Object Oriented Programming With C++

- C++ separates the class definition (.h file) from the class implementation (.cpp file)
- The structure of a class definition is as follows (using `account.h` as an example):

```
class account {
   public:
    // Members that are externally visible
    // These are member functions
    // Constructors
    account();
    account(int open_amt);
    // could use account(int open_amt = 0);

    // Members that mutate data
    void deposit(int amount);
    void withdraw(int amount);

    // Members that query data but do not change it
    int balance() const;
    bool has_funds() const;

   private:
    // Members that are encapsulated
    // These are member variables
    int acct_balance;
}
```

The Member List

- In the class body we only list member function *prototypes* and *not the implementation* code
  - Member functions that are not permitted to modify the member variables are designated `const`
- Items in the *public* section are visible and available to everyone who uses the data type
- The *private* section lists members that are not visible outside an instance of the class

Comments

- Notice that the *implementations* of the member functions have not been provided
  - This allows separation of the definition of the functions from their realisation in code
- Typically the `.h` file would include detailed comments on the operation of each member function or variable
  - Including pre- and post-conditions
  - This informs users of the class(es) about their behaviour while maintaining a 'black box' approach to how it is achieved.
- It is not necessary to provide formal parameter names in the prototypes
  - So `void deposit(int);` would suffice in the `.h` file

Implementation of Member Functions

- These are implemented in a separate file called `account.cpp` as follows:

```
#include "account.h"
// Constructors
account::account() {
    acct_balance = 0;
}
account::account(int open_amt) {
    acct_balance = open_amt;
}
void account::deposit(int amount) {
    acct_balance += amount;
}
// ... Other implementations ... ///
bool account::has_funds() const {
    return (acct_balance > 0); // a bit dodgy
   // another option would be
   // if (acct_balance > 0) {return true;}
   // else {return false;}
}
```

Comments

- Note that `account.h` is included so that the member function definitions are placed in the correct context
- Each member function definition starts with the name of the class to which it applies, followed by `::`
- This example provides two constructors, with and without creation parameters
  - In some cases this can be reduced to a single parameter using the definition of an initial value in the class definition

    `int acct_balance = 0;`
- Also note the use of `return` when the member function is not `void`

Creating & Using Instances

- An instance of class `account` can be created by simply using the declaration

```
account my_account; // no need for "new"
```

- The member functions for the new instance are used as follows:

```
if (my_account.has_funds())
   cout << "Balance is "
        << my_account.balance()
        << endl;
else
   cout << "Account is empty" << endl;
```

or

```
int gift = 3000;
my_account.deposit(gift);
```

Compiling

- Your `Makefile` should list all the `.cpp` files
  - The `.h` files are not listed in `Makefile` since they are referred to by your `#include` statements in the header of the .cpp files

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=bank.cpp account.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=test

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
	$(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:
	$(CC) $(CFLAGS) $< -o $@

clean:
	rm -rf *.o core
```

Memory Management

- A running computer program can create and manipulate data in two different locations
  - On the program *stack*
  - In the *heap* associated with the program
- In Java, all objects (i.e. non-primitive types) are stored in the heap
  - Thus all non-primitive data types are accessed using reference variables
  - The Java heap is automatically garbage collected by the JVM

Memory Management

- In C++, local variables *including objects* can be created on the stack or in the heap
  - The objects we have created to date have all been created on the stack
    - Indicated by the absence of `new` in their creation
  - Objects created on the stack can be semantically treated just like primitive types
    - So the use of = causes the *state* of objects to be copied (i.e. the LH object becomes a duplicate of the RH object)
      - Whereas in Java only the value of the reference variable is copied
    - When the context of the object ends, the space occupied by the object is automatically freed

Memory Management

Heap vs. Stack

- Objects can be created with commands of the following form:
  - On the stack:

```
account my_account(100);
```

  - In the heap:

```
account *my_account = new my_account(100);
```

- Objects created on the stack have an automatic garbage collection. Variables are deleted when they become out of scope. No memory leak.
- Objects created in the heap do NOT have automatic garbage collection. If pointer is not deleted, the object is still there, occupying memory space – memory leak.
- Why stack, why heap? Stack is small. For example, with MS Visual C++ 6.0 the default stack size is 1 MB. Heap is large, all RAM plus Hard Disks (Dynamic memory allocation)

Accessing Objects

- In C++ a programmer can access objects in varying ways:
  - A variable can be defined that refers to the object on the run-time stack, or
  - A *pointer variable* can be defined that stores the memory address of the object, or
  - A *reference variable* can be defined as an alias for an object
    - These are initialised when they are defined and cannot later be changed
    - An automatic and invisible de-reference occurs when a reference variable is used

Pointer Variables

- A C++ pointer variable stores *the memory address* of a primitive type or class instance
  - Though typically you would not use a pointer variable to access a primitive type
- The memory address of an entity is found using the *unary address-of operator* `&`
  - So `&obj` is the memory location that stores `obj`
- We can define a variable that points to an instance of class `obj_class` using the statement `obj_class *ptr;`
  - `*` is the *unary dereferencing operator*
  - `ptr` is the name of a value that stores an address and `ptr` IS NOT INITIALISED by this statement
  - To initialize it we use

```
ptr = new obj_class;
```

Pointer Variables

- If we created an object of type `obj_class` using the statement `obj_class obj;` then `ptr` could be made to point at `obj` using the statement `ptr = &obj;`
  - Or `obj_class* ptr = &obj;` would do it all in a single statement
- If `obj_class` has a member function with header `int balance();` then we could use the statement

```
int current = obj.balance();
```

- or alternatively

```
int current =(*ptr).balance();
```

Comments

- It is necessary to use the brackets in `int current = (*ptr).balance();` because of precedence rules
  - In their absence the compiler tries to dereference the outcome of `ptr.balance()`
- The use of the brackets is a nuisance, so the notation `->` can be used instead, so

```
int current = ptr->balance();
```

does the same thing!

Example

Reference Variables

- A *reference type* is a pointer constant that is always automatically dereferenced
- It is like an alias for another object, so:

```
account acc1;
account& acc2 = acc1;
acc2.deposit(30);
```

would result in:

  - Creation of an instance of class `account` called `acc1`
  - Creation of an alias to `acc1` called `acc2`
  - Addition of 30 to the balance of the `account` object
- It only makes good sense to use this feature when passing an object as a parameter to a function.
- In this case, you do not pass the entire object, but just a reference, which means less overhead when dealing with large objects.

Multiple Definitions of the Same Class

- Good Software Engineering involves *re-use* of software components
  - As we have seen in our example, the `account` class has been created so it can be re-used in some other application if required
- As a software project grows, it is possible that the same class may be included in different components
  - E.g. class `account` may be used in a Savings module and again in a Loan module, then those two modules may be used in an all-encompassing Banking module
  - In such a situation the compiler would fail with a *duplicate class definition* error
- This problem is avoided using the *macro guard* compiler directive structure

Macro Guard

- The macro guard compiler directive involves changing the `.h` file name that defines your class
  - Create a guard name
    - A good idea is to use your name plus the class name e.g. `ALEX_ACCOUNT`
  - Bracket the class definition with the guard statements:

```
#ifndef ALEX_ACCOUNT
#define ALEX_ACCOUNT

// The class definition goes here

#endif
```

- This causes the compiler to check when the class definition is included, and if `ALEX_ACCOUNT` has been previously defined (indicating that account has also been defined) then the compiler does not do the include again

Namespaces

- As more and more components are used to build a software system, the possibility of duplicate entity names increases
  - Examples of such names include variables and member functions
- This is avoided using a structure that defines the *scope of names* used in the code
  - This structure is called a *namespace*
  - To avoid duplicate namespaces, a good idea is to include the programmer's own name in the namespace name
    - E.g. `alex_project`

Use of Namespaces

- To restrict the scope of names used in class `account`:
  - Before the implementation code, start the namespace grouping using the keyword `namespace` followed by the name you decided upon
  - Use the curly brackets `{` and `}` to show the extent of the namespace
    - Do this for the `.h` and the `.cpp` files that define the class
  - Add another `using` statement to the bootstrap program file, specifying the namespace and the class to be used
    - E.g. `using alex_project::account;`

Example

```
#include <iostream>
#include <cstdlib>
#include "account.h"

using namespace std;
using alex_project::account;

int main() {
// Code goes here
    return EXIT_SUCCESS:
}
```

Main program .cpp

Header file .h

```
#ifndef ALEX_ACCOUNT_H
#define ALEX_ACCOUNT_H

namespace alex_project {
    class account {
    // Class documentation goes here, including pre
    // and post conditions and member definitions
    }
}
#endif
```

Class implementation .cpp

```
#include "account.h"

namespace alex_project {
// Member functions implementations
// go here
}
```

Default Arguments

- Default values can be defined for parameters of member functions by listing them in the member function prototypes (i.e. in the `.h` file)
  - Note that the default *should not be defined in the implementation.*
  - E.g. in the class definition of `account`, we used:

```
account(); // For when there are no arguments
account(int open_amt); // For when an initial
                       // value is provided
```

- This could have been replaced with a single statement:

```
account(int open_amt = 0);
```

- In this case the constructor code would store the value of `open_amt` in `acct_balance`
  - Using the value zero by default, or a caller-provided value if one has been provided
    - This covers both forms of initialization, say `account()` and `account(50);`

Multiple Parameters

- A function with multiple arguments does not need to have defaults defined for all of them
  - But if only some have defaults, they must be listed *last*
  - Parameters that are provided will fill the arguments from left to right
- E.g. if a day-from-date function named `ddate` is defined as:

```
#include<string>
…
string ddate(int yr, int mth = 1, int day = 1);
```

```
ddate(2006);
```
  - would use 1 for mth and 1 for day

```
ddate(2006,7);
```
  - would use 1 for day

```
ddate(2006,7,23);
```
  - would use provided parameters

Parameter Types

- Like Java, C++ uses *copies* of primitive parameters when called functions are executed. Thus:

```
void swap(int num1, int num2){
    int temp = num1;
    num1 = num2;
    num2 = temp;
}
```

when called using:

```
int x = 5;
int y = 7;
swap(x,y);
cout << x << " : "<< y << endl;
```

would output `5 : 7` i.e. the original parameter values have not been swapped

Objects as Parameters

- In Java, all non-primitive parameter calls result in changes to object parameters existing after method return
  - Because the copied parameters are *references* to the target object
    - So the target is accessed by the method
- In C++ this *is not* always the case
  - E.g. an *instance* of account would be copied for use by a called function if it had been created as `account acc;` and then `acc` was a parameter to a function call
  - To avoid that overhead, it is possible to define *reference parameters*

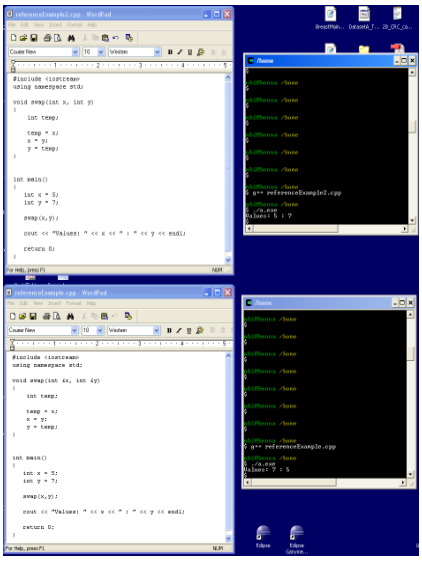
Reference Parameters

- An instance of `account` can be used as a reference parameter using the notation:

```
int balance(account& acc);
```

  - If the member function `balance` changes `acc`, these changes will persist beyond the call
    - Noting that `acc` is a reference variable, so it is automatically dereferenced
- Example…

Reference Parameters



Returning Objects

- A function can return an object, e.g.:

```
account getHighest(account& a1,account& a2){
    if (a1.balance() > a2.balance())
        return a1;
    else
        return a2;
}
```

could be called using the code:

```
account acc3 = getHighest(acc1,acc2);
cout << "Bigger balance is "
     << acc3.balance() << endl;
```

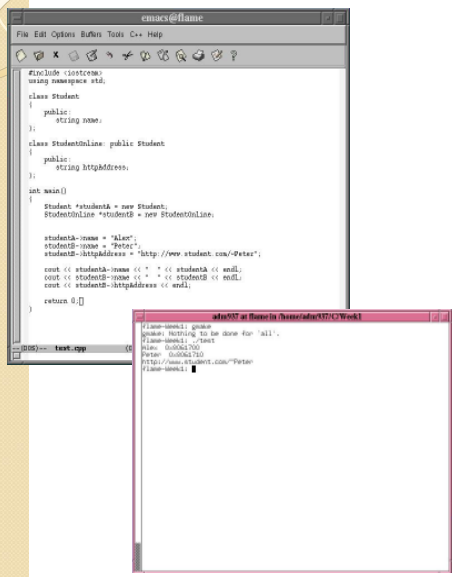
Inheritance / Abstraction

- **Inheritance** is used when a new class adds new functionalities to another more general class. The syntax for inheritance is given by:

```
# include...
class Base {
   public:
   // member functions/variables
}

class Derived:public Base {
  public:
  // specialized member functions
}
```

Example



Inheritance / Abstraction

- **Abstraction** is used when you need to create several implementations for functions, depending on the type of specialization that you want:

```
# include…
class AbstractClassName{
   public:
   // at least one virtual member
  function
}

class Derived:public AbstractClassName{
  public:
  // specialized member functions
}
```

Documenting programs

- What is expected from you in assignments?
  - Inline comments whenever necessary in the .cpp files

```
#include "account.h"
// Constructors
account::account()
{
    acct_balance = 0;
}
account::account(int open_amt)
{
   if (open_amt >=0) // checks if initial value is OK
   {acct_balance = open_amt;}
   else
       {cout << "Initialization value is negative!";}

}
void account::deposit(int amount)
{
    if (amount >=0) // checks if deposit value is OK
    {acct_balance += amount;}
    else {cout << "Deposit value is negative!";}
}
```

Documenting programs

- What is expected from you in assignments?
  - Pre and post conditions

```
class account
{
    public:
    // Constructors
    account();
    account(int open_amt);

   // Members that mutate data
    void deposit(int amount);
    // Pre conditions: Receives a positive integer
    //                 value that will be added to the
    //                 variable acct_balance
    // Post conditions: If the integer is positive, the
    //                 variable acct_balance will be
    //                 changed, otherwise an error
    //                 message will be displayed

    // Members that query data but do not change it
    int balance() const;
    // Pre conditions: None
    // Post conditions: Returns the value of
    //                 acct_balance
...
```