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: 1

Date: September 9, 2015

What we’re learning in this Slide Set:

- Quick Overview: how a computer operates
- Quick Overview: Programming Paradigms (Models)
- Creating Custom Data Types:
  - Remembering Structures & struct
  - Remember Scope
  - Unions
  - Creating Classes

Textbook Chapters:

Relevant to this slide set:

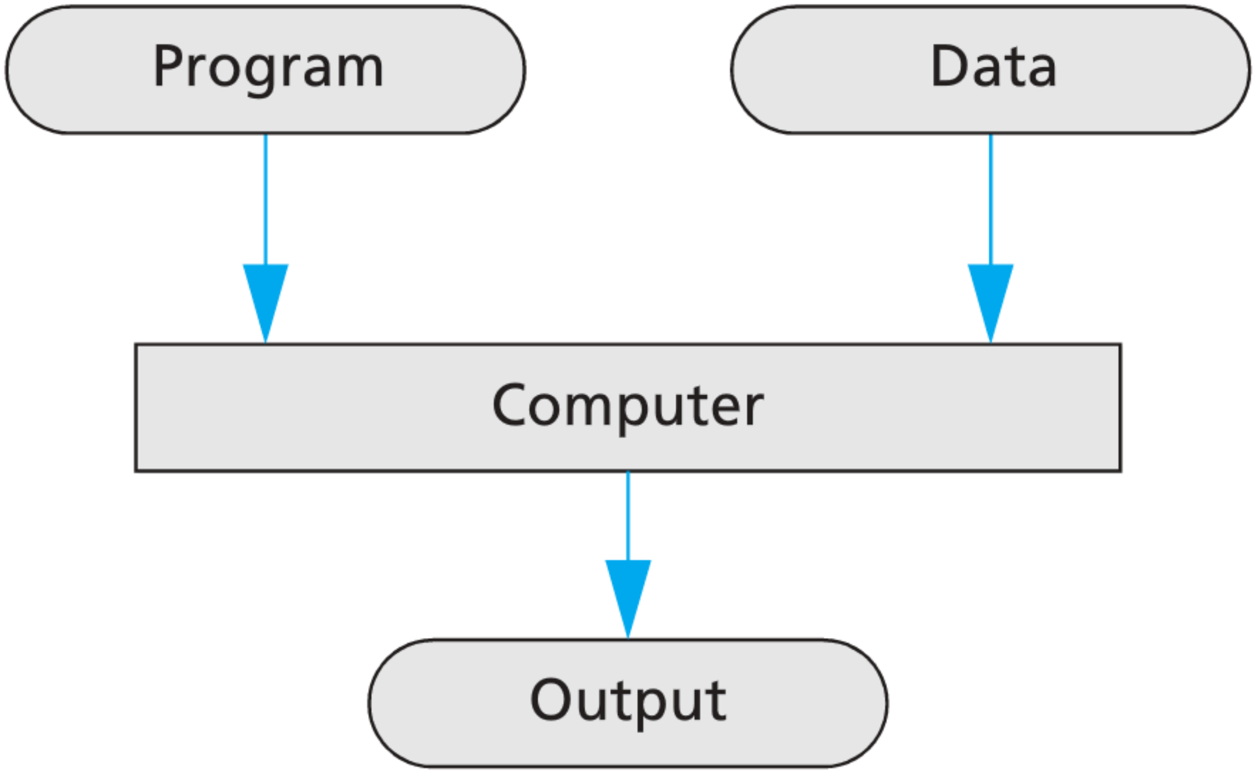
- Section 4.5
- Sections 10.1-10.2

Relevant to the next slide set:

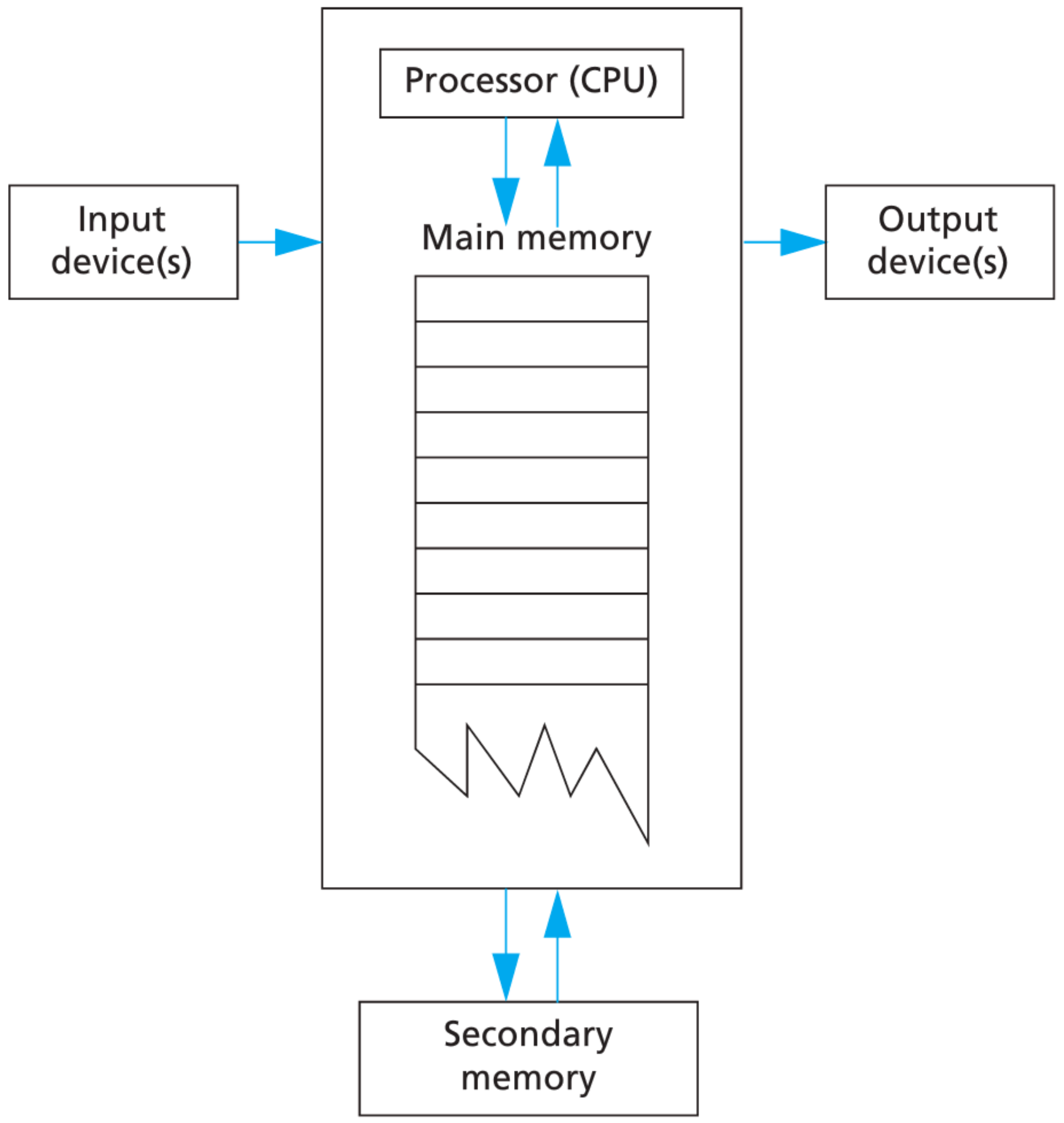
- Sections 10.3-10.4
- Chapter 11

How a Computer Operates

A simple view of computer processing:



Simple view of a computing system



What are some realistic constraints?

Quick Review: Programming Paradigm

Last semester in CMPT 128, you focused on the fundamental syntax of C++ and C and used a ***imperative programming paradigm (model)***

Imperative Programming Paradigm

Key characteristic: expresses a program as a *sequence of commands/actions*

Contrast: *declarative programming*- focuses on what a program should achieve but not how it is accomplished

Imperative Programming Paradigm

Other characteristics/things to keep in mind:

- You do this all the time when you create a “To Do” list to accomplish a task/set of tasks
- Tends to be the easiest model to learn as people tend to think this way intuitively
- Easiest way to create an algorithm to solve a problem

Object-Oriented Programming Paradigm

This semester we will focus on Object-oriented programming

- Objects have both
  - state
    - Implemented/stored as data fields/data members/member variables
  - behaviour
    - Implemented as member functions/methods
- Objects are instances of classes (“types+”)
  - Instances are specific realizations of classes or “Instantiations”-> this term will come up in ENSC 252

Object-Oriented Programming Paradigm

Key characteristics of Object-oriented programming:

- Encapsulation:
  - Provides information hiding and abstraction
  - Results in modular objects (and code) as all related data and functions should be packed (encapsulated) into single components (classes)
    - In other words each object is responsible for its own data and behaviour

Object-Oriented Programming Paradigm

Key characteristics of Object-oriented programming:

- Inheritance:
  - Code should be reusable
  - Can create new classes that extend previously existing classes, inheriting at least some of their behaviours (e.g. member functions)

- Polymorphism:
  - A single name may have multiple meanings (in the context of inheritance)
  - Provides a single interface to objects of different types

Just because you are using an Object-oriented programming language does ***<u>NOT</u>*** mean that your code is written using an Object-Oriented Programming Paradigm

In fact …

Many (Most) people who program using object-oriented programming languages, still write programs using an imperative programming paradigm.

One of the objectives of this semester is to get you comfortable with the fundamentals of object-oriented programming …

As opposed to just the syntax of C++

Classes

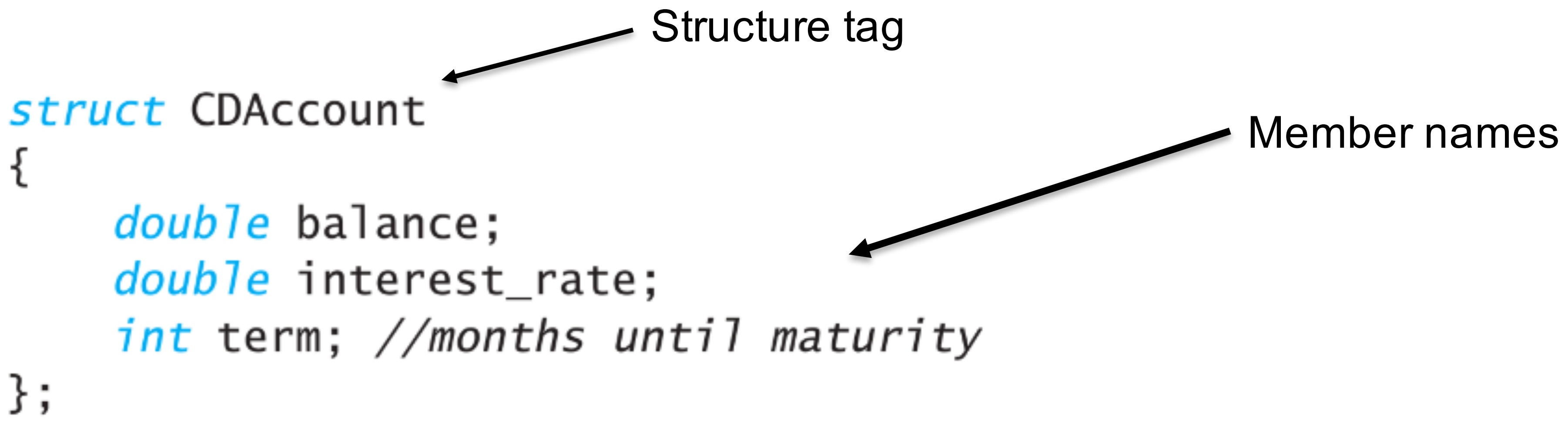
To create object-oriented programs, you need to know how to create classes.

Before we learn how to create classes, let me remind you about what you learned about structures.

Structures

You have already learned this, but a good review.

A structure is a type that contains multiple fields:



```
struct CDAccount
{
    double balance;
    double interest_rate;
    int term; //months until maturity
};
```

Structures

You can declare a variable who's type is your new structure:

```
struct CDAccount
{
    double balance;
    double interest_rate;
    int term; //months until maturity
};
```

CDAccount account, my_account;

Structures

To access members (aka member fields/member variables) of your structure:

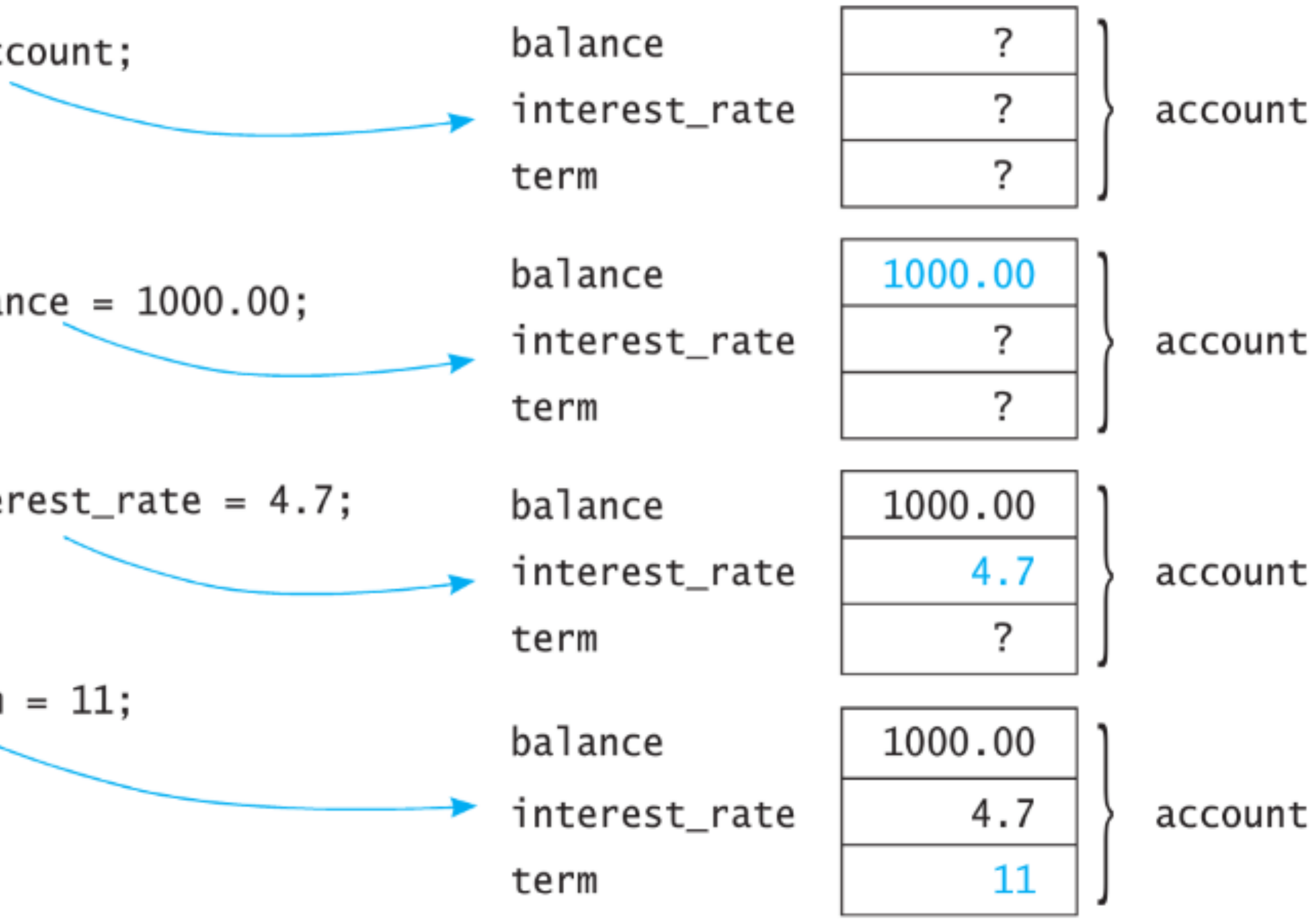
```
struct CDAccount
{
    double balance;
    double interest_rate;
    int term; //months until maturity
};
```

CDAccount account, my_account;

```
account.balance = 1000.00;
account.interest_rate = 4.7;
account.term = 11;
```

```
struct CDAccount
{
    double balance;
    double interest_rate;
    int term; //months until maturity
};
int main( )
{
  CDAccount account;
     ...

  account.balance = 1000.00;

  account.interest_rate = 4.7;

  account.term = 11;
```

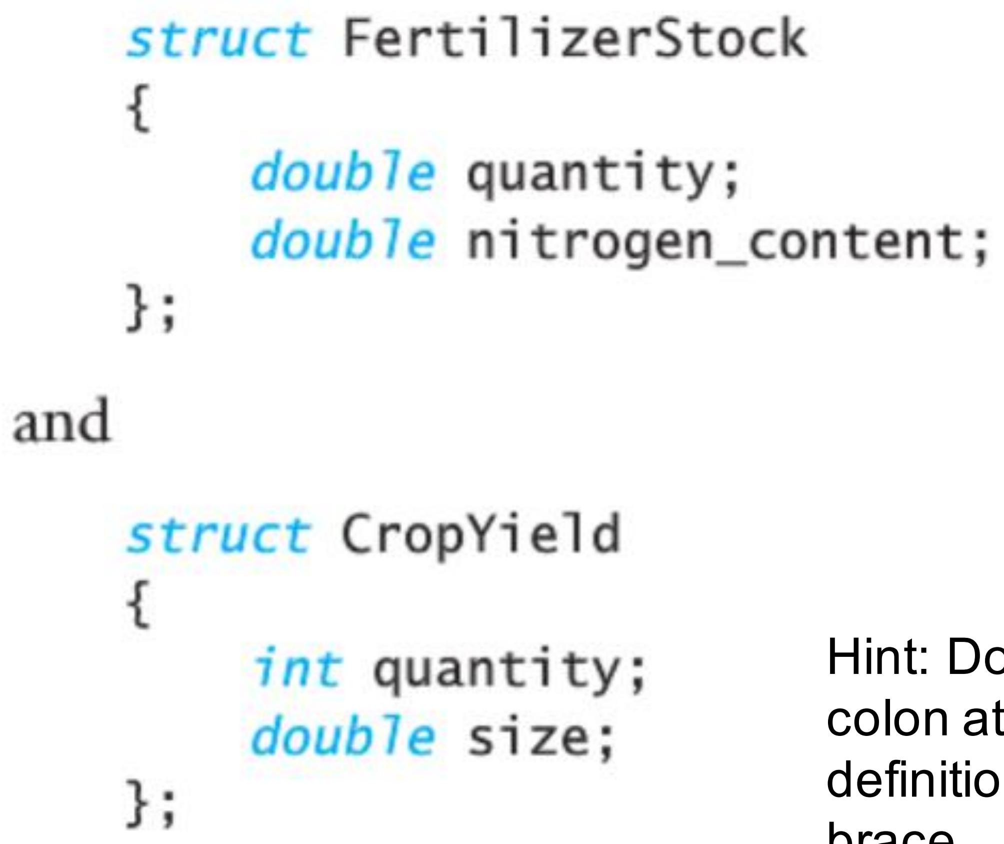
A structure's *value* is the collection of the individual member variables values.



```
//Program to demonstrate the CDAccount structure type.
#include <iostream>
using namespace std;
//Structure for a bank certificate of deposit:
struct CDAccount
{
    double balance;
    double interest_rate;
    int term; //months until maturity
};


void get_data(CDAccount& the_account);
//Postcondition: the_account.balance and the_account.interest_rate
//have been given values that the user entered at the keyboard.


int main( )
{
    CDAccount account;
    get_data(account);

    double rate_fraction, interest;
    rate_fraction = account.interest_rate / 100.0;
    interest = account.balance * rate_fraction * (account.term / 12.0);
    account.balance = account.balance + interest;
    cout.setf(ios::fixed);
    cout.setf(ios::showpoint);
    cout.precision(2);
    cout << "When your CD matures in "
         << account.term << " months,\n"
         << "it will have a balance of $"
         << account.balance << endl;
    return 0;
}

//Uses iostream:
void get_data(CDAccount& the_account)
{
    cout << "Enter account balance: $";
    cin >> the_account.balance;
    cout << "Enter account interest rate: ";
    cin >> the_account.interest_rate;
    cout << "Enter the number of months until maturity\n"
         << "(must be 12 or fewer months): ";
    cin >> the_account.term;
}
```

Structures:

Aside: It is ok to have multiple structures with the same member name:



```
struct FertilizerStock
{
      double quantity;
      double nitrogen_content;
};
```

and

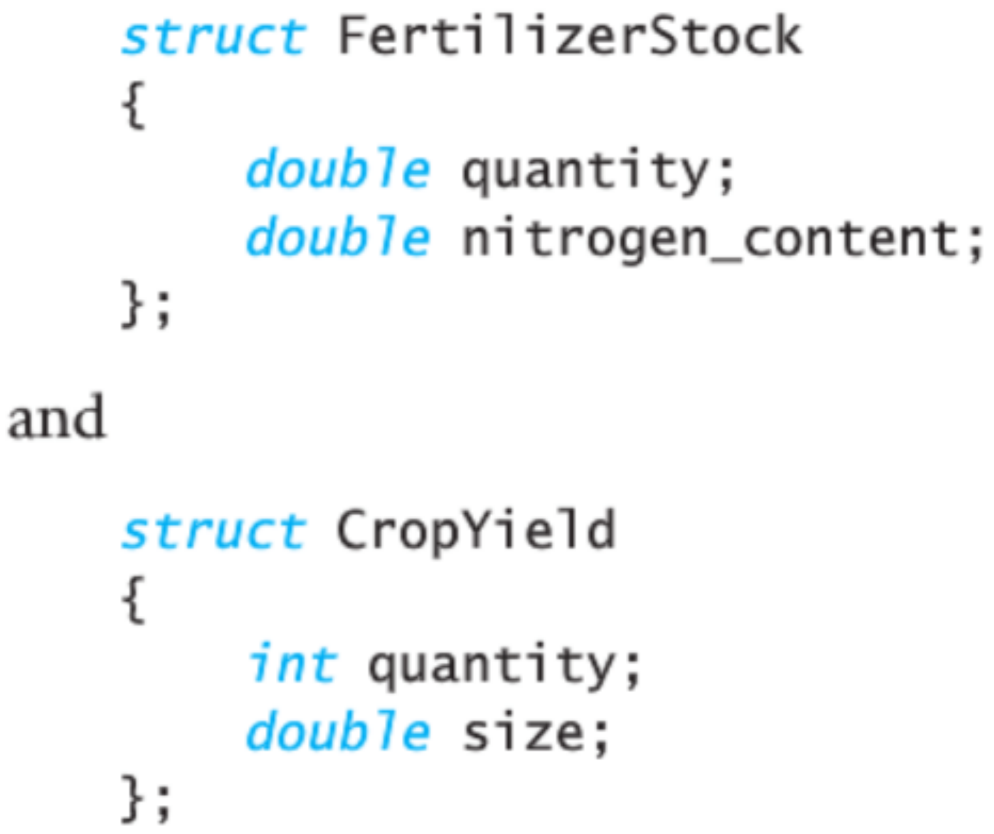
```
struct CropYield
{
      int quantity;
      double size;
};
```

Hint: Don’t forget the semi-colon at the end of structure definitions after the closing brace.

WHY????

Recall Scope

What would the variable declarations look like for these types?



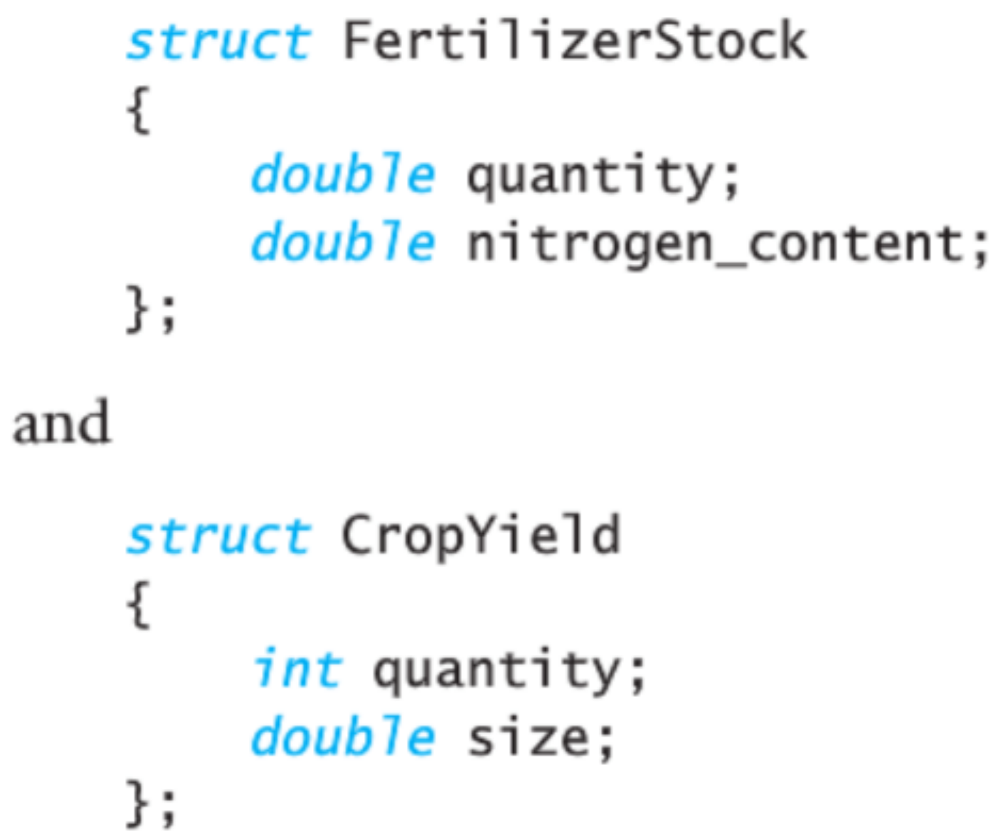
```
struct FertilizerStock
{
    double quantity;
    double nitrogen_content;
};
```

and

```
struct CropYield
{
    int quantity;
    double size;
};
```

Recall Scope

How would I indicate the `quantity` member variable for each?

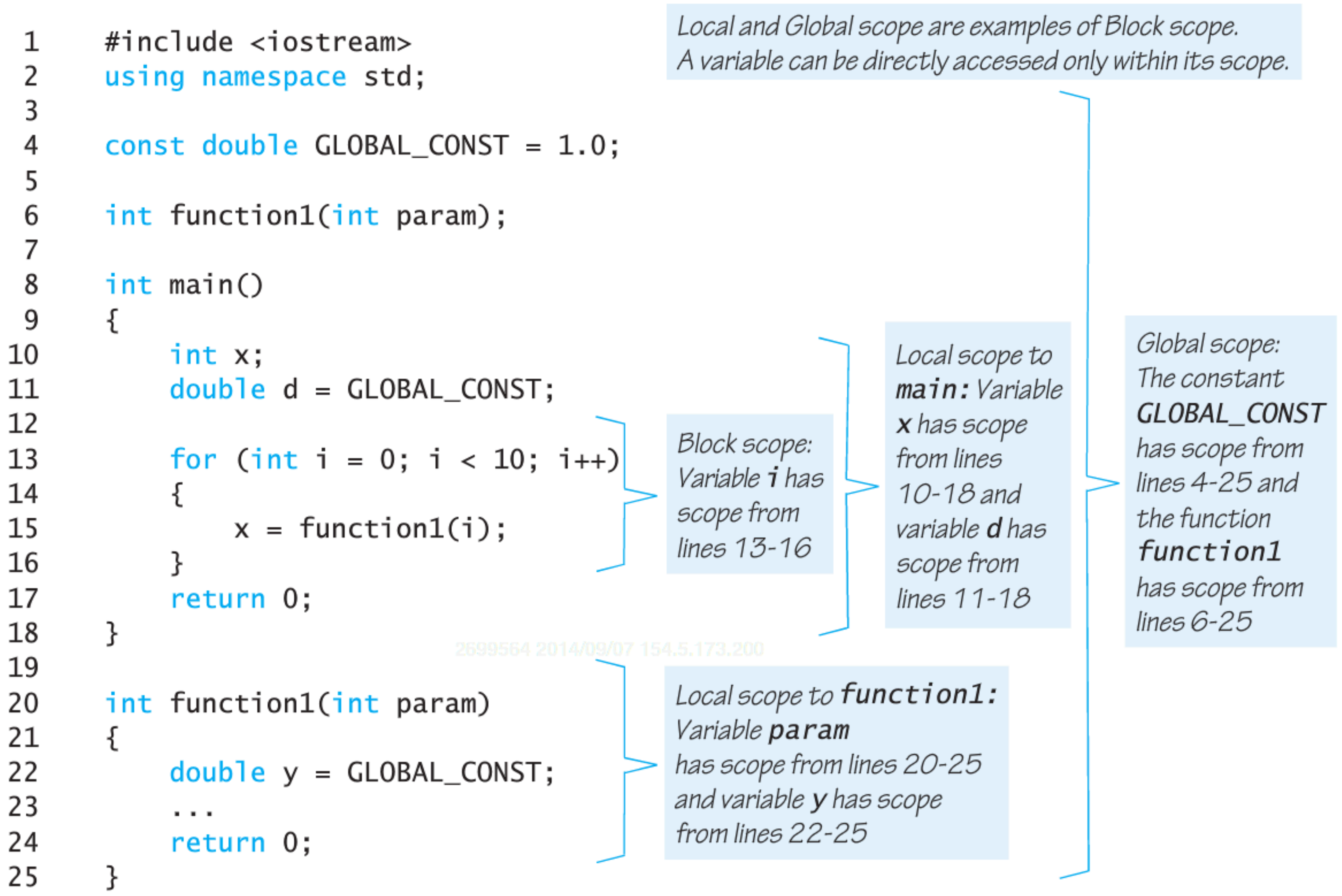


```
struct FertilizerStock
{
    double quantity;
    double nitrogen_content;
};
```

and

```
struct CropYield
{
    int quantity;
    double size;
};
```

Recall Local, Global, and Block Scope



```
 1   #include <iostream>
 2   using namespace std;
 3
 4   const double GLOBAL_CONST = 1.0;
 5
 6   int function1(int param);
 7
 8   int main()
 9   {
10       int x;
11       double d = GLOBAL_CONST;
12
13       for (int i = 0; i < 10; i++)
14       {
15           x = function1(i);
16       }
17       return 0;
18   }
19
20   int function1(int param)
21   {
22       double y = GLOBAL_CONST;
23       ...
24       return 0;
25   }
```

Local and Global scope are examples of Block scope.
A variable can be directly accessed only within its scope.

*Block scope: Variable **i** has scope from lines 13-16*

*Local scope to **main**: Variable **x** has scope from lines 10-18 and variable **d** has scope from lines 11-18*

*Global scope: The constant **GLOBAL_CONST** has scope from lines 4-25 and the function **function1** has scope from lines 6-25*

*Local scope to **function1**: Variable **param** has scope from lines 20-25 and variable **y** has scope from lines 22-25*

2699564 2014/09/07 154.5.173.200

Operators used to indicate scope

Along with what we just looked at, the '.' ("dot" operator) also indicates scope:

- Member variables within a structure
- Member variables within a class

Similarly, the ***scope resolution operator*** "::" is also used to indicate scope:

-Member Functions in a class

-Flags in a Class (recall the iostream flags "`ios::fixed`," etc.)

Final notes for Structures:

You can return a structure from a function:

```
CDAccount shrink_wrap(double the_balance,
                      double the_rate, int the_term)
{
    CDAccount temp;
    temp.balance = the_balance;
    temp.interest_rate = the_rate;
    temp.term = the_term;
    return temp;
}


CDAccount new_account;
new_account = shrink_wrap(10000.00, 5.1, 11);
```

Final Notes for Structures:

You can declare structures within structures, creating Hierarchical Structures:

```
struct Date
{
    int month;
    int day;
    int year;
};

struct PersonInfo
{
    double height; //in inches
    int weight; //in pounds
    Date birthday;
};
```

```
cout << person1.birthday.year;
```

Unions- The Structure’s First Cousins

Unions look almost exactly the same as structures.

Their descriptions (and variable declarations) are similar as a union also contains multiple fields:

union

```
~~struct~~ CDAccount
{
    double balance;
    double interest_rate;
    int term; //months until maturity
};
```

CDAccount account, my_account;

Unions- The Structure’s First Cousins

The key difference between a Union and a structure is:

- a Structure contains all member fields *concurrently*
- a Union is only able to store *one of the fields at a time* (and is only allocated sufficient memory to store the largest field)

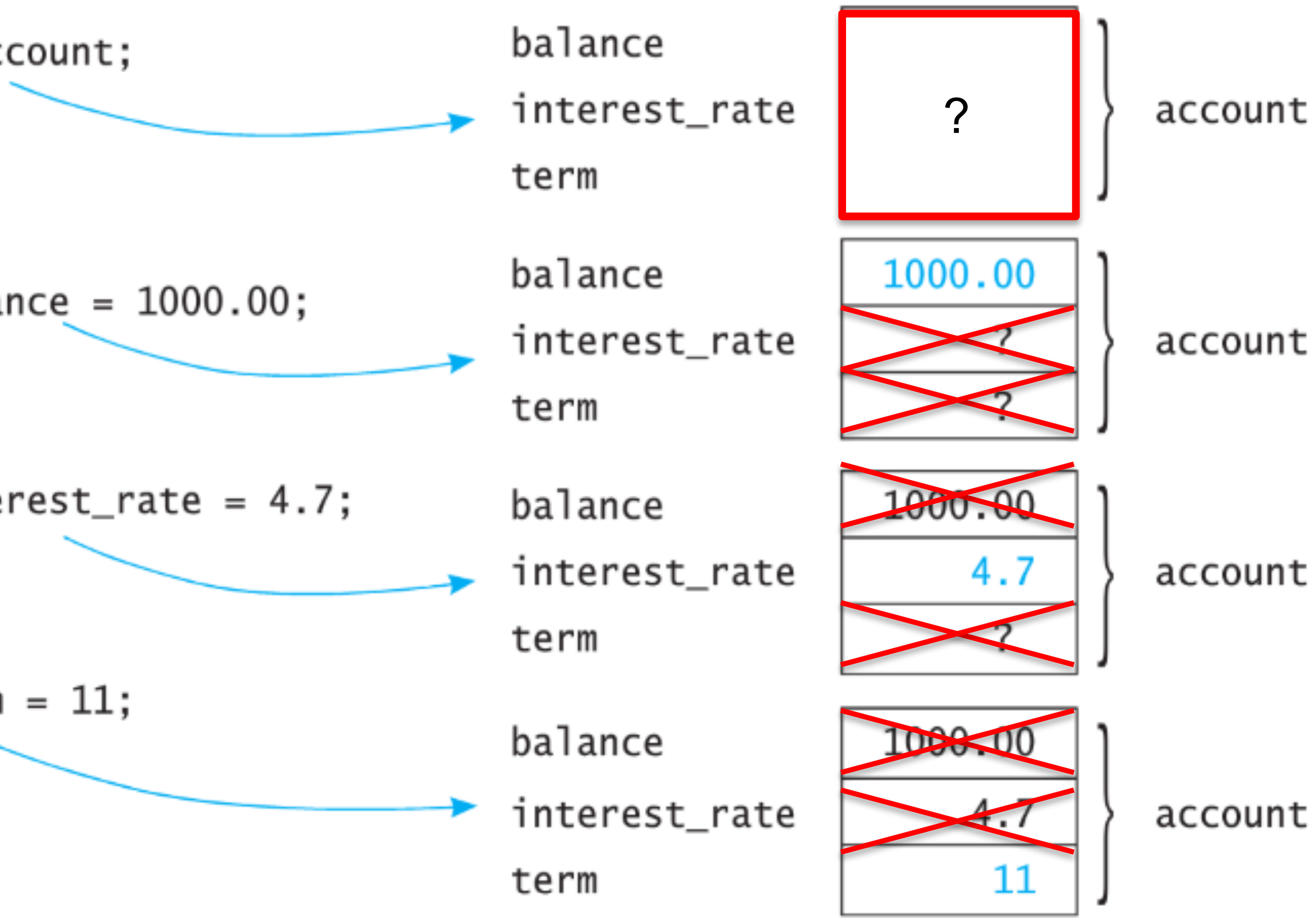
```
union
~~struct~~ CDAccount
{
    double balance;
    double interest_rate;
    int term; //months until maturity
};
```

CDAccount account, my_account;

```
account.balance = 1000.00;          OR
account.interest_rate = 4.7;        OR
account.term = 11;
```

```
union
~~struct~~ CDAccount
{
    double balance;
    double interest_rate;
    int term; //months until maturity
};
int main( )
{
   CDAccount account;
       ...

   account.balance = 1000.00;

   account.interest_rate = 4.7;

   account.term = 11;
```

Only one field's value is stored at any given time; whatever was previously there is over written



Structures are quite a common compound data type.

Unions are a bit more specialized, but have common applications in things like device drivers, which are commonly written by engineers (and important in ENSC 351)

Structures (and Unions) provide a convenient mechanism for organizing large amounts of data.

Object-Oriented languages go farther, and generalize the concept of structures -> Classes

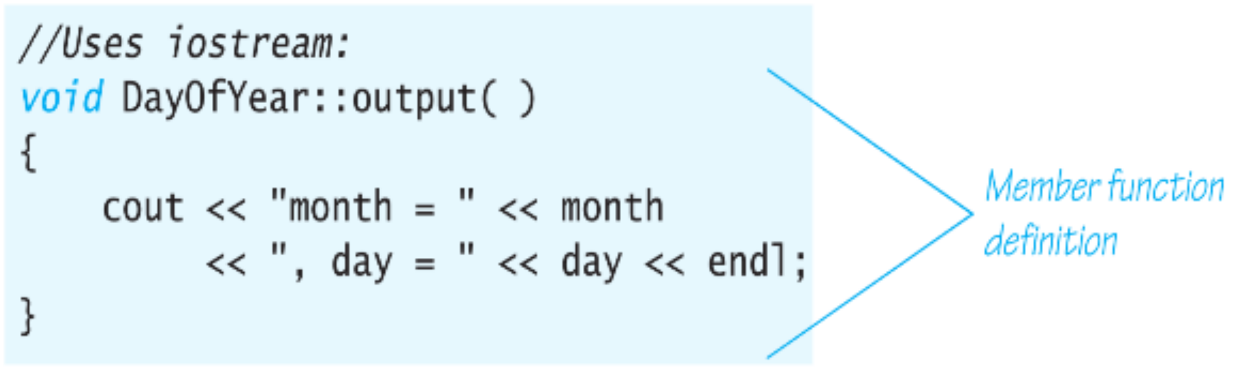
A class is like a structure, but it can contain functions too.

```
class DayOfYear
{
public:
    void output( );
    int month;
    int day;
};

int main( )
{
    DayOfYear today, birthday;

    cout << "Enter today's date:\n";
    cout << "Enter month as a number: ";
    cin >> today.month;
    cout << "Enter the day of the month: ";
    cin >> today.day;
    cout << "Enter your birthday:\n";
    cout << "Enter month as a number: ";
    cin >> birthday.month;
    cout << "Enter the day of the month: ";
    cin >> birthday.day;

    cout << "Today's date is ";
    today.output( );
    cout << "Your birthday is ";
    birthday.output( );
```

Member function declaration

Calls to the member function **output**

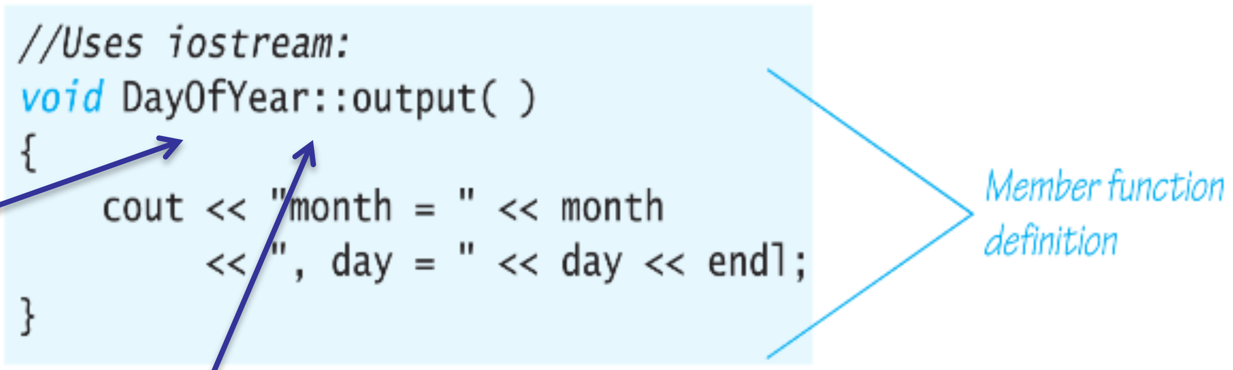
Defining the member function:



```
//Uses iostream:
void DayOfYear::output( )
{
    cout << "month = " << month
         << ", day = " << day << endl;
}
```

Member function definition

Defining the member function:



```
//Uses iostream:
void DayOfYear::output( )
{
    cout << "month = " << month
         << ", day = " << day << endl;
}
```

"Type qualifier"

+

"scope resolution operator"

Indicates what class this function is part of.

- You might have several classes, each with a member function called output()
- Remember our discussion of scope

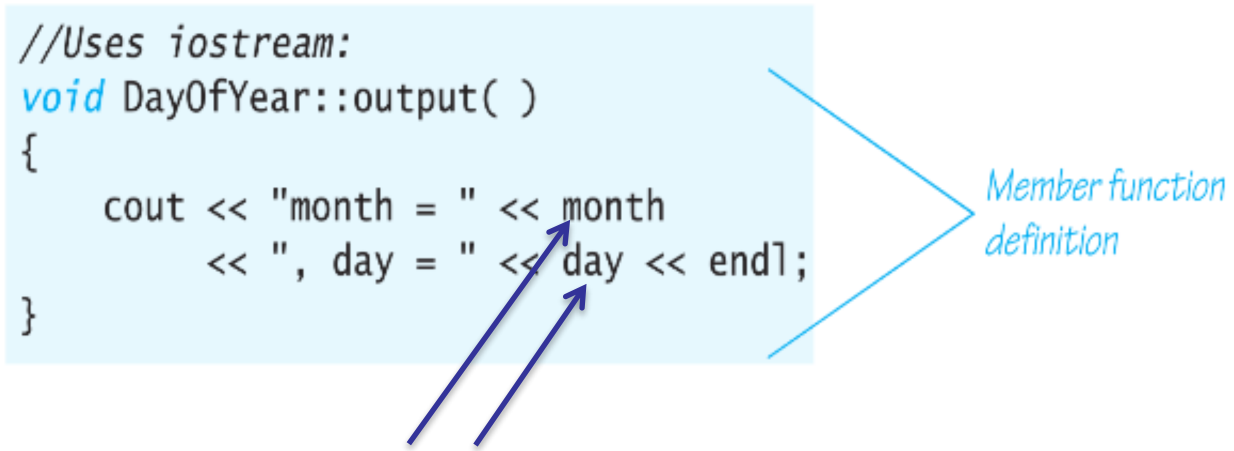
Important distinction:

DayOfYear is a class.

today and birthday are objects.

Remember objects are specific instances of a class.

When you define DayOfYear::output() you are defining a function that is associated with the class.

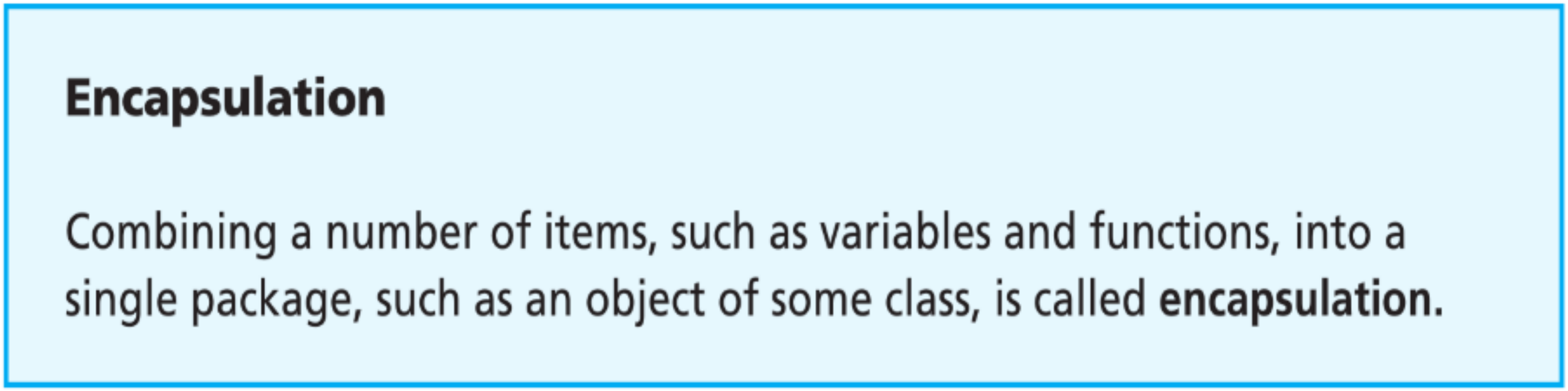


```
//Uses iostream:
void DayOfYear::output( )
{
    cout << "month = " << month
         << ", day = " << day << endl;
}
```

Refers to month and day fields inside this object

If you call today.output() then the output function prints the month and day fields within the today object.

If you call birthday.output() then the output function prints the month and day fields within the birthday object.



Encapsulation

Combining a number of items, such as variables and functions, into a single package, such as an object of some class, is called **encapsulation.**

For a given object you can:

1. Call a member function of that object
   eg. today.output()

2. Access fields inside the object directly
   eg. today.month

#2 is bad. Can you suggest why?

Accessing an object's data from outside the object is bad.

- If the implementation of the object changes, all the code that uses that object might have to change.
- Remember, technically the object is the ***value*** of the variable and not the variable itself

Your object will be much more re-usable if there is a clearly defined set of access methods for the object.

This provides a defined interface for an object (like an API).

This will also lead to code that is less buggy.

In an object-oriented language, the ideal is:

Given an object, you should never access the internal data directly. Only interact with the data inside an object through the member functions.

This is an ideal, and there are important exceptions.

Private vs. Public

Data and member functions in a class can be defined as either public or private:

Public: anyone can access this from outside the object
(normally only member functions)

Private: no one outside can access this field
(normally data fields and some functions)

Think of the public members as the API for your object

Private vs. Public

Ideally, you should separate:

- -the rules for using the class (Public) from
- -how the class computation is implemented/performed (Private)

Realistically, the rules and interface for using a class should not change. However, if the implementation of the class computation changes, this should not impact the rest of your software.

we gave in Display 10.3.

```
class DayOfYear
{
public:
    void input( );
    void output( );

    void set(int new_month, int new_day);
    //Precondition: new_month and new_day form a possible date.
    //Postcondition: The date is reset according to the arguments.

    int get_month( );
    //Returns the month, 1 for January, 2 for February, etc.

    int get_day( );
    //Returns the day of the month.
private:
    void check_date( );
    int month;
    int day;
};
```

Private member function

Private member variables

Any function/variable declared after the `public` label is public.

Any function/variable declared after the `private` label is private.

By default, if you don't put a label before the first members everything is private until otherwise specified:

- But that isn't good style and if everything is private, it isn't very useful (Why is that?)

```
class SampleClass
{
public:
    void do_something();
    int stuff;
private:
    void do_something_else();
    char more_stuff;
public:
    double do_yet_another_thing();
    double even_more_stuff;
};
```

You can interleave public and private function and variable declarations by adding a new label as appropriate, but this is confusing and not very readable – not good style.

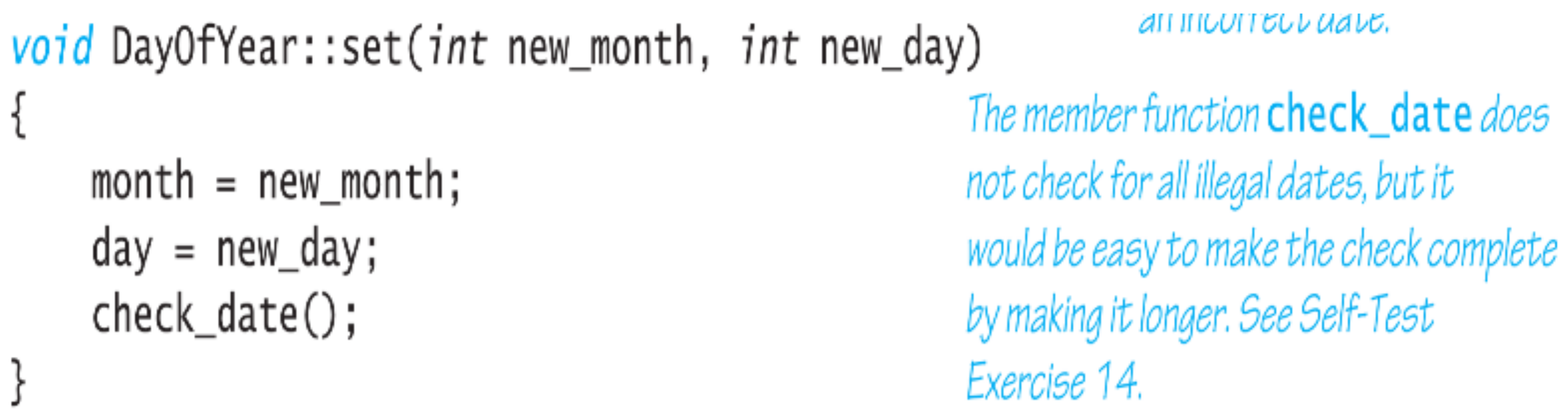
From outside the object:

```
DayOfYear today; //This line is OK.
today.month = 12; //ILLEGAL
today.day = 25; //ILLEGAL
```

because month and day are declared as private.

So what if you *do* want to set the month or day???

Luckily, this object provides a public member function to set the month and day.



```
void DayOfYear::set(int new_month, int new_day)
{
    month = new_month;
    day = new_day;
    check_date();
}
```

an incorrect date.

The member function **check_date** *does not check for all illegal dates, but it would be easy to make the check complete by making it longer. See Self-Test Exercise 14.*

If you always set the day and month using this function, your code will not depend on the internal representation of month and day.

How would you do this?

Don’t be confused: a member function that is part of an object can always access data within that object, whether it is private or public.

The private and public distinction only matters from outside the object.

Since your member variables are now private, this would also be illegal from outside the object:

```
cout << today.month; //ILLEGAL
cout << today.day; //ILLEGAL
if (today.month == 1) //ILLEGAL
    cout << "January";
```

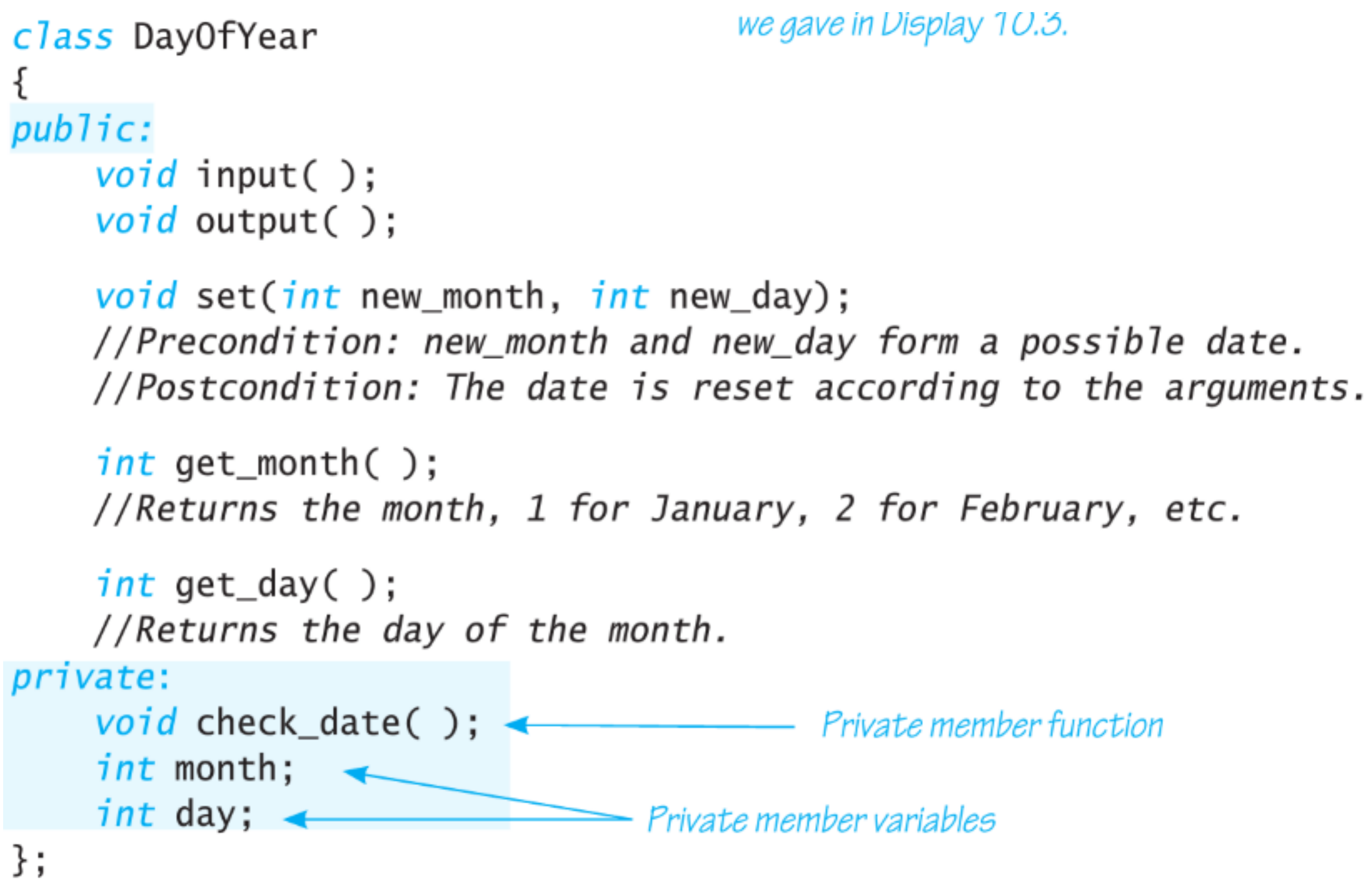
Since your member variables are now private, this would also be illegal from outside the object:

```
cout << today.month; //ILLEGAL
cout << today.day; //ILLEGAL
if (today.month == 1) //ILLEGAL
    cout << "January";
```

But you can get the data through public functions:

```
int DayOfYear::get_month( )
{
    return month;
}

int DayOfYear::get_day( )
{
    return day;
}
```

Recall:



we gave in Display 10.3.

```
class DayOfYear
{
public:
    void input( );
    void output( );

    void set(int new_month, int new_day);
    //Precondition: new_month and new_day form a possible date.
    //Postcondition: The date is reset according to the arguments.

    int get_month( );
    //Returns the month, 1 for January, 2 for February, etc.

    int get_day( );
    //Returns the day of the month.
private:
    void check_date( );
    int month;
    int day;
};
```

```
int main( )
{
    DayOfYear today, bach_birthday;
    cout << "Enter today's date:\n";
    today.input( );
    cout << "Today's date is ";
    today.output( );

    bach_birthday.set(3, 21);
    cout << "J. S. Bach's birthday is ";
    bach_birthday.output( );

    if (today.get_month( ) == bach_birthday.get_month( ) &&
        today.get_day( ) == bach_birthday.get_day( ) )
        cout << "Happy Birthday Johann Sebastian!\n";
    else
        cout << "Happy Unbirthday Johann Sebastian!\n";
    return 0;
}
```

Here’s the main function

Here's the DayOfYear member functions

```
//Uses iostream:
void DayOfYear::input( )
{
    cout << "Enter the month as a number: ";
    cin >> month;
    cout << "Enter the day of the month: ";
    cin >> day;
    check_date( );
}

void DayOfYear::output( )
 <The rest of the definition of DayOfYear::output is
  given in Display 10.3.>

void DayOfYear::set(int new_month, int new_day)
{
    month = new_month;
    day = new_day;
    check_date();
}
```

Private members may be used in member function definitions (but not elsewhere).

A better definition of the member function **`input`** *would ask the user to reenter the date if the user enters an incorrect date.*

The member function **`check_date`** *does not check for all illegal dates, but it would be easy to make the check complete by making it longer. See Self-Test Exercise 14.*

Here’s the DayOfYear member functions

```
void DayOfYear::check_date( )
{
    if ((month < 1) || (month > 12) || (day < 1) || (day > 31))
    {
        cout << "Illegal date. Aborting program.\n";
        exit(1);
    }
}

int DayOfYear::get_month( )
{
    return month;
}

int DayOfYear::get_day( )
{
    return day;
}
```

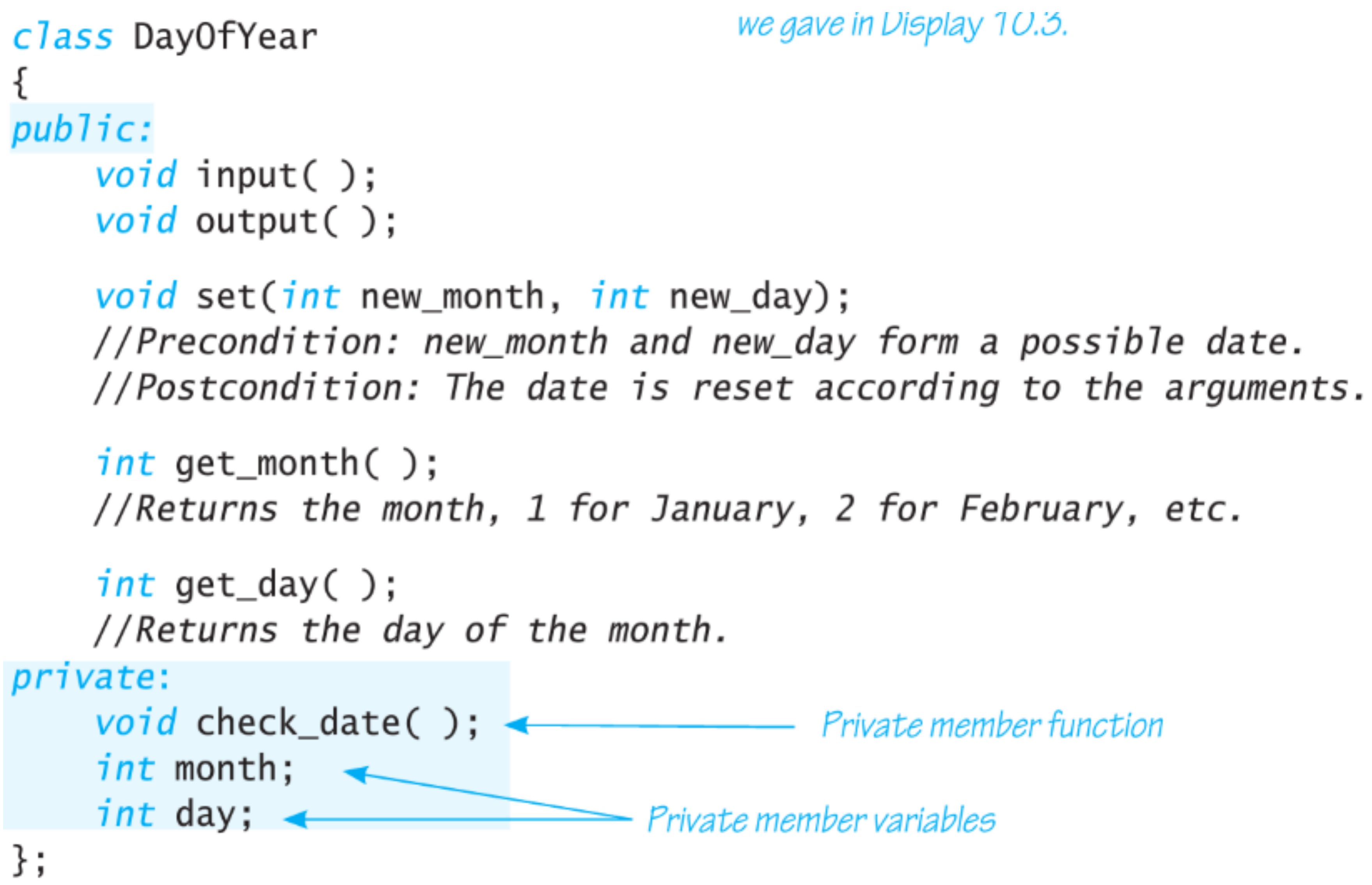
The function **`exit`** *is discussed in Chapter 6. It ends the program.*

With the `check_date` Function, this code is a bit more robust.

However, something is missing.

What do we need to fix to make it more robust?

Again Recall:


we gave in Display 10.3.
Private member function
Private member variables

```
class DayOfYear
{
public:
    void input( );
    void output( );

    void set(int new_month, int new_day);
    //Precondition: new_month and new_day form a possible date.
    //Postcondition: The date is reset according to the arguments.

    int get_month( );
    //Returns the month, 1 for January, 2 for February, etc.

    int get_day( );
    //Returns the day of the month.
private:
    void check_date( );
    int month;
    int day;
};
```

What if we change the private data members:

```
class DayOfYear
{
public:
    void input();
    void output();

    void set(int new_month, int new_day);
    //Precondition: new_month and new_day form a possible date.
    //Postcondition: The date is reset according to the
    //arguments.

    int get_month();
    //Returns the month, 1 for January, 2 for February, etc.

    int get_day();
    //Returns the day of the month.
private:
    void DayOfYear::check_date( );
    char first_letter; //of month
    char second_letter; //of month
    char third_letter; //of month
    int day;
};
```

Is our programming interface (the public functions) still usable?

If yes, why?

If not, can we do anything about it?

Comparing objects of the same type

Ideally, you want to be able to compare to objects of the same type to see if they are equal.

For user defined classes, this is not inherently possible.

You will need to create a public member function to support this:

-Soon we will look at how you can overload the “==“ operator to do this

The assignment operator

The assignment operator, ‘=‘ is legal so you can simply assign one object to another:

```
DayOfYear       due_date, tomorrow;
due_date = tomorrow;
```

This will assign all data field in one object (both public and private) to the other.

- There are some situations where we will want to redefine the assignment operator (overload it).

*We’ll talk about this later, but the following is an example for now

```
//Class for a bank account:
class BankAccount
{
public:
    void set(int dollars, int cents, double rate);
    //Postcondition: The account balance has been set to $dollars.cents;
    //The interest rate has been set to rate percent.

    void set(int dollars, double rate);
    //Postcondition: The account balance has been set to $dollars.00.
    //The interest rate has been set to rate percent.

    void update( );
    //Postcondition: One year of simple interest has been
    //added to the account balance.

    double get_balance( );
    //Returns the current account balance.

    double get_rate( );
    //Returns the current account interest rate as a percentage.

    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then
    //outs has already been connected to a file.
    //Postcondition: Account balance and interest rate have
    //been written to the stream outs.
private:
    double balance;
    double interest_rate;

    double fraction(double percent);
    //Converts a percentage to a fraction. For example, fraction(50.3)
    //returns 0.503.
};
```

The member function set is overloaded.

```
int main( )
{
    BankAccount account1, account2;
    cout << "Start of Test:\n";
    account1.set(123, 99, 3.0);
    cout << "account1 initial statement:\n";
    account1.output(cout);
    account1.set(100, 5.0);
    cout << "account1 with new setup:\n";
    account1.output(cout);

    account1.update( );
    cout << "account1 after update:\n";
    account1.output(cout);

    account2 = account1;
    cout << "account2:\n";
    account2.output(cout);
    return 0;
}
```

Calls to the overloaded member function set

2699564 2014/09/07 154.5.173.200

```
void BankAccount::set(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;
}

void BankAccount::set(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;
}
```

Definitions of overloaded member function **set**

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

double BankAccount::fraction(double percent_value)
{
    return (percent_value / 100.0);
}

double BankAccount::get_balance( )
{
    return balance;
}

double BankAccount::get_rate( )
{
    return interest_rate;
}

//Uses iostream:
void BankAccount::output(ostream& outs)
{
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);
    outs << "Account balance $" << balance << endl;
    outs << "Interest rate " << interest_rate << "%" << endl;
}
```

In the definition of a member function, you call another member function like this.

Stream parameter that can be replaced either with cout or with a file output stream

Based on the previous code:

```
account1.update();
```

Becomes:

```
{
    account1.balance = account1.balance +
    account1.fraction(account1.interest_rate) * account1.balance;
}
```

So the call to the private function “`fraction`” is handled the same as an access to a member variable. From the object’s perspective they are both (basically) the same.

Other notes about the previous code:

`output` accesses a stream-> not the std output (cout)

```
//Uses iostream:
void BankAccount::output(ostream& outs)
{
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);
    outs << "Account balance $" << balance << endl;
    outs << "Interest rate " << interest_rate << "%" << endl;
}
```

Doing this makes your code much more reusable and easier to debug in batch mode

*****This will be important very soon and you should look at using this style of coding***

Other notes about previous code: the overloaded `set` function

When you overload a member function, it lets the user call it based on the available parameters

-Sometimes users may not have all of the parameters (e.g. cents in this case).

-It makes your code more reusable and more portable

-For example, can be extremely valuable for Mutator functions

```
void BankAccount::set(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;
}

void BankAccount::set(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;
}
```

Definitions of overloaded member function `set`

Other notes on Classes:

For now, you can use **Accessor Functions**, public methods that enable you to determine the values of private data fields.

-They provide an equivalent value instead of the actual value

-Commonly named “get” functions

Mutator Functions are public member functions allow you to alter the value of private data members.

-Commonly named “set” functions

These functions are important for every class definition so that you can assign values to an object.

Final Notes on Structures and Classes

Structures are set with all member variables as public

-Unless you explicitly state otherwise (`private`)

Structures in C++ can have member functions

-Don't do it; it will just be confusing-> use classes

Structures in C cannot have member functions:

-They can have pointers to functions (void *), but you really shouldn't do it

ESPECIALLY not until you know what you are doing.

Final Notes on Structures and Classes

In C++, Structures can do anything that Classes can do, but the notation is different.

So why confuse yourself with 2 sets of syntax? Use Structures to declare compound types and Classes to encapsulate variables and member functions.

This is how most people do it and will make your use of structures portable to C programming.

People don’t use unions in C++ (as far as I know)

Final Notes on Structures and Classes

Classes have member variables and methods (member functions)

Both data and methods can be public or private

-Don’t forget they are *private* by default

-Normally, all member variables are *private*

Private methods and variables can only be accessed within the class definition (in the definitions of its own member functions)

Final Notes on Structures and Classes

Member functions can be overloaded just like normal functions

A class may use another class for the type of a member variable

-This creates a hierarchy of classes similar to hierarchical structures

Final Notes on Structures and Classes

A function may have formal parameters whose types are classes:

```
double difference(BankAccount account1, BankAccount account2);
//Precondition: account1 and account2 have been given values
//(that is, their member variables have been given values).
//Returns the balance in account1 minus the balance in
account2.
```

A function may return an object (i.e. the return type of a function may be a class):

```
BankAccount new_account(BankAccount old_account);
//Precondition: old_account has previously been given a value
//(that is, its member variables have been given values).
//Returns the value for a new account that has a balance of zero
//and the same interest rate as the old_account.
```

Final Notes on Structures and Classes

In short, Classes can be viewed as compound types with additional member functions that let you access/manipulate their member variables through public methods

Review Questions for Slide Set 1

- What are the constraints on the performance of a computing system?
- What is the key characteristic of the imperative programming paradigm?
- What is the key characteristic of the declarative programming paradigm?
- Objects have two aspects- what are they?
- What is the difference between objects and classes?
- What are the three characteristics of object-oriented programming?
- Is it possible to write imperative styled programs with an object oriented programming language?

Review Questions for Slide Set 1

- What is a structure?
- What operator do you use to dereference a member field in a structure?
- Can multiple different structure types have the same member variable names?
- What is the difference between Block, Local and Global Scope?
- Name the two scope operators we looked at in this slide? What are they and what are they called? (slide 24)
- Can you declare a structure within a structure?

Review Questions for Slide Set 1

- True or False: Unions and Structures have exactly the same structure, excluding the change of keywords. (If not, how are they different?)
- True or False: Unions and Structures have exactly the same memory requirements. (If not, how are they different?)
- Are classes more like structures or unions?
- Can different classes have the same member functions?
- Classes are an example of which of the characteristics of object-oriented programming? Why?
- Why is it bad to access an object’s data members outside of its object?

Review Questions for Slide Set 1

- How should you access the values stored in an object’s member fields outside of the object?
- What is the difference between private and public class member fields?
- By default, are class members private or public?
- If all of a class’ members are private, what is the problem?
- Can public members of a class access that class’ private member fields?
- Can you inherently compare two objects of a user defined class?

Review Questions for Slide Set 1

- Can you inherently assign one object of a user defined class to another?
- Why might you want to overload a class member function?
- What are Mutator and Accessor Functions?
- Are Structures set with all member variables as private or public by default?
- Can Structures have member functions in both C and C++?
- What is the difference between Structures and Unions in C++?
- Can classes be hierarchical (i.e. can a class have another class as the type for a member variable)?

Review Questions for Slide Set 1

- Can class functions have formal parameters of that class type?
- Can functions return values of class type?
- True or False: Member functions can be overloaded just like normal functions?
- What happens if you declare a struct inside a function as opposed to outside of all of the function definitions?