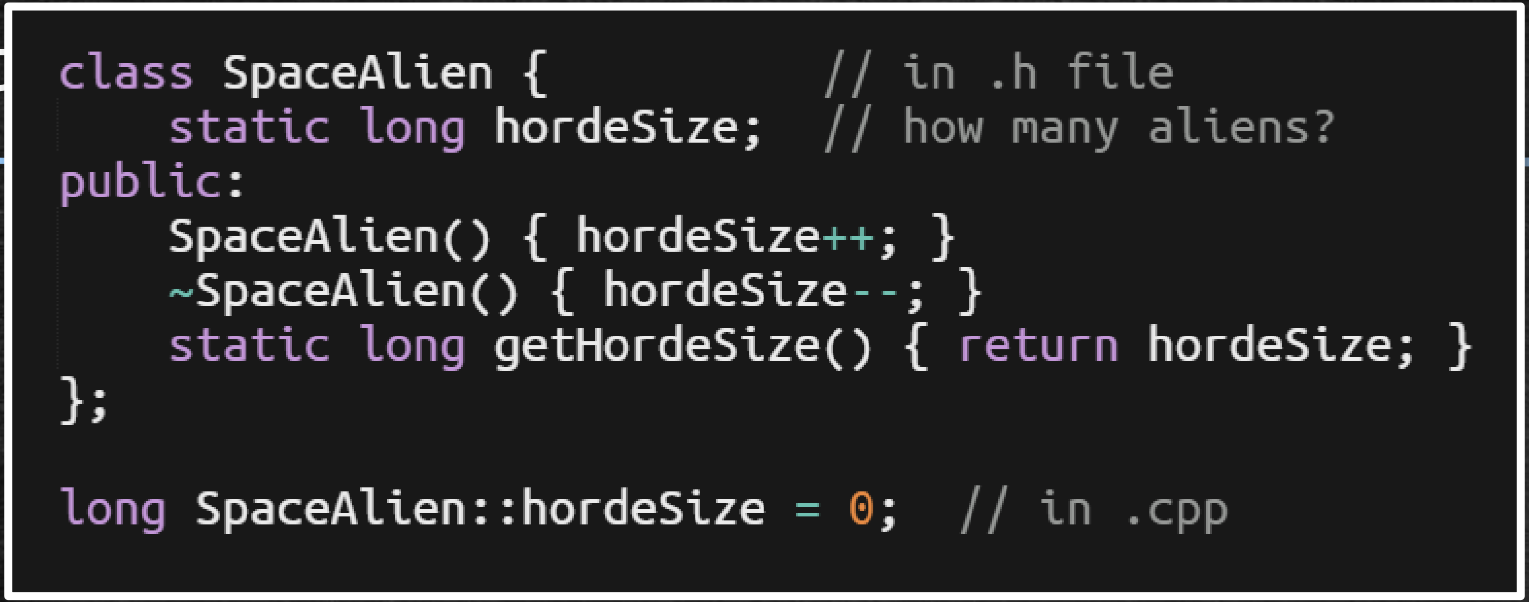
- A one-argument constructor will implicitly convert from the argument type to your object type

  -
    ```
    Employee(double);
    bob = 23;    // calls Employee(double)
    ```
  - Can't assign a `double` to an `Employee`, but given a `double`, C++ can implicitly (silently) create an `Employee`
- As you can imagine, this is somewhat dangerous!
  - Adding the `explicit` modifier to the prototype is safer
  - `explicit Employee(double);`

Constructors and Destructors

- When you initialize an object with another (of the same class), a copy constructor is called
  - `Employee(const Employee&)`
  - Also called when passing by value, but not for assignment
  - Like the default constructor, C++ synthesizes this for you
- A destructor is called whenever an object is destroyed
  - You may have one destructor per class, taking no arguments
  - Syntax: `~Classname();`
  - C++ synthesizes this for you as well. More important in CS 250
- Exercise: complete the `Troll` exercise

stat



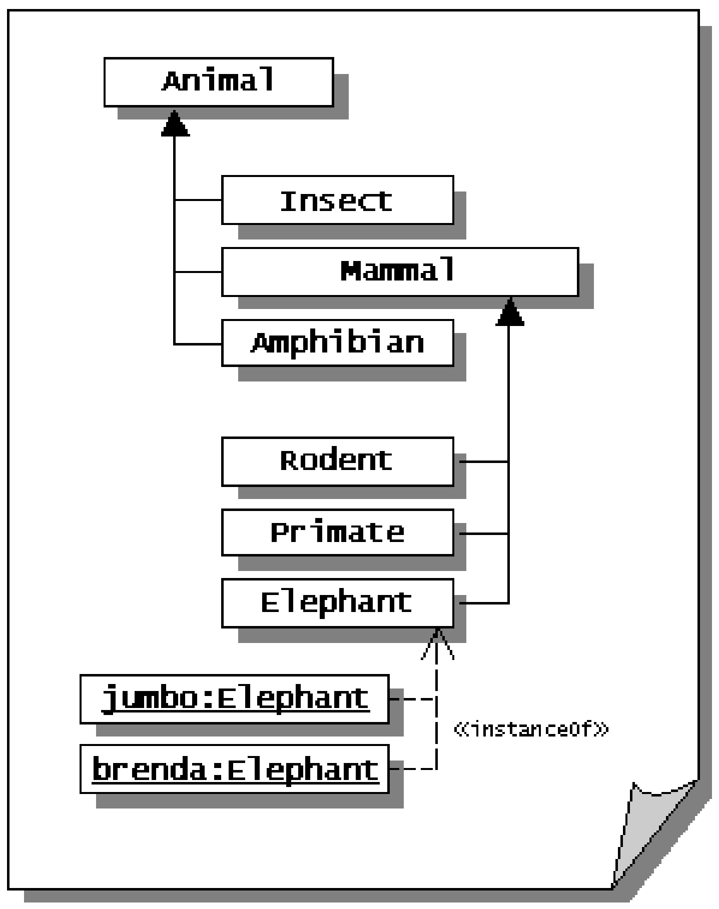
```
class SpaceAlien {            // in .h file
    static long hordeSize;   // how many aliens?
public:
    SpaceAlien() { hordeSize++; }
    ~SpaceAlien() { hordeSize--; }
    static long getHordeSize() { return hordeSize; }
};

long SpaceAlien::hordeSize = 0;  // in .cpp
```

- A `static` data member is a shared variable
  - Like a global, but only for that class
- Two parts: use `static` in the declaration (`.h` file)
  - Call `static` function: `SpaceAlien::getHordeSize();`
- Exercise: complete the `Fleas` problem

What is Inheritance?

- A mechanism for extending existing classes and creating families of classes called class hierarchies
- Families organized from general to specific
  - In C++, the general (parent) class is called a base class (`Animal`)
    - In Java and classic OO it is called a superclass
  - In C++, the specialized (child) class is called a derived class (`Insect`)
    - In Java it is called a subclass



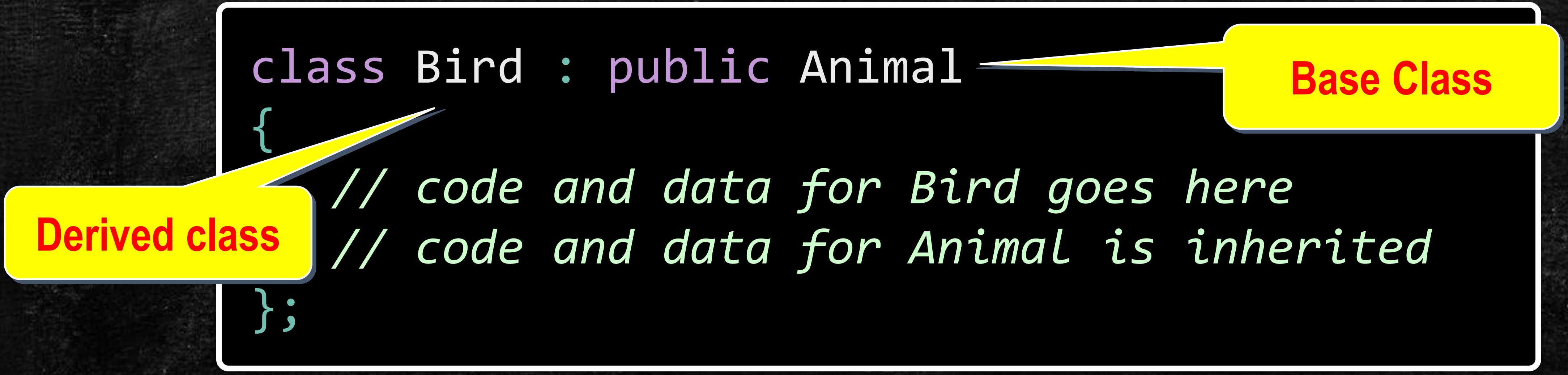
Class Relationships

- Classes have different kinds of relationships with each other
- Association is when one class uses another class to do some of its work. Informally this is called a Uses-A relationship
  - `ostream& operator<<(ostream& out, const Point& p);`
  - The `Point` class uses the `ostream` class to do its output
- Composition or aggregation is when one class actually contains an instance of another class that does its work
  - `class Student { Date birthday; ...}`
  - Informally this is called a Has-A relationship
  - The `Student` contains or has a `Date` as part of itself

The Is-A Relationship

- Is when one class is a specialized kind of another class
  - Ostrich is a Bird which is also an Animal
  - In a GUI, a Label is a Component, but so is a Button

- In C++, Is-A is implemented by using public inheritance:



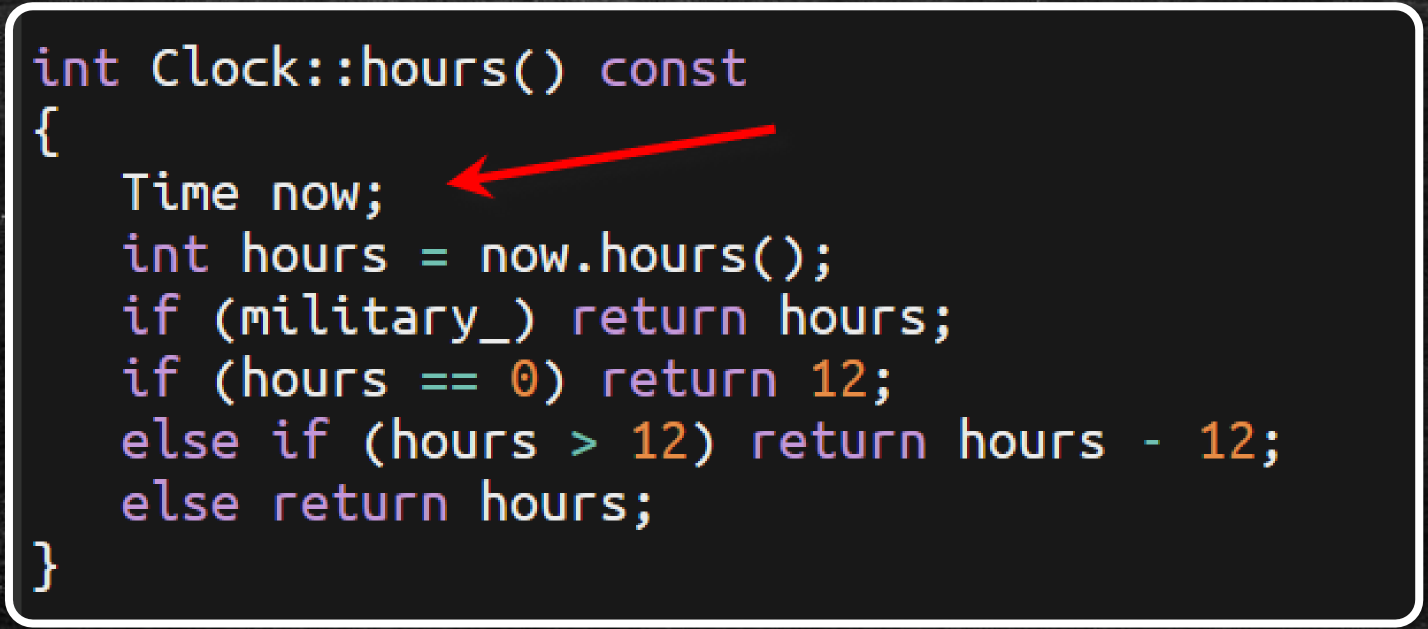
```
class Bird : public Animal
{
   // code and data for Bird goes here
   // code and data for Animal is inherited
};
```

Base Class

Derived class

An Inheritance Example (Clocks)

- The base class (Clock) can tell the current time



```
int Clock::hours() const
{
    Time now;
    int hours = now.hours();
    if (military_) return hours;
    if (hours == 0) return 12;
    else if (hours > 12) return hours - 12;
    else return hours;
}
```

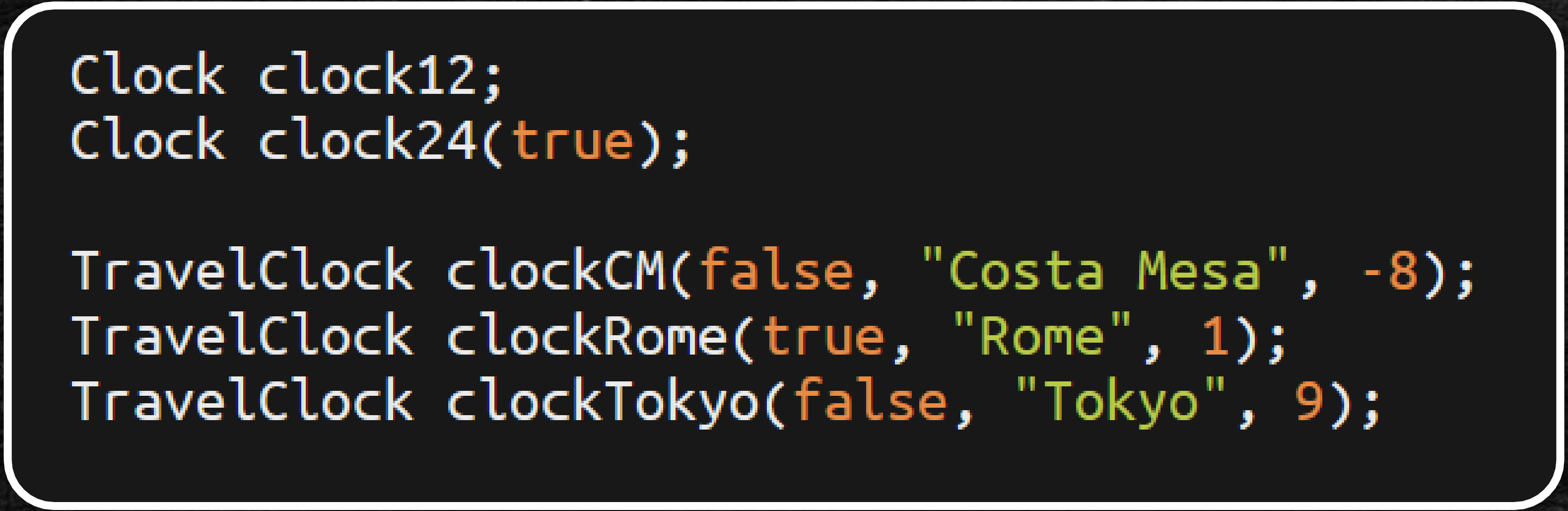
  - Has an association with Time class which uses <ctime>
  - Can report time in military (24 hour) or am/pm (12 hour)

Base Class Design Decisions

- `Clock` (base) class designer decides
  - How derived classes access the base class data members
  - Which member functions should be used as-is (inherited)
  - Which member functions may be redefined (overridden)
- Inherited member functions are those which the derived class is expected to use, but not change
- Virtual member functions are those which the derived class may, but need not, redefine or override

The TravelClock Class

- A derived class which reports time from different locations
  - location_ – where the clock is reporting (string)
  - timeDifference_ – hours from from GMT (UTC) time
- You can use both classes like this:

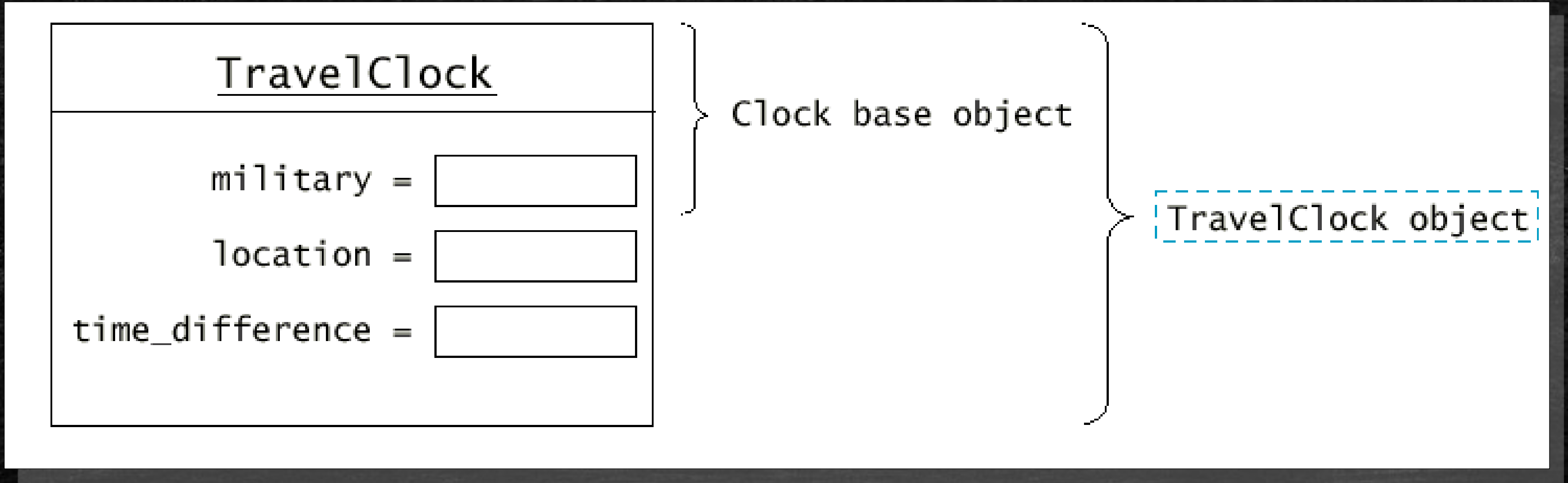


```
Clock clock12;
Clock clock24(true);

TravelClock clockCM(false, "Costa Mesa", -8);
TravelClock clockRome(true, "Rome", 1);
TravelClock clockTokyo(false, "Tokyo", 9);
```

TravelClock Memory Layout

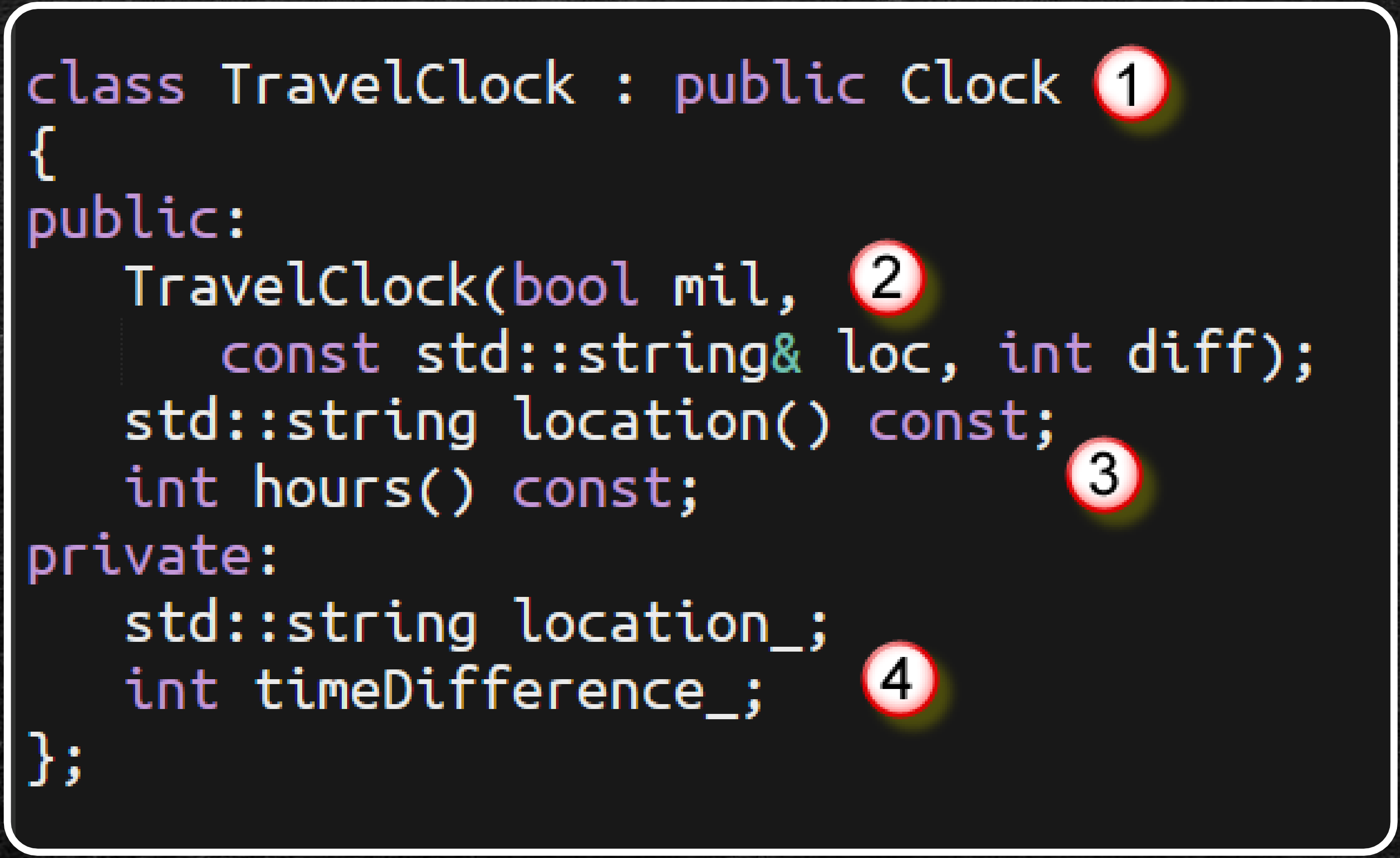
- Every *TravelClock* physically has a *Clock* part
  - This is a little different than as a data member
- The data members are combined in memory, so when you take the address of *TravelClock*, you'll find a *Clock*
  - That's why we can say every *TravelClock* ***IS-A*** *Clock*



TravelClock Differences

- The *TravelClock* is a specialized kind of *Clock*, and it differs from a plain *Clock* in three ways:
  - It contains data members for location & time difference
  - *TravelClock*::hours() adds in the time difference
  - location() returns the actual location, not "UTC (GMT)"
- The *TravelClock* class definition needs only to reference the *Clock* class and then spell out these three differences

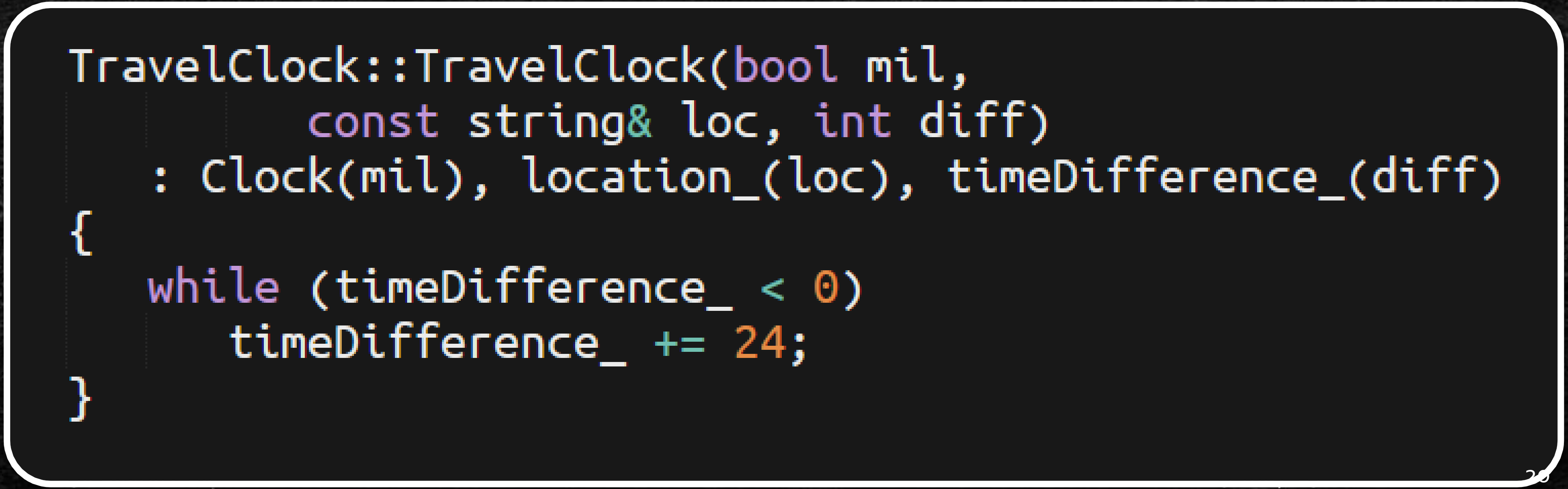
The *TravelClock* Definition



```
class TravelClock : public Clock
{
public:
   TravelClock(bool mil,
      const std::string& loc, int diff);
   std::string location() const;
   int hours() const;
private:
   std::string location_;
   int timeDifference_;
};
```

Derived Constructors

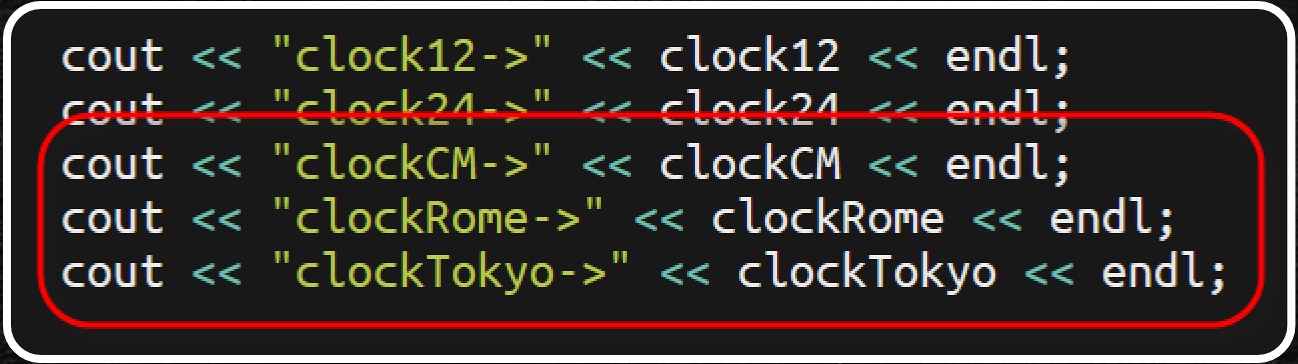
- The derived-class constructor has two tasks:
  - Initialize the base-class data members, before anything else
  - Then, initialize all of its own (derived) data members
- The derived class has no access to base-class private data
  - To initialize the base members, it must use the initializer list



```
TravelClock::TravelClock(bool mil,
        const string& loc, int diff)
   : Clock(mil), location_(loc), timeDifference_(diff)
{
   while (timeDifference_ < 0)
      timeDifference_ += 24;
}
```

Inherited Member Functions

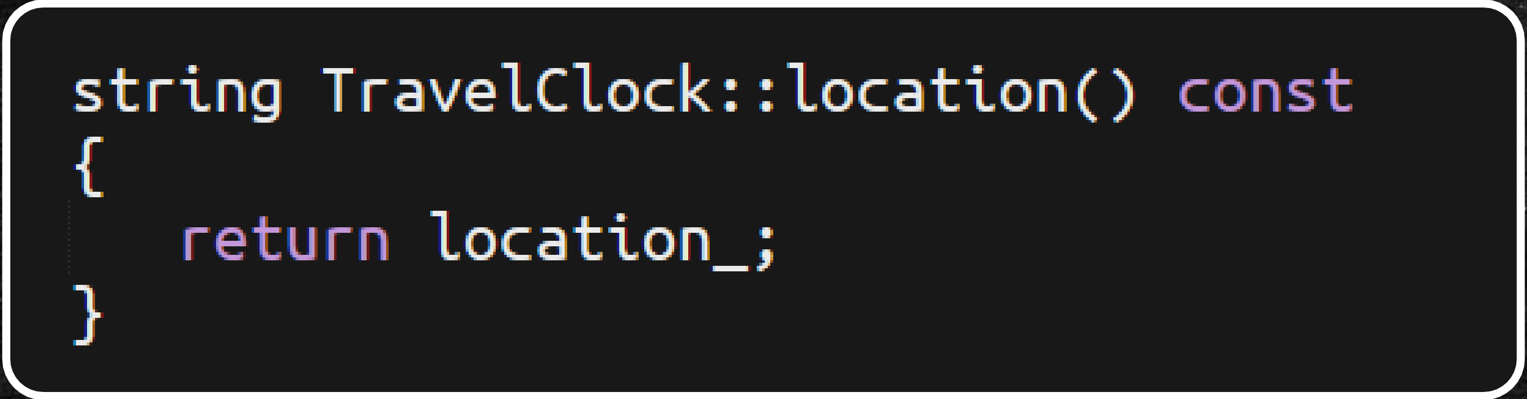
- The derived class inherits all of the base-class members
  - Use them without any changes at all
- The *TravelClock* class inherits these member functions:
  - *Clock::minutes()*
  - *Clock::isMilitary()*
  - The overloaded output operator defined in *Clock*



```
cout << "clock12->" << clock12 << endl;
cout << "clock24->" << clock24 << endl;
cout << "clockCM->" << clockCM << endl;
cout << "clockRome->" << clockRome << endl;
cout << "clockTokyo->" << clockTokyo << endl;
```

Overriding Member Functions

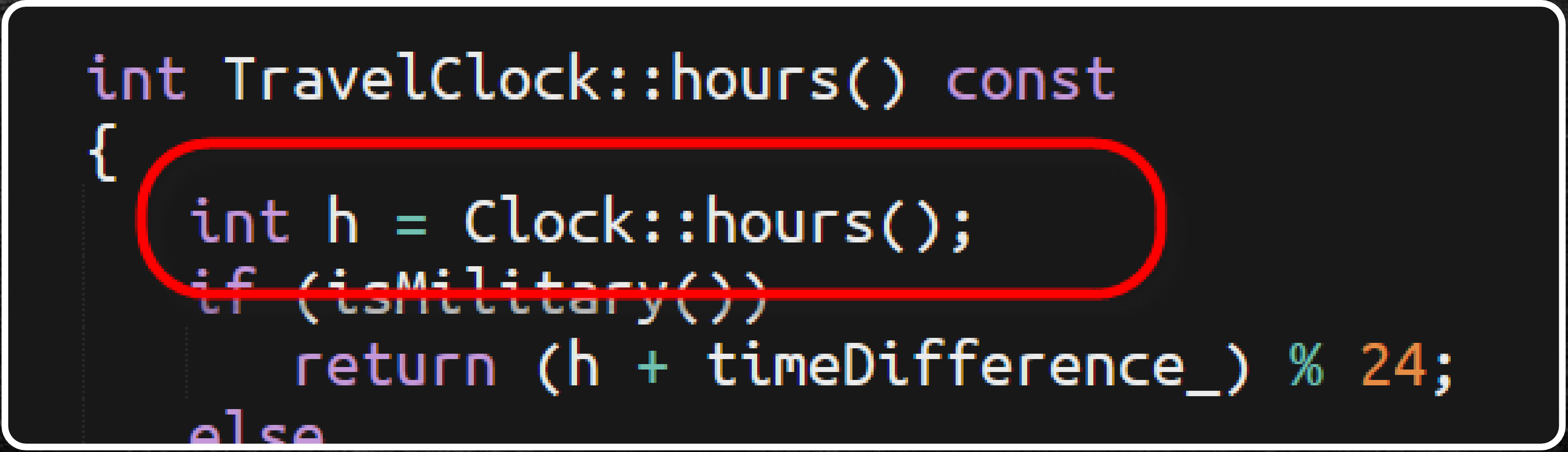
- Derived classes may override a *virtual* member function
  - Different than overloading; means to redefine in derived class
  - Overloading: functions must have different signatures
  - Overriding: functions must have exactly the same signature
- *TravelClock*::location() overrides *Clock*::location()



```
string TravelClock::location() const
{
    return location_;
}
```

Extending a Member Function

- A derived class can extend a virtual function by calling the base-class version from inside the derived-class version
- *TravelClock*::hours() extends *Clock*::hours()



```
int TravelClock::hours() const
{
    int h = Clock::hours();
    if (isMilitary())
        return (h + timeDifference_) % 24;
    else
```