Software Design and Analysis for Engineers

by

Dr. Lesley Shannon

Email: lshannon@ensc.sfu.ca

Course Website: http://www.ensc.sfu.ca/~lshannon/courses/ensc251

Simon Fraser University

Slide Set: 2

Date: September 14, 2015

What we’re learning in this Slide Set:

- Using Custom Data Types:
  - Constructors
  - Abstract Data Types
  - Inheritance

Textbook Chapters:

Relevant to this slide set:

- Sections 10.2-10.4

Coming soon:

- Chapter 11
- Reminders from Chapter 5 (Sections 5.3 and 5.4)

Object Initialization

When you instantiate a variable, you often want it to have a known starting value.

For example:

```
    for (int i=0; i<100; i++)
```

Object Initialization

The same is true for member variables in objects.

By initializing them when you instantiate an object, you have a “known” starting condition*

-This is *very* helpful

*There are other initializing actions you can take but we will talk about it later

Constructors

C++ provides a special type of member function for initialization called a ***constructor***

A constructor is automatically called when an object is declared (i.e. “instantiated).

Constructors perform the necessary initialization as part of the creation of the object, such as initializing member variables

Constructors are defined as member functions, but their definition has some specific constraints

Constructors

```
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.

    void set(int dollars, int cents, double rate);
    void set(int dollars, double rate);
    void update();

    double get_balance();
    double get_rate();
    void output(ostream& outs);
private:
    double balance;
    double interest_rate;
    double fraction(double percent);
};
```

1. ***A constructor must have the same name as the class***

Constructors

```
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.

    void set(int dollars, int cents, double rate);
    void set(int dollars, double rate);
    void update();

    double get_balance();
    double get_rate();
    void output(ostream& outs);
private:
    double balance;
    double interest_rate;
    double fraction(double percent);
};
```

2. ***A constructor cannot return a value; it cannot even have a return value stated in its declaration***

Constructors

```
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.

    void set(int dollars, int cents, double rate);
    void set(int dollars, double rate);
    void update();

    double get_balance();
    double get_rate();
    void output(ostream& outs);
private:
    double balance;
    double interest_rate;
    double fraction(double percent);
};
```

3. ***A constructor should be a public member function***

Otherwise you won't be able to declare objects of this class type

Constructors

Since the constructor member function looks like:

```
BankAccount(int dollars, int cents, double rate);
//Initializes the account balance to $dollars.cents and
//initializes the interest rate to rate percent.
```

You can now declare objects of type BankAccount and initialize them:

```
BankAccount account1(10, 50, 2.0), account2(500, 0, 4.5);
```

Constructors

Note, this declaration doesn't break our rule of not accessing member variables outside of an object, because the constructor variables and member variables are not the same

```
BankAccount(int dollars, int cents, double rate);
//Initializes the account balance to $dollars.cents and
//initializes the interest rate to rate percent.
```

Versus

```
double balance;
double interest_rate;
double fraction(double percent);
```

Remember ***SCOPE***

Here's some sample code of our constructor

```
BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    balance = dollars + 0.01*cents;
    interest_rate = rate;
}
```

The first BankAccount refers to the class, the second refers to the member function (the constructor).

Note that there is no specified return type (not even void) or return statement.

Otherwise this is the same as a normal member function

Other Notes on Constructors:

- You cannot call constructors in your main program the way you call normal member functions:

```
BankAccount account1, account2; //PROBLEMS--BUT FIXABLE
account1.BankAccount(10, 50, 2.0); //VERY ILLEGAL
account2.BankAccount(500, 0, 4.5); //VERY ILLEGAL
```

Although this is what “automagically” happens when you declare a BankAccount Object with our defined constructor, you cannot call them in your executable code

What is wrong with the first line of the declaration?

How could you fix it?

Other Notes on Constructors:

- You can overload constructors (this is actually very common as you may only be able to initialize some of the member variables:

```
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.

    BankAccount(int dollars, double rate);
    //Initializes the account balance to $dollars.00 and
    //initializes the interest rate to rate percent.

    BankAccount( );
    //Initializes the account balance to $0.00
    //and the interest rate to 0.0%.
                     •     •     •
```

```
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.

    BankAccount(int dollars, double rate);
    //Initializes the account balance to $dollars.00 and
    //initializes the interest rate to rate percent.

    BankAccount( );
    //Initializes the account balance to $0.00
    //and the interest rate to 0.0%.
```

Possible Declarations:

```
    BankAccount account1(100, 2.3);

    BankAccount account2;
```

But ***not***:

```
    BankAccount account2(); //WRONG! DO NOT DO THIS!
```

Note: if you are only using your mutator functions to initialize your objects, you don’t need them anymore

You can use Constructors

However, if you want to be able to “set” your object’s after initialization, then you still need your mutator functions.

Each time you call a constructor you create and initialize a new object.

An example from the text:

```
//Program to demonstrate the class BankAccount.
#include <iostream>
using namespace std;

//Class for a bank account:
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    //Initializes the account balance to $dollars.cents and
    //initializes the interest rate to rate percent.

    BankAccount(int dollars, double rate);
    //Initializes the account balance to $dollars.00 and
    //initializes the interest rate to rate percent.

    BankAccount( );
    //Initializes the account balance to $0.00
    //and the interest rate to 0.0%.
```

This definition of `BankAccount` *is an improved version of the class* `BankAccount` *given in Display 10.5.*

… (We'll skip the rest of the class definition)

An example from the text (cont'd):

This declaration causes a call to the default constructor. Notice that there are no parentheses.

```
int main( )
{
    BankAccount account1(100, 2.3), account2;

    cout << "account1 initialized as follows:\n";
    account1.output(cout);
    cout << "account2 initialized as follows:\n";
    account2.output(cout);

    account1 = BankAccount(999, 99, 5.5);
    cout << "account1 reset to the following:\n";
    account1.output(cout);
    return 0;
}
```

An explicit call to the constructor `BankAccount::BankAccount`

```
BankAccount::BankAccount(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    balance = dollars + 0.01 * cents;
    interest_rate = rate;
}

BankAccount::BankAccount(int dollars, double rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
    balance = dollars;
    interest_rate = rate;
}

BankAccount::BankAccount( ) : balance(0), interest_rate(0.0)
{
    //Body intentionally empty
}
```

Let’s talk more about Constructors with initialization sections:

```
BankAccount::BankAccount( ) : balance(0), interest_rate(0.0)
{
    //Body intentionally empty
}
```

The new part starts with a colon

It is called the ***initialization*** section

-Starts after the closing brace of the parameter list

-Ends before the opening brace of the function body

-The colon is followed by a list of some subset (or all) of the member variables with their initialization values in parentheses and the individual variables separated by commas.

In fact this:

```
BankAccount::BankAccount( ) : balance(0), interest_rate(0.0)
{
    //Body intentionally empty
}
```

is equal to this:

```
BankAccount::BankAccount( )
{
    balance = 0;
    interest_rate = 0.0;
}
```

Constructors with parameters can have initialization sections, and the initialization values need not be constant (the constructor's parameters can be used).

Also, having an initialization section does not preclude code in the function body:

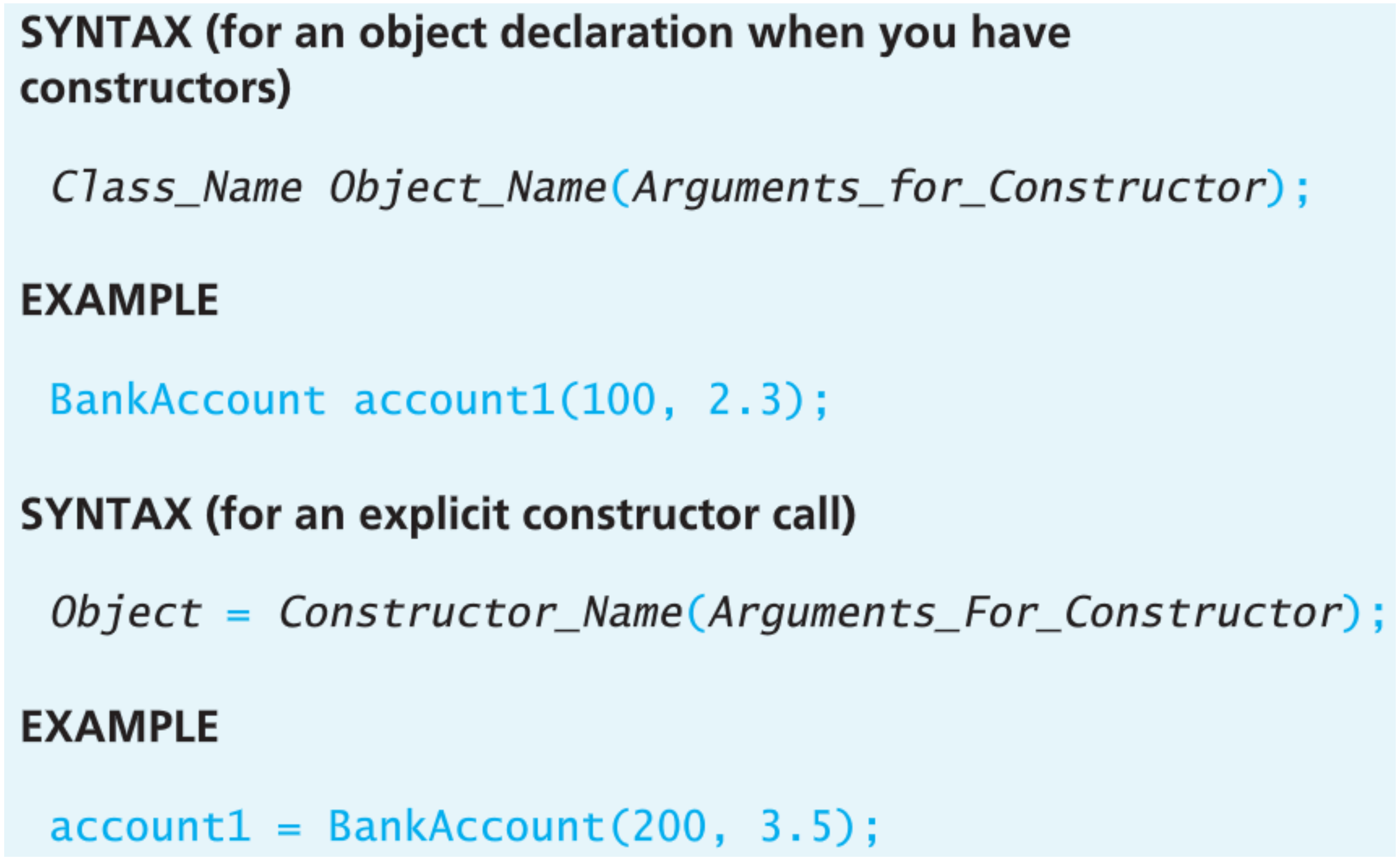
```
BankAccount::BankAccount(int dollars, double rate)
            : balance(dollars), interest_rate(rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }
}
```

Here’s another example:

```
BankAccount::BankAccount(int dollars, int cents,
                         double rate)
    : balance(dollars + 0.01*cents), interest_rate(rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout <<
           "Illegal values for money or interest rate.\n";
        exit(1);
    }
}
```

Your initialization section can have simple equations in terms of function parameters and constants

So the two ways to call a constructor are:



SYNTAX (for an object declaration when you have constructors)

```
Class_Name Object_Name(Arguments_for_Constructor);
```

EXAMPLE

```
BankAccount account1(100, 2.3);
```

SYNTAX (for an explicit constructor call)

```
Object = Constructor_Name(Arguments_For_Constructor);
```

EXAMPLE

```
account1 = BankAccount(200, 3.5);
```

When you make an explicit call to a constructor, you create an anonymous object

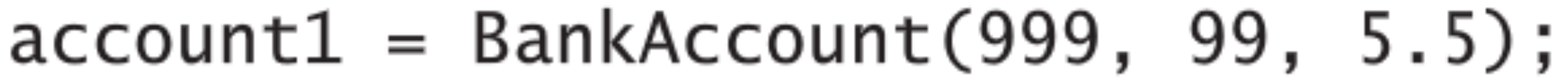
An anonymous object is not named by any variable

As such when you make the following assignment, you assign the anonymous object to the named class variable:

```
account1 = BankAccount(999, 99, 5.5);
```

Note: this is not an efficient way to change member variables as the constructor creates a whole new object and then assigns it

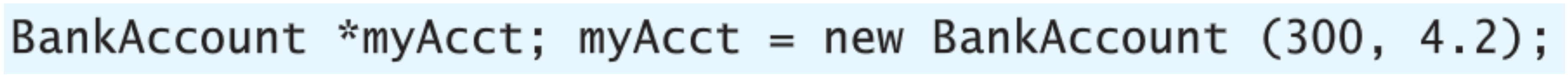
That’s why you want to use a mutator function to change the values of member variables after initialization



```
account1 = BankAccount(999, 99, 5.5);
```

So why is it important to be able to explicitly call a constructor?

Hint: Remember dynamic memory allocation



```
BankAccount *myAcct; myAcct = new BankAccount (300, 4.2);
```

When you create linked lists and dynamically generate objects, you need to be able to initialize them.

Remember this because it will be important later.

Some important things to remember

If you don’t explicitly create a constructor, the compiler will create a default constructor (one with no arguments) that does nothing.

However, once you have defined at least one constructor, then no default constructors will be provided.

As such, you need to be sure that every instantiation of an object has a parameter list matching one of your constructors, or create a default constructor.

For example:

```
class SampleClass
{
public:
    SampleClass(int parameter1, double parameter2);
    void do_stuff();
private:
    int data1;
    double data2;
};
```

Constructor that requires two arguments

This will work:

```
SampleClass my_object(7, 7.77);
```

But this is illegal:

```
SampleClass your_object;
```

WHY?

Since you may not always want/know an object’s initialization parameters, always include a default constructor:

```
class SampleClass
{
public:
    SampleClass(int parameter1, double parameter2);
    SampleClass();             Default constructor
    void do_stuff();
private:
    int data1;
    double data2;
};
```

Giving your default constructor an empty body will have the same behaviour as the C++ compiler’s default constructor.

It will do nothing:

```
SampleClass::SampleClass()
{
    //Do nothing.
}
```

If an object of a class type is created as a dynamic variable using the new operator, then the default constructor is invoked unless you include initializers.

```
BankAccount *myAcct; myAcct = new BankAccount (300, 4.2);
```

Remember we said this was bad:

```
    BankAccount account2(); //THIS WILL CAUSE PROBLEMS.
```

and that you could use:

```
    BankAccount account2;
```

or:

```
    account1 = BankAccount();
```

Why do you think the first statement is bad?

-Hint: it may not produce an error message since it is not an illegal statement; but the action will likely be unintended.

Final Notes on Constructors

Constructors give you control over how an object is initialized.

Be sure to make them public

Also, create a default constructor (even if it does nothing) to keep your code reuse clean

Don’t use constructors to reinitialize your member variables, use mutator functions

Special notes for C++11

```
class Coordinate
{
  public:
        Coordinate();
        Coordinate(int x);
        Coordinate(int x, int y);
        int getX();
        int getY();
  private:
        int x=1;
        int y=2;
};
Coordinate::Coordinate()
{ }
Coordinate::Coordinate(int xval) : x(xval)
{ }
Coordinate::Coordinate(int xval, int yval) : x(xval), y(yval)
{ }
int Coordinate::getX()
{
  return x;
}
int Coordinate::getY()
{
  return y;
}
```

- You can perform member initialization
- This is useful for critical systems, ensuring that fields always start with known values.

Special notes for C++11

```
class Coordinate
{
  public:
        Coordinate();
        Coordinate(int x);
        Coordinate(int x, int y);
        int getX();
        int getY();
  private:
        int x=1;
        int y=2;
};
~~Coordinate::Coordinate()~~
~~{ }~~
Coordinate::Coordinate(int xval) : x(xval)
{ }
Coordinate::Coordinate(int xval, int yval) : x(xval), y(yval)
{ }
```

Using this to replace the original default constructor:

```
Coordinate::Coordinate() : Coordinate(99,99)
{ }
```

- It also supports ***constructor delegation***
- This allows one constructor to call another constructor (e.g. the default calling the fully specified constructor as shown here)

Special notes for C++11

WARNING:

-These features are new and are not supported in older compilers aimed at older versions of C++

-This creates backwards compatibility issues of which you need to be aware

-So then the question becomes: should you use them?

Now let’s look at Abstract Data Types and take a sneak peek at Inheritance…

Data Types:

Consists of a collection of values ***combined*** with the basic set of operations defined on those values

Abstract Data Types:

Data types are only considered abstract if the programmers using the type cannot access the details of how values are stored and the operations are implemented

Predefined types such as `int` are abstract:

-you don't know exactly how its operators (+, *, etc.) are implemented

-you cannot use this information in your own programs

Abstract Data Types (ADTs):

Programmer defined data types are not automatically abstract:

Remember I originally showed you code for custom data types where the data fields had been made public?

If you let a programmer directly access the class' data fields and manipulate how its operators behave, you cannot maintain encapsulation and guarantee the way objects of this type will behave.

Next will be a series of notes on how to make sure the classes you create can be considered abstract data types.

Recall:

```
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    BankAccount(int dollars, double rate);
    BankAccount();
    void set(int dollars, int cents, double rate);
    void set(int dollars, double rate);
    void update();
    double get_balance();
    double get_rate();
    void output(ostream& outs);
private:
    double balance;
    double interest_rate;
    double fraction(double percent);
};
```

What does the user need to know?

Recall:

```
class BankAccount
{
public:
    BankAccount(int dollars, int cents, double rate);
    BankAccount(int dollars, double rate);
    BankAccount();
    void set(int dollars, int cents, double rate);
    void set(int dollars, double rate);
    void update();
    double get_balance();
    double get_rate();
    void output(ostream& outs);
private:
    double balance;
    double interest_rate;
    double fraction(double percent);
};
```

You could store the balance in terms of dollars and cents (both ints).

Then:

```
class BankAccount
{
public:
    <This part is exactly the same as before>
private:
    int dollars_part;
    int cents_part;
    double interest_rate;
    double fraction(double percent);
};
```

And the function `get_balance()` becomes:

```
double BankAccount::get_balance()
{
    return (dollars_part + 0.01 * cents_part);
}
```

You could also get rid of the function fraction, and change update from:

```
void BankAccount::update()
{
    balance = balance + fraction(interest_rate) * balance;
}
```

to:

```
void BankAccount::update()
{
    balance = balance + (interest_rate / 100.0) * balance;
}
```

The key to creating ADTs (and maintaining encapsulation and reusability): <u>separating the specification of how the type is used by a programmer from how it is implemented.</u>*

*In fact, you should be able to completely change a class' implementation without having to change the programs that use that class

How to maintain separation/abstraction

General Rules:

1) Make all member variables private members of the class
2) Make the basic operations that the programmer needs public member functions and fully specify how to use each member function (i.e. defined interface).
3) Make all helping functions private member functions.

The interface

When you define an ADT, the interface comprises the public member functions of the class ***and the comments that tell a programmer how to use these public functions***

The interface should be all a programmer needs to know to use your class

This means someone can use the interface while someone else works on the implementation…

This also means that once you define the interface, you cannot change it (i.e. the public functions)

The implementation:

How the class is realized in C++

Comprises private member functions and the member variables.

Users should not need to know anything about your implementation to be able to use your ADT

You can “harden” this separation of interface and implementation by placing the interface and implementation in separate files from each other and main (we’ll see this later).

Maintaining your ADTs:

-You can’t change your public function declarations (but you can change their definitions).

-You can’t change the resulting operation(s) of calling your public functions (and you shouldn’t change the comments), but you can change how the results are achieved.

-You can change your private member functions and private member variable types.

In short, as long as any code using your ADT requires NO CHANGES when you change it internally, you have maintained the separation for ADTs

Inheritance & Derived Classes

C++ supports inheritance through derived classes

A derived class is obtained from another class by adding features.

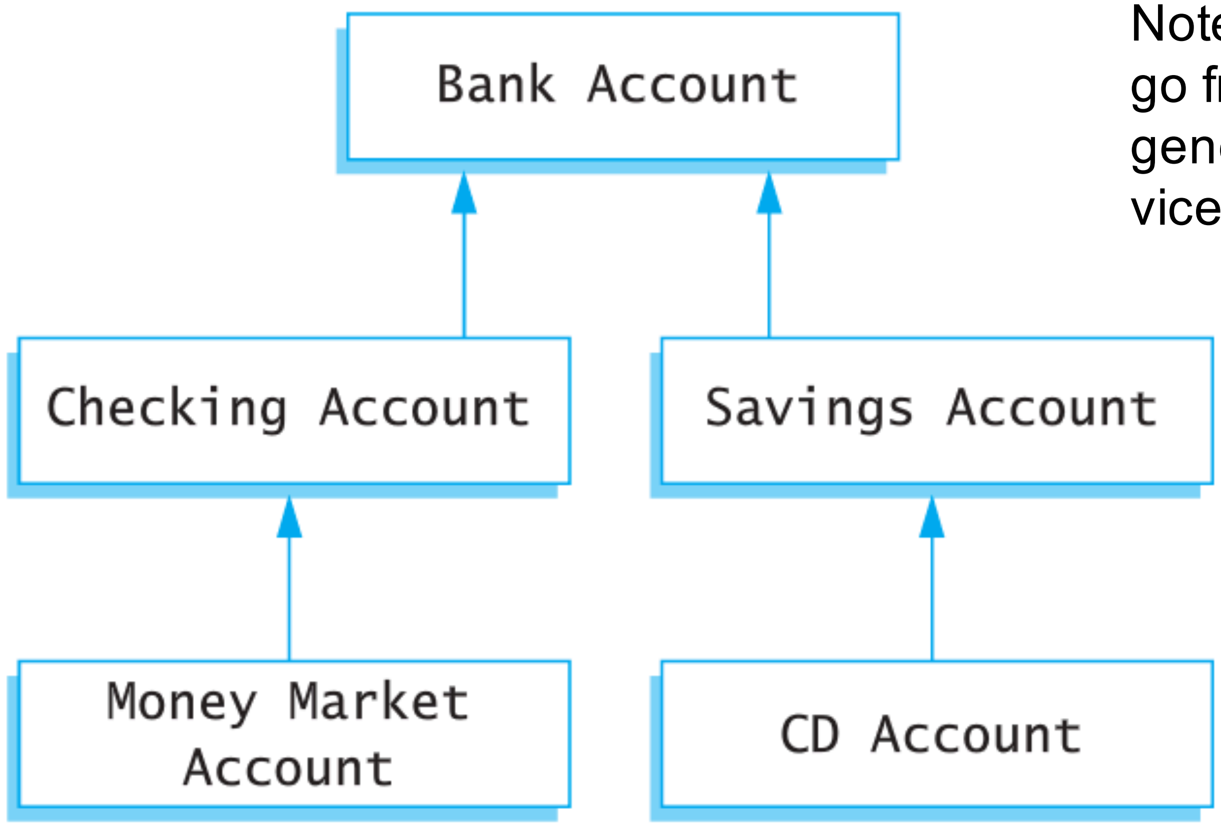
Inheritance allows you to define a general class and then add more specialized features to create a new class.

This means that you need only add the new features to the derived class

For example, the general class could be mammals, and a derived class could be dogs.

*This is only an introduction to the ideas, we’ll talk more about it later.

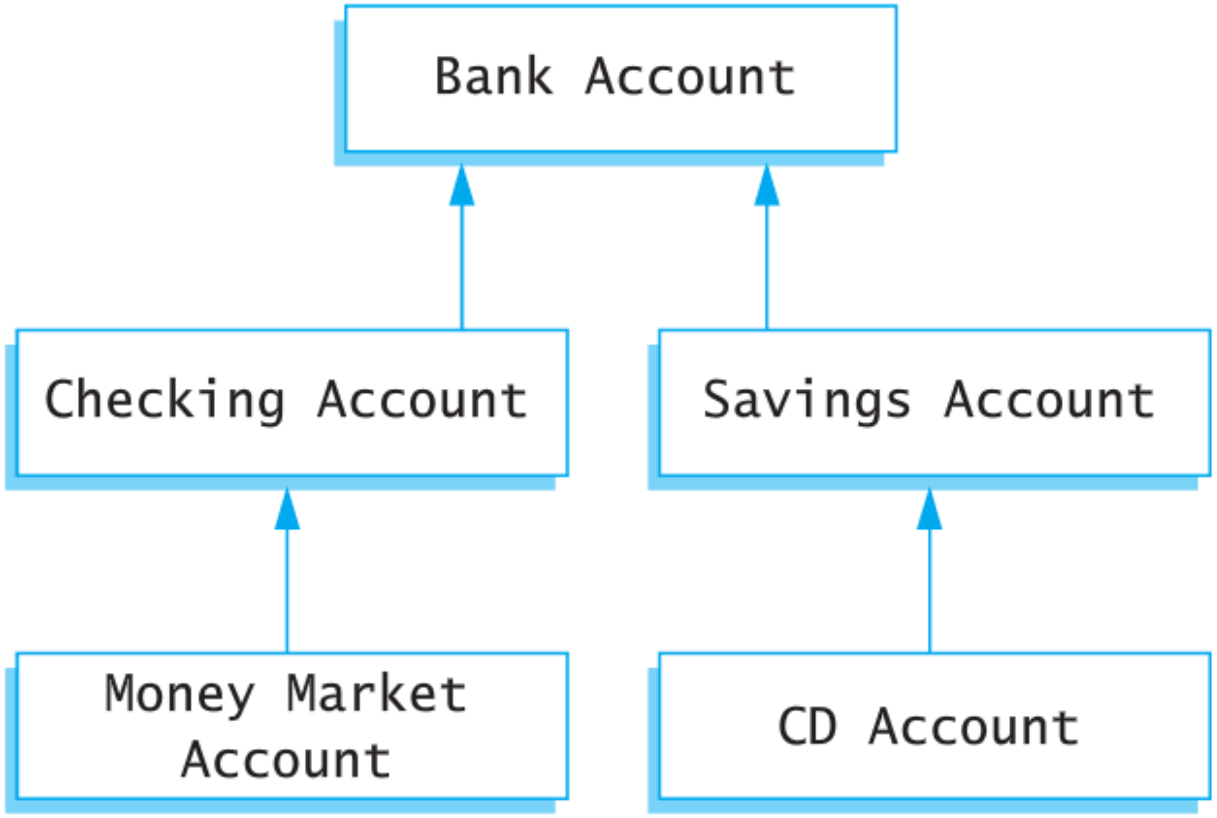
Class Hierarchy



Note: The arrows go from specific to general (and not vice versa)

Checking Account is a derived class of Bank Account; it has all of the features of the Bank Account class as well as some added features (e.g. you can write cheques, make deposits, etc.)

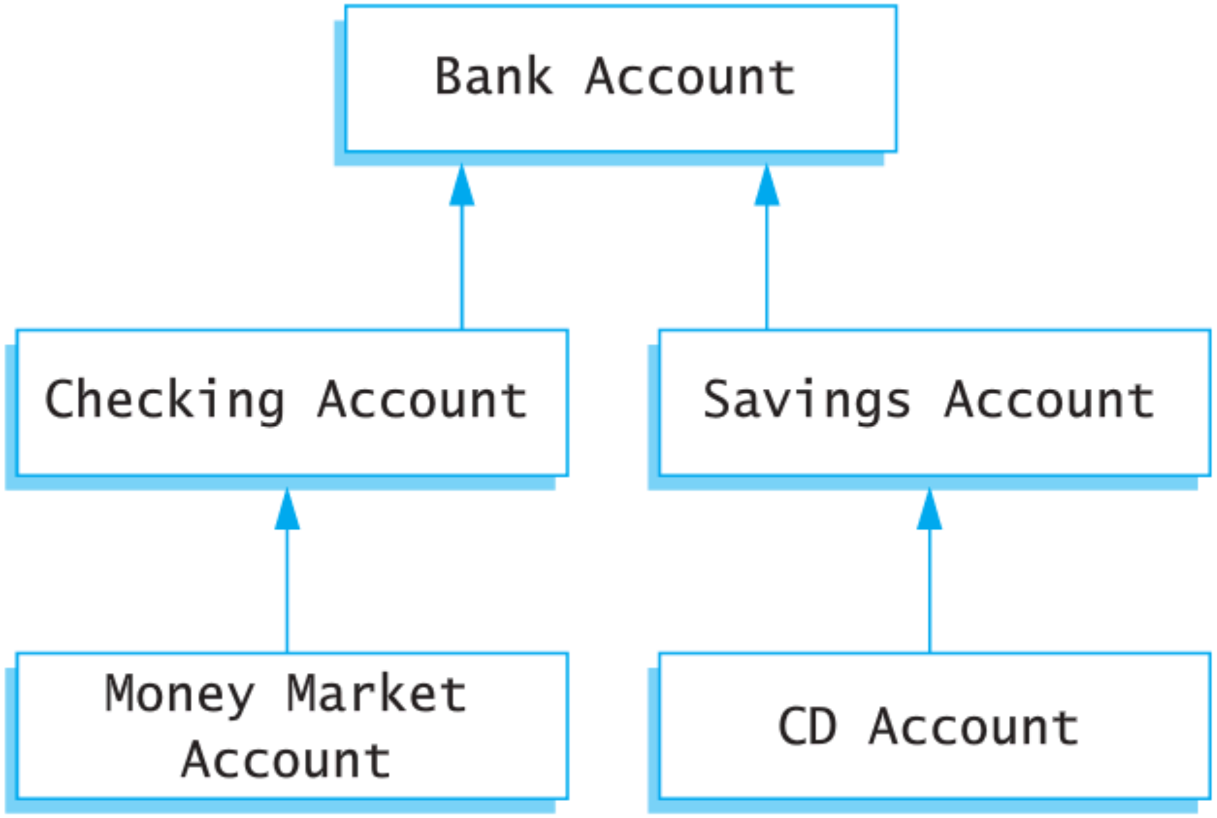
Class Hierarchy



Derived Classes can be derived from other derived classes

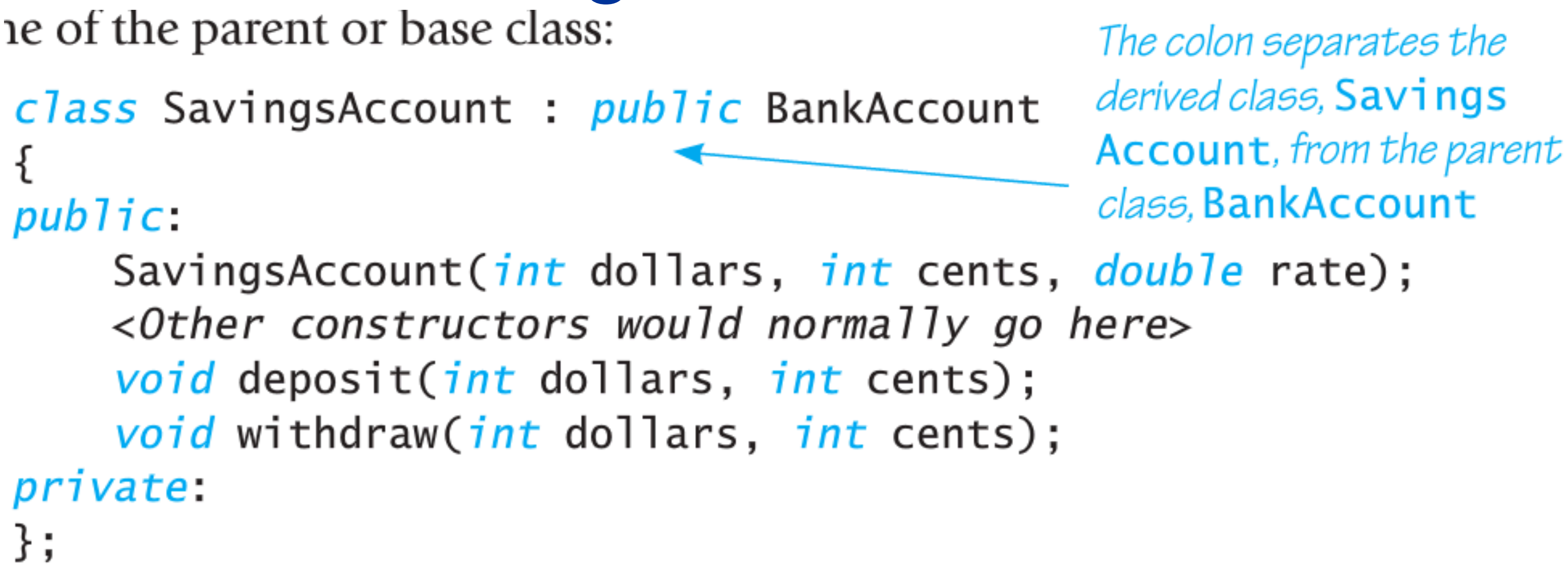
For example, Money Market Accounts are a special case of Checking Accounts which are a special case of Bank Accounts

Class Hierarchy



Since Checking Account is a derived class of Bank Account:
-Checking Account is the ***child*** of Bank Account,
-Bank Account is the ***parent*** or ***base class*** of Checking Account, and
-Checking Account ***inherits*** the member functions of Banking Account

Defining a Derived Class



ıe of the parent or base class:

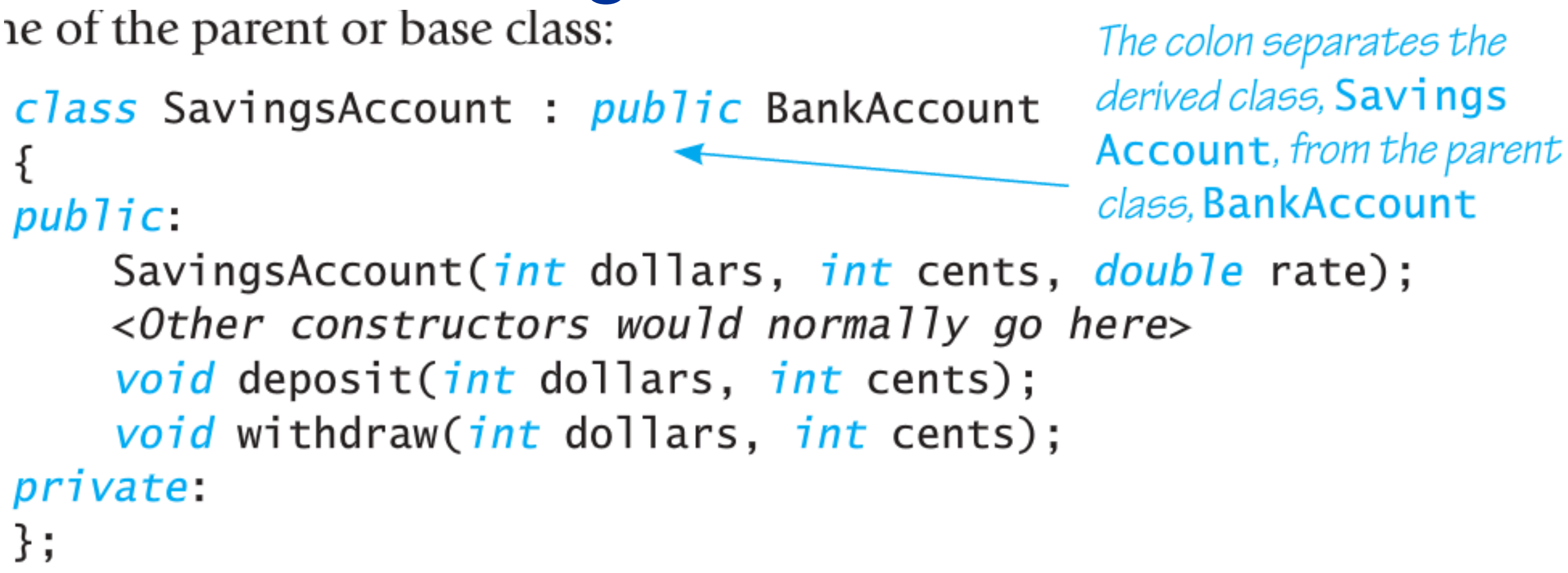
```
class SavingsAccount : public BankAccount
{
public:
    SavingsAccount(int dollars, int cents, double rate);
    <Other constructors would normally go here>
    void deposit(int dollars, int cents);
    void withdraw(int dollars, int cents);
private:
};
```

The colon separates the derived class, `SavingsAccount`*, from the parent class,* `BankAccount`

Any object of class type SavingsAccount, will now have all of the member functions and fields of BankAccount class type plus the additional member functions `deposit` & `withdraw`.

Note that the SavingsAccount class has its own constructor(s)

Defining a Derived Class

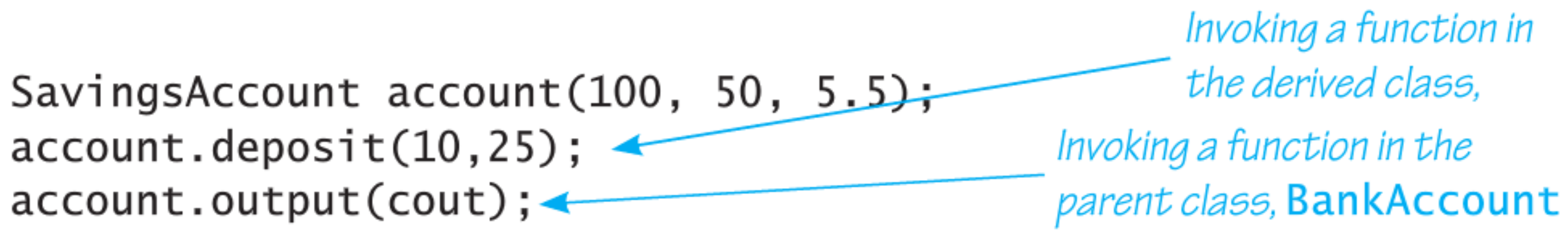


ıe of the parent or base class:

```
class SavingsAccount : public BankAccount
{
public:
    SavingsAccount(int dollars, int cents, double rate);
    <Other constructors would normally go here>
    void deposit(int dollars, int cents);
    void withdraw(int dollars, int cents);
private:
};
```

The colon separates the derived class, `SavingsAccount`*, from the parent class,* `BankAccount`

With this definition all of these calls are legal:

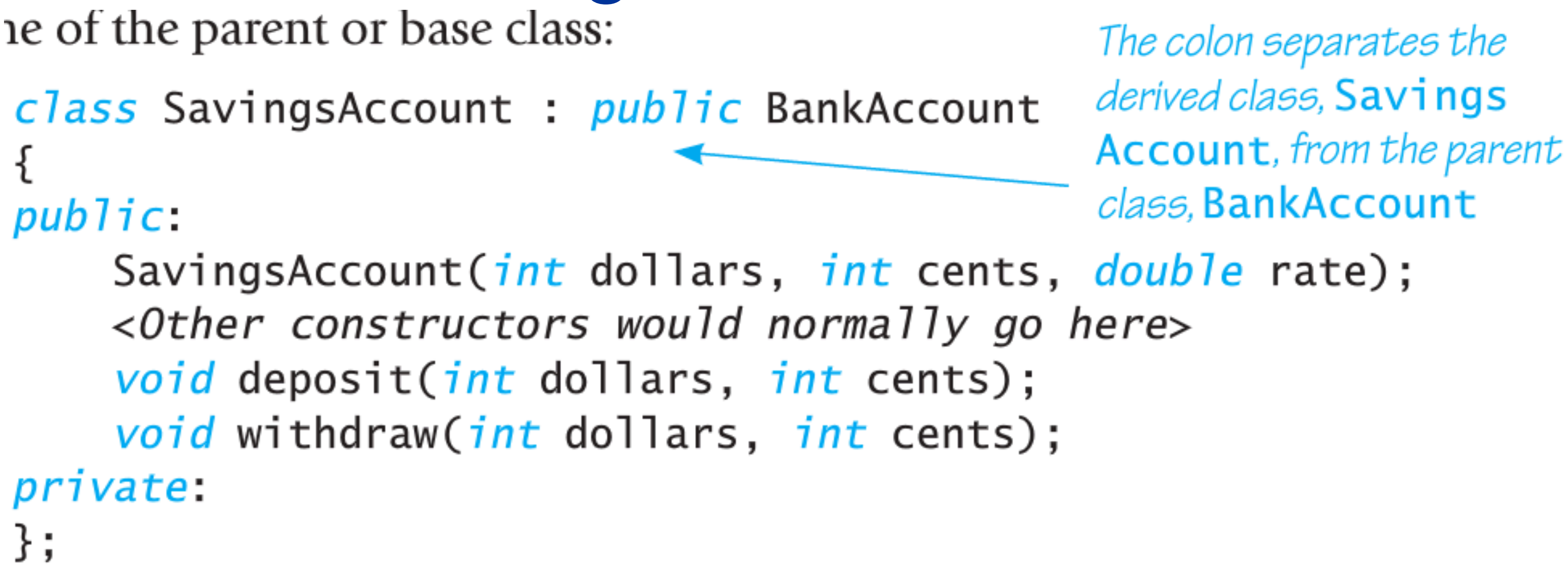


```
SavingsAccount account(100, 50, 5.5);
account.deposit(10,25);
account.output(cout);
```

Invoking a function in the derived class,

Invoking a function in the parent class, `BankAccount`

Defining a Derived Class



ıe of the parent or base class:

```
class SavingsAccount : public BankAccount
{
public:
    SavingsAccount(int dollars, int cents, double rate);
    <Other constructors would normally go here>
    void deposit(int dollars, int cents);
    void withdraw(int dollars, int cents);
private:
};
```

The colon separates the derived class, `SavingsAccount`*, from the parent class,* `BankAccount`

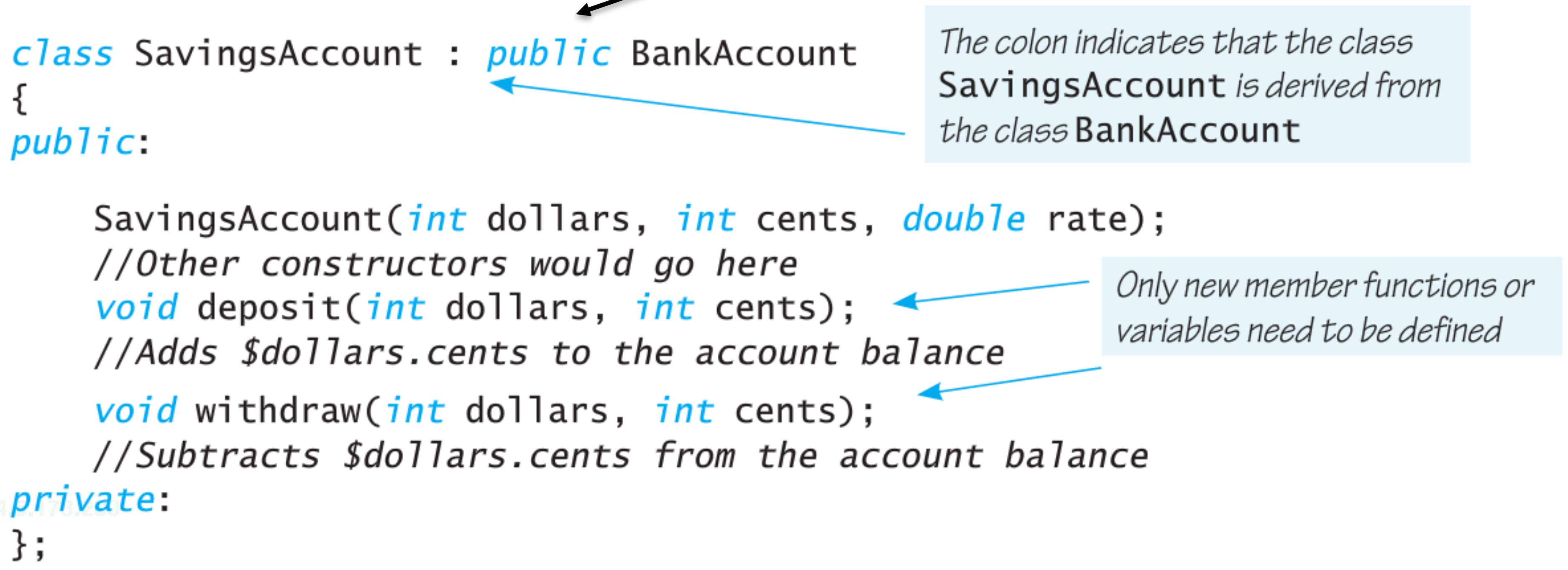
Why is this approach better than simply copying BankAccount class and adding the new functionality?

-saves memory (SavingsAccount definition is smaller)

-facilitates code reuse (only have to modify BankAccount functionality in one place)

Using this class definition

Make sure you use the ***public*** label for BankAccount. I'll explain why when we revisit Inheritance.



```
class SavingsAccount : public BankAccount
{
public:

    SavingsAccount(int dollars, int cents, double rate);
    //Other constructors would go here
    void deposit(int dollars, int cents);
    //Adds $dollars.cents to the account balance

    void withdraw(int dollars, int cents);
    //Subtracts $dollars.cents from the account balance
private:
};
```

The colon indicates that the class `SavingsAccount` *is derived from the class* `BankAccount`

Only new member functions or variables need to be defined

The main function becomes:

```
int main( )
{
    SavingsAccount account(100, 50, 5.5);
    account.output(cout);
    cout << endl;
    cout << "Depositing $10.25." << endl;
    account.deposit(10,25);
    account.output(cout);
    cout << endl;
    cout << "Withdrawing $11.80." << endl;
    account.withdraw(11,80);
    account.output(cout);
    cout << endl;
    return 0;
}
```

The class member functions are:

The **SavingsAccount** *constructor invokes the* **BankAccount** *constructor. Note the preceding colon.*

```
SavingsAccount::SavingsAccount(int dollars, int cents, double rate):
    BankAccount(dollars, cents, rate)
{
    //deliberately empty

}

void SavingsAccount::deposit(int dollars, int cents)
{

    double balance = get_balance();
    balance += dollars;
    balance += (static_cast<double>(cents) / 100);
    int new_dollars = static_cast<int>(balance);
    int new_cents = static_cast<int>((balance - new_dollars) * 100);
     set(new_dollars, new_cents, get_rate());
}

void SavingsAccount::withdraw(int dollars, int cents)
{
    double balance = get_balance();
    balance -= dollars;
    balance -= (static_cast<double>(cents) / 100);
    int new_dollars = static_cast<int>(balance);
    int new_cents = static_cast<int>((balance - new_dollars) * 100);
    set(new_dollars, new_cents, get_rate());
}
```

The **deposit** *function adds the new amount to the balance and changes the member variables via the* **set** *function*

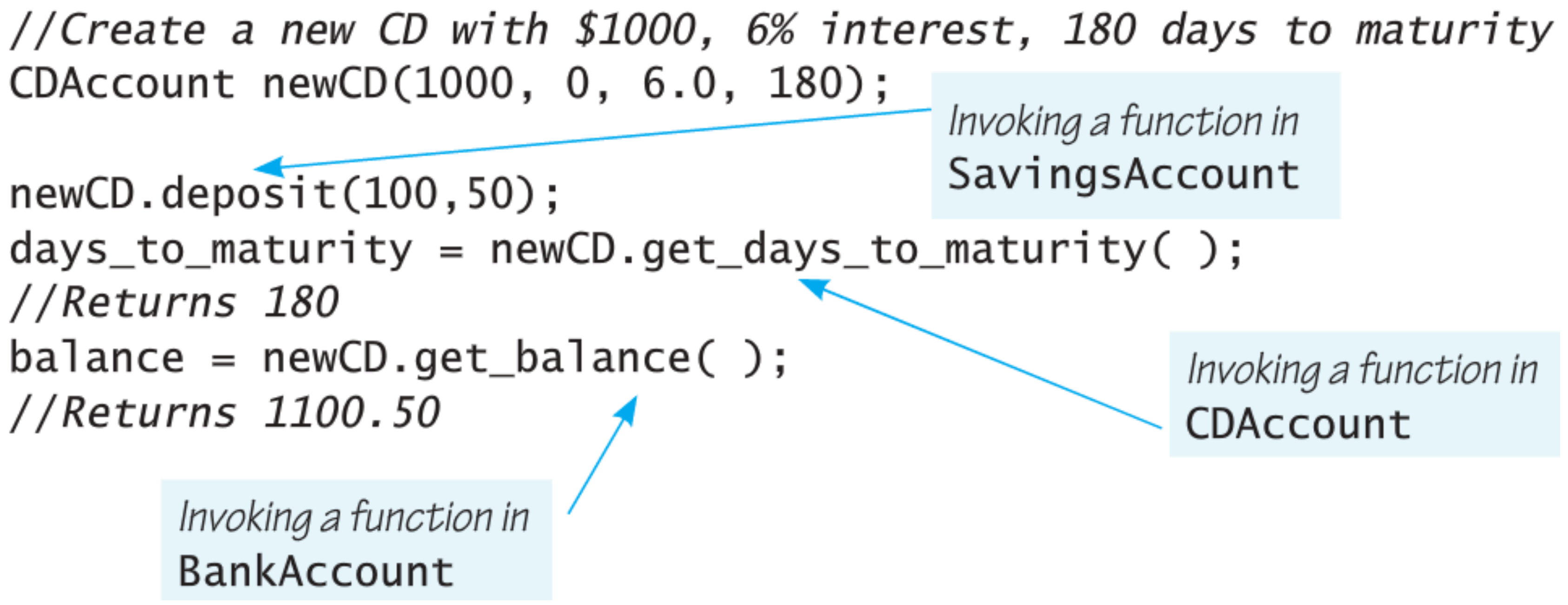
The **withdraw** *function subtracts the amount from the balance and changes the member variables via the* **set** *function*

Deriving from a Derived Class

```
class CDAccount : public SavingsAccount
{
public:
    CDAccount(int dollars, int cents, double rate,
              int days_to_maturity);
    <Other constructors would normally go here>
    int get_days_to_maturity( );
    //Returns the number of days until the CD matures
    void decrement_days_to_maturity( );
    //Subtracts one from the days_to_maturity variable
private:
    int days_to_maturity; //Days until the CD matures
};
```

Objects of CDAccount class type support all the member variables and functions in SavingsAccount and BankingAccount.

Deriving from a Derived Class



```
//Create a new CD with $1000, 6% interest, 180 days to maturity
CDAccount newCD(1000, 0, 6.0, 180);

newCD.deposit(100,50);
days_to_maturity = newCD.get_days_to_maturity( );
//Returns 180
balance = newCD.get_balance( );
//Returns 1100.50
```

Invoking a function in SavingsAccount
Invoking a function in CDAccount
Invoking a function in BankAccount

Objects of CDAccount class type support all the member variables and functions in SavingsAccount and BankingAccount as shown by this example.

Review Questions for Slide Set 2

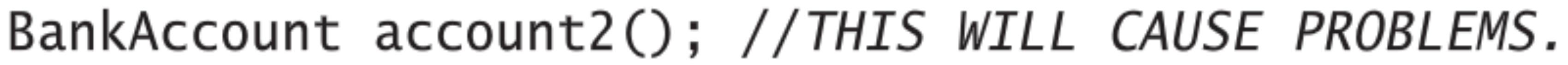
- What is the name of the member function used to initialize an object when it is created?
- Are Constructors class member functions?
- What are the constraints for Constructors relative to other member functions?
- Why do you make Constructors public member functions?
- Why would you want to overload your Constructor functions?
- With the option of Constructors, why might you still want/need Mutator Functions?

Review Questions for Slide Set 2

- What is a default Constructor? When do you need to create one?
- How do you explicitly call a Constructor inside of a function?
- Do Constructors with Initialization sections need to have empty function bodies?
- Do initialization section values need to be constants or can they be function parameters (e.g. constructor function parameters).
- Why is it important to be able to explicitly call a Constructor?

Review Questions for Slide Set 2

- What type of constructor is called by default with the new function when you instantiate a new object? What are the exceptions?
- Assuming you have a class BankAccount and want to instantiate an object account2, what is the problem with the following:



```
BankAccount account2(); //THIS WILL CAUSE PROBLEMS.
```

- What is Constructor Delegation and what are its limitations?

Review Questions for Slide Set 2

- What is the difference between data types and abstract data types
- Are all programmer defined data types abstract data types?
- Are int and float abstract data types?
- What are the general rules for making class definitions ADTs?
- What is the interface of an ADT (i.e. what defines it)?
- What defines the implementation of an ADT?
- How do you “harden” the separation of interface and implementation for ADTs?

Review Questions for Slide Set 2

- What is a derived class?
- Is the derived class the child or parent class?
- Is the base class the child or parent class?
- What is inheritance?