Inheritance

Saurav Samantaray

Department of Mathematics

Indian Institute of Technology Madras

October 15, 2024

- "The Big Picture."
- let's take a look at our first UML feature: the class diagram.
- This diagram offers a new way of looking at object-oriented programs, and may throw some additional light on the workings of the TIMES1 and TIMES2 programs.
- Looking at the listing for TIMES1 we can see that there are two classes: time12 and time24.
- In a UML class diagram, classes are represented by rectangles, as shown

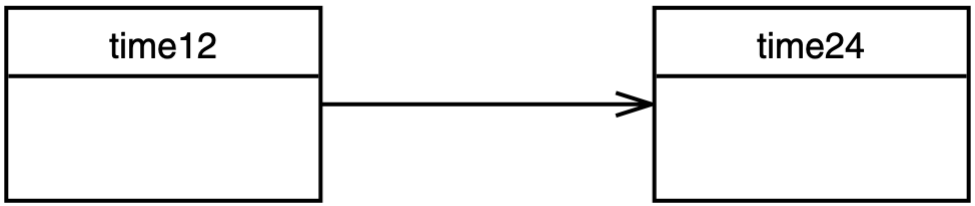


Figure 1: UML class diagram of the TIMES1 program.

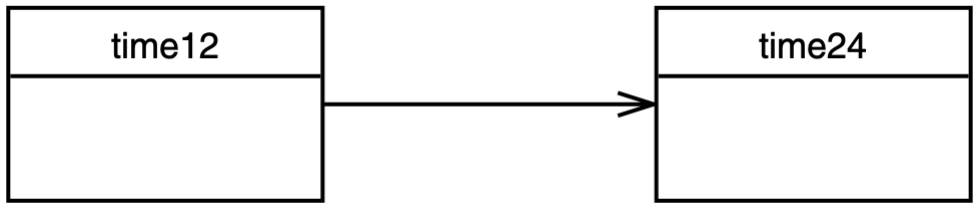


Figure 2: UML class diagram of the TIMES1 program.

- Each class rectangle is divided into sections by horizontal lines.
- The class name goes in the top section.
- We can include sections for member data (called attributes in the UML) and member functions (called operations).

Associations

- Classes may have various kinds of relationships with each other.
- The classes in TIMES1 are related by association.
- We indicate this with a line connecting their rectangles.
- What constitutes an association?
- Conceptually, the real-world entities that are represented by classes in the program have some kind of obvious relationship.
- Drivers are related to cars, books are related to libraries, race horses are related to race tracks.
- If such entities were classes in a program, they would be related by association.

Associations

- In the TIMES2.CPP program, we can see that class time12 is associated with class time24 because we are converting objects of one class into objects of the other.
- A class association actually implies that objects of the classes, rather than the classes themselves, have some kind of relationship.
- Typically, two classes are associated if an object of one class calls a member function (an operation) of an object of the other class.
- An association might also exist if an attribute of one class is an object of the other class.

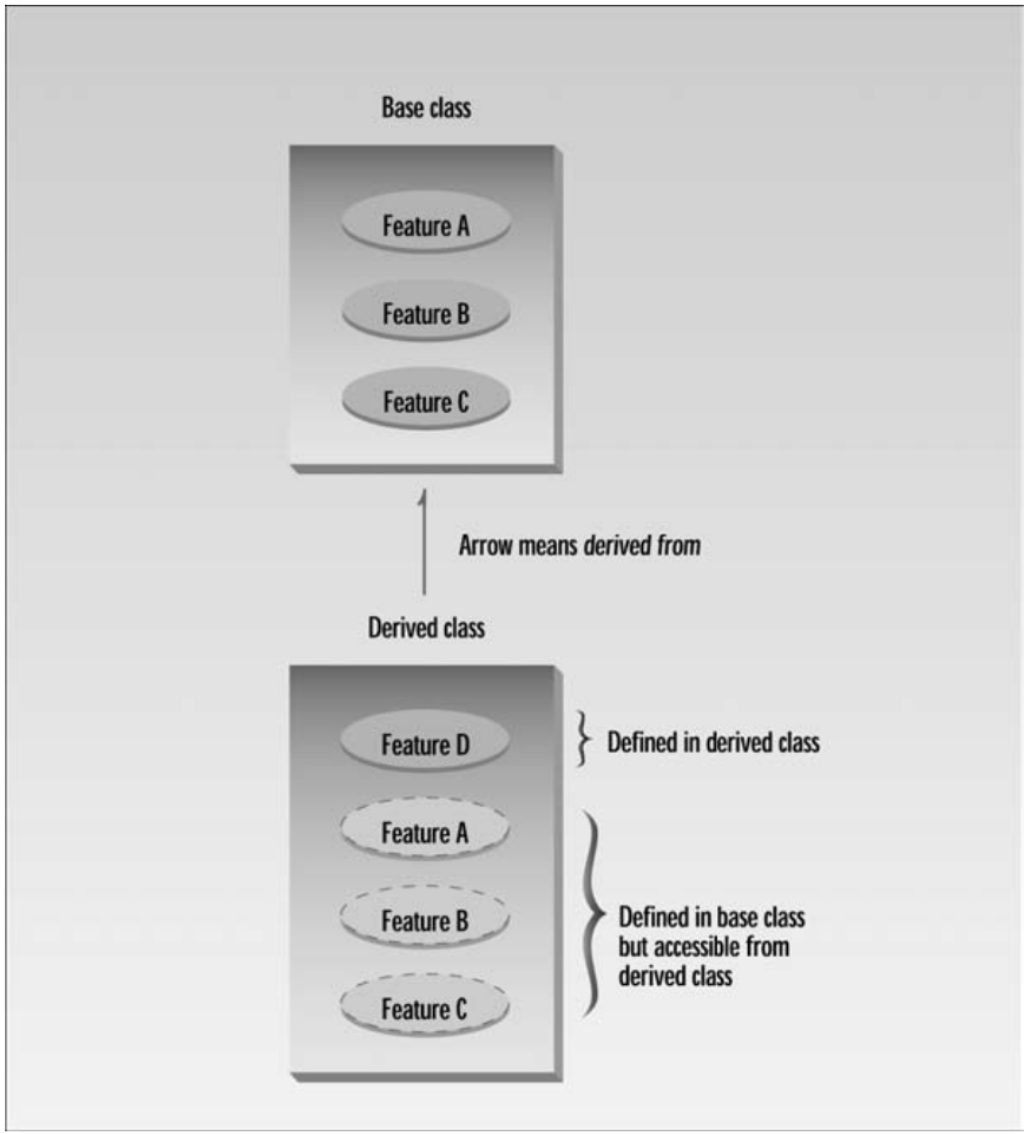
Navigability

- We can add an open arrowhead to indicate the direction or navigability of the association.
- Because time12 calls time24, the arrow points from time12 to time24.
- It's called a unidirectional association because it only goes one way.
- If each of two classes called an operation in the other, there would be arrowheads on both ends of the line and it would be called a bidirectional association.

Inheritance

- Inheritance is probably the most powerful feature of object-oriented programming, after classes themselves.
- Inheritance is the process of creating new classes, called derived classes, from existing or base classes.
- The derived class inherits all the capabilities of the base class but can add embellishments and refinements of its own.
- The base class is unchanged by this process.
- An important result of re-usability is the ease of distributing class libraries.
- A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

The inheritance relationship



- The arrow in the figure goes in the opposite direction of what one might expect.
- If it pointed down we would label it *inheritance*.
- However, the more common approach is to point the arrow up, from the derived class to the base class, and to think of it as a "derived from" arrow.

The inheritance relationship

- Inheritance is an essential part of OOP.
- Once a base class is written and debugged, it need not be touched again,
- but, using inheritance, can nevertheless be adapted to work in different situations.
- Reusing existing code saves time and money and increases a program's reliability.
- Inheritance can also help in the original conceptualisation of a programming problem, and in the overall design of the program.

Derived Class and Base Class

- In ”COUNTPP3” the program used a class Counter as a general-purpose counter variable.
- A count could be initialised to 0 or to a specified number with constructors, incremented with the ++ operator, and read with the `get_count()` operator.
- Let’s suppose that we have worked long and hard to make the Counter class operate just the way we want.
- we’re counting people entering a bank, and we want to increment the count when they come in
- and now lets say we want to decrement it when they go out, so that the count represents the number of people in the bank at any moment.

- Now we really need a way to decrement the count.
- Easiest way is to insert a decrement routine directly into the source code of the `Counter` class.
- There are several reasons that we might not want to do this.
- First, the class works very well and may have undergone many hours of testing and debugging.
- Of course that's an exaggeration in this case,
- but it would be true in a larger and more complex class.
- If we start fooling around with the source code
  - the testing process will need to be carried out again,
  - off course we may foul something up and,
  - spend hours debugging code that worked fine before we modified it

Derived Class and Base Class

- In some situations there might be another reason for not modifying the Counter class:
- We might not have access to its source code, especially if it was distributed as part of a class library. (to be discussed later)
- To avoid these problems we can use inheritance to create a new class based on Counter, without modifying Counter itself.

Specifying the Derived Class

- see ”counten.cpp”
- The listing starts off with the Counter class, which has not changed !!
- with one small exception, which we’ll look at later
- we haven’t modeled this program on the POSTFIX program
- Following the `Counter` class in the listing is the specification for a new class, `CountDn`.
- This class incorporates a new function, `operator--()`, which decrements the `count`.
- **the key point**—the new `CountDn` class inherits all the features of the `Counter` class.
- The first line of `CountDn` specifies that it is derived from `Counter`:

```
class CountDn : public Counter
```

- only a single colon (not the double colon used for the scope resolution operator) is used, followed by the keyword `public` and the name of the base class `Counter`.

Specifying the Derived Class

- This sets up the relationship between the classes.
- it says that `CountDn` is derived from the base class `Counter`.
- `CountDn` doesn't need a constructor or the `get_count()` or `operator++()` functions, because these already exist in `Counter`.

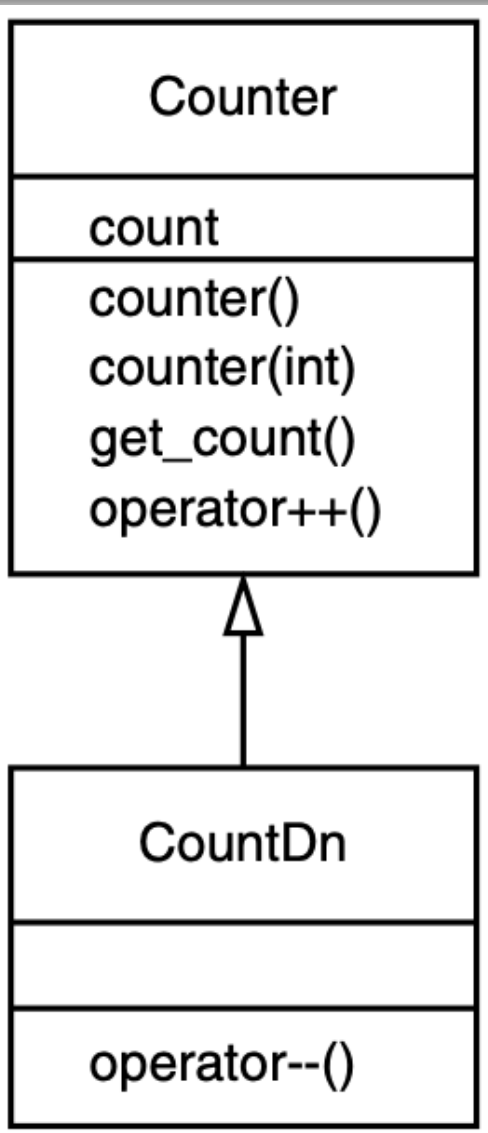
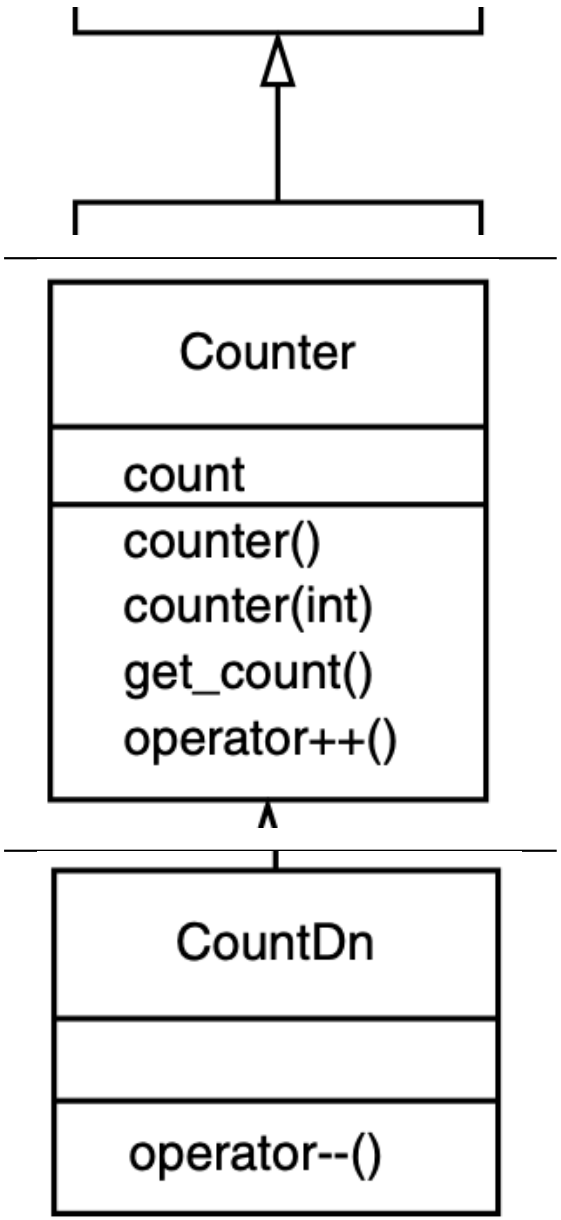


Figure 3: UML class diagram for COUNTEN

- In the UML, inheritance is called generalization, because the parent class is a more general form of the child class.
- Or to put it another way, the child is more specific version of the parent.
- In UML class diagrams, generalization is indicated by a triangular arrowhead on the line connecting the parent and child classes.
- *the arrow means inherited from or derived from or is a more specific version of.*

UML Class Diagrams



- The direction of the arrow emphasises that the derived class refers to functions and data in the base class, while the base class has no access to the derived class.
- Notice that we've added attributes (member data) and operations (member functions) to the classes in the diagram.
- The top area holds the class title, the middle area holds attributes, and the bottom area is for operations.

Accessing Base Class Members

- An important topic in inheritance is *accessibility*,
- i.e. knowing when a member function in the base class can be used by objects of the derived class
- Let's see how the compiler handles the accessibility:

Substituting Base Class Constructors

- In the `main()` part of COUNTEN we create an object of class `CountDn`:

```
CountDn c1;
```

- This causes `c1` to be created as an object of class `CountDn` and initialised to 0.
- There is no constructor in the `CountDn` class specifier, so what entity carries out the initialisation?
- It turns out that—at least under certain circumstances—if we don't specify a constructor, the derived class will use an appropriate constructor from the base class.
- This flexibility on the part of the compiler—using one function because another isn't available— appears regularly in inheritance situations.

Substituting Base Class Member Functions

- The object c1 of the CountDn class also uses the `operator++()` to increment c1:
  `++c1;`
- and `get_count()` functions to display the count in c1:
  `cout << "\nc1=" << c1.get_count();`
- These two functions are from the Counter class.
- Again the compiler, not finding these functions in the class of which `c1` is a member, uses member functions from the base class.

The protected Access Specifier

- It may appear we have increased the functionality of a class without modifying it.
- In reality, its almost without modifying it.
- There is just the single change that we made to the `Counter` class.
- The data in the previous classes we've looked at so far, including count in the Counter class, have used the private access specifier.
- In the Counter class in "COUNTEN", count is given a new specifier: `protected`.
- What does this do?
- We know a member function of a class can always access class members, whether they are public or private.
- But an object declared externally can only invoke (using the dot operator, for example) public members of the class.
- It's not allowed to use private members.

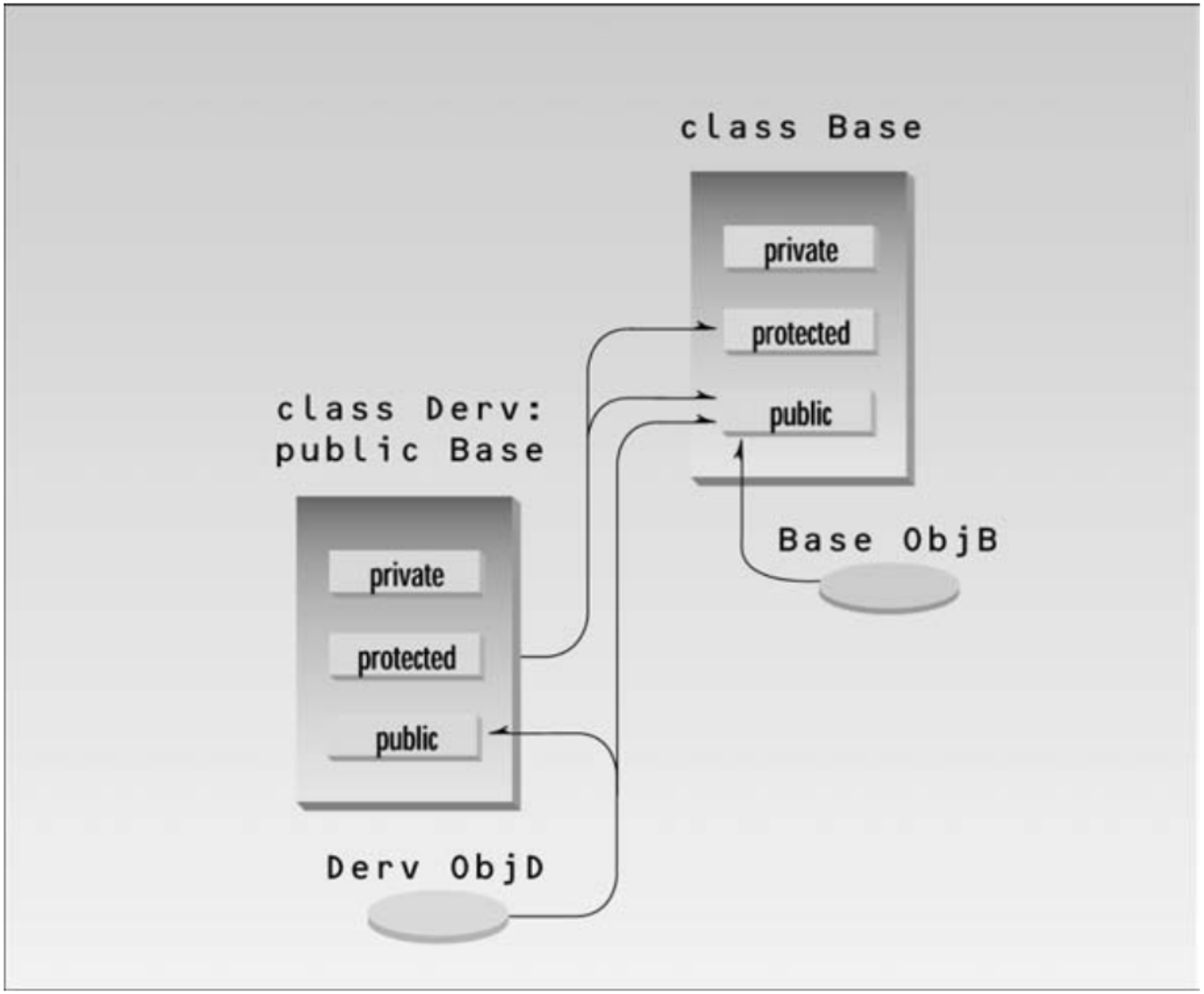
The protected Access Specifier

- suppose an object `objA` is an instance of class `A`, and function `funcA()` is a member function of `A`.
- Then in `main()` or any other function that is not a member of `A`, for the statement
  `objA.funcA();` will not be legal unless `funcA()` is `public`.
- The object `objA` cannot invoke private members of class `A`.
- Private members are, well, private.
- With inheritance, however, there is a whole raft of additional possibilities.
- The question that concerns us at the moment is, can member functions of the derived class access members of the base class?
- The answer is that member functions can access members of the base class if the members are `public`, or if they are `protected`.
- They can't access `private` members.

The protected Access Specifier

- We don't want to make count public, since that would allow it to be accessed by any function anywhere in the program
- and eliminate the advantages of data hiding.
- A protected member, on the other hand, can be accessed by
  - member functions in its own class or—and
  - any class derived from its own class.
- It can't be accessed from functions outside these classes, such as main().
- This is just what we want.

Access specifiers with inheritance.



The protected Access Specifier

Access Specifier	*Accessible from Own Class*	*Accessible from Derived Class*	*Accessible from Objects Outside Class*
public	yes	yes	yes
protected	yes	yes	no
private	yes	no	no

- The moral is that if we are writing a class that we suspect might be used, at any point in the future, as a base class for other classes,
- then any member data that the derived classes might need to access should be made protected rather than private.
- This ensures that the class is "inheritance ready."

Dangers of protected

- there's a disadvantage to making class members protected.
- Say we've written a class library, which we're distributing to the public.
- Any programmer who buys this library can access protected members of your classes simply by deriving other classes from them.
- This makes protected members considerably less secure than private members.
- To avoid corrupted data, it's often safer to force derived classes to access data in the base class using only public functions in the base class,
- just as ordinary `main()` programs must do.
- Using the `protected` specifier leads to simpler programming,
- We need to weigh the advantages of protected against its disadvantages in our own programs.

Base Class Unchanged

- even if other classes have been derived from it, the base class remains unchanged.
- In the `main()` part of "COUNTEN", we could define objects of type Counter:
  `Counter c2;` ⟵ `object of base class`
- Such objects would behave just as they would if `CountDn` didn't exist.
- The base class and its objects don't know anything about any classes derived from the base class.
- inheritance doesn't work in reverse.
- In this example that means that objects of class `Counter`, such as `c2`, can't use the `operator--()` function in `CountDn`.
- If we want a counter that you can decrement, it must be of class `CountDn`, not `Counter`.

Derived Class Constructors

- There's a potential glitch in the "COUNTEN" program.
- What happens if we want to initialise a `CountDn` object to a value?
- Can the one-argument constructor in Counter be used?
- The answer is no.
- the compiler will substitute a no-argument constructor from the base class,
- but it draws the line at more complex constructors
- To make such a definition work we must write a new set of constructors for the derived class.

Derived Class Constructors

- See ”counten2.cpp”
- This program uses two new constructors in the `CountDn` class.
- Here is the no-argument constructor:
  `CountDn() : Counter()`
- This constructor has an unfamiliar feature: the function name following the colon.
- This construction causes the `CountDn()` constructor to call the `Counter()` constructor in the base class.
- In main(), when we say
  `CountDn c1;`
  the compiler will create an object of type `CountDn` and then call the `CountDn` constructor to initialize it.
- This constructor will in turn call the `Counter` constructor, which carries out the work.
- The `CountDn()` constructor could add additional statements of its own, but in this case it doesn't need to, so the function body between the braces is empty.

Derived Class Constructors

- The statement
  `CountDn c2(100);`
  in `main()` uses the one-argument constructor in `CountDn`.
- This constructor also calls the corresponding one-argument constructor in the base class:

```
CountDn(int c) :  Counter(c)   ⟵ argument
c is passed to Counter
  { }
```

- This construction causes the argument c to be passed from `CountDn()` to `Counter()`, where it is used to initialize the object.
- In `main()`, after initializing the `c1` and `c2` objects, we increment one and decrement the other and then print the results.

Overriding Member Functions

- We can use member functions in a derived class that override—that is, have the same name as—those in the base class.
- We might want to do this so that calls in our program work the same way for objects of both base and derived classes.
- See ”STAKARAY”
- Earlier the program modeled a stack, a simple data storage device.
- It allowed to push integers onto the stack and pop them off.
- However, STAKARAY had a potential flaw.
- If we tried to push too many items onto the stack, the program might bomb, since data would be placed in memory beyond the end of the `st[]` array.
- Or if we tried to pop too many items, the results would be meaningless,
- since we would be reading data from memory locations outside the array.

- To cure these defects we've created a new class, Stack2, derived from Stack.
- Objects of Stack2 behave in exactly the same way as those of Stack, except that
- we will be warned if you attempt to push too many items on the stack or,
- if we try to pop an item from an empty stack
- see "staken.cpp"

Which Function Is Used?

- The `Stack2` class contains two functions, `push()` and `pop()`.
- These functions have the same names, and the same argument and return types, as the functions in `Stack`.

Which Function Is Used?

- When we call these functions from main(), in statements like `s1.push(11);`
  how does the compiler know which of the two `push()` functions to use?
- The rule: When the same function exists in both the base class and the derived class, the function in the derived class will be executed.
- Off course this is just true for objects of the derived class.
- Objects of the base class don't know anything about the derived class and will always use the base class functions.
- We say that the derived class function *overrides* the base class function.

Scope Resolution with Overridden Functions

- How do `push()` and `pop()` in `Stack2` access `push()` and `pop()` in `Stack`?
- They use the scope resolution operator, ::, in the statements `Stack::push(var);` and `return Stack::pop();`
- These statements specify that the `push()` and `pop()` functions in `Stack` are to be called.
- Without the scope resolution operator, the compiler would think the `push()` and `pop()` functions in `Stack2` were calling themselves,
- which—in this case—would lead to program failure

Inheritance in the English Distance Class

- English Distance class it was assumed that the distances to be represented would always be positive.
- This is usually the case.
- However, if we were measuring, say, the water level of the Pacific Ocean as the tides varied, we might want to be able to represent negative feet-and-inches quantities.
- Tide levels below mean-lower-low-water are called minus tides; they prompt clam diggers to take advantage of the larger area of exposed beach.
- Let's derive a new class from Distance.
- This class will add a single data item to our feet-and- inches measurements: a sign, which can be positive or negative.
- When we add the sign, we'll also need to modify the member functions so they can work with signed distances.

Inheritance in the English Distance Class

- The `DistSign` class is derived from `Distance`.
- The `Distance` class in this program is just the same as in previous programs, except that the data is protected.
- Actually in this case it could be private, because none of the derived-class functions accesses it.
- it's safer to make it protected so that a derived-class function could access it if necessary.
- It adds a single variable, `sign`, which is of type `posneg`.
- The sign variable will hold the sign of the distance.
- The `posneg` type is defined in an `enum` statement to have two possible values: `pos` and `neg`.
- See "englen.cpp"

Constructors in `DistSign`

- `DistSign` has two constructors, mirroring those in `Distance`.
- The first takes no arguments, the second takes either two or three arguments.
- The third, optional, argument in the second constructor is a sign, either `pos` or `neg`.
- Both constructors in `DistSign` call the corresponding constructors in Distance to set the feet- and-inches values.
- They then set the sign variable.
- The no-argument constructor always sets it to `pos`.
- The second constructor sets it to `pos` if no third-argument value has been provided, or to a value (`pos` or `neg`) if the argument is specified.
- The arguments `ft` and `in`, passed from `main()` to the second constructor in `DistSign`, are simply forwarded to the constructor in `Distance`.