# Basics of C++

**Structure of a class**
**Creating instances of classes**
**Memory management (Heap vs stack allocation)**
**Pointer variables and reference variables**
**Macroguards**

Read chapter 10 of the textbook!

These slides will cover some of the contents, but the textbook is much more detailed.

## Structure of a class

- In C++, classes are defined using two files. One for the class definition (.h extension) and one for the implementation (.cpp extension)
- Consider the following example of a class that stores information related to a bank account. The class has very basic functionality and is shown next

## Structure of a class

```
class account {
    public:
    // Members that are externally visible
    // These are member functions
    // Constructors
    account(const double open_amt = 0.0);

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

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

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

In the class body we only list member method *prototypes* and *not the implementation* code

Member methods that are not permitted to modify the member variables are designated `const`

Parameters that are not permitted to change are also designated `const`

## Structure of a class

```
class account {
    public:
    // Members that are externally visible
    // These are member functions
    // Constructors
    account(const double open_amt = 0.0);

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

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

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

Items in the *public* section are visible and available to everyone who uses the class

The *private* section lists members that are not visible outside an instance of the class

## Structure of a class

```
class account {
    public:
    // Members that are externally visible
    // These are member functions
    // Constructors
    account(const double open_amt = 0.0);

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

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

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

- Notice that the *implementations* of the member methods have not been provided
  - This allows separation of the definition of the methods from their implementation in code
- Typically the `.h` file would include detailed comments on the operation of each member method 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(double);` would suffice in the `.h` file

## Implementation of member methods

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

```
#include "account.h"

// Constructors
account::account(const double open_amt = 0.0)
{
    acct_balance = open_amt;
}
void account::deposit(const double amount)
{
    acct_balance += amount;
}
// ... Other implementations ... ///
bool account::has_funds() const
{
    if (acct_balance > 0.0) {return true;}
    else {return false;}
}
```

- Note that `account.h` is included so that the member method definitions are placed in the correct context
- Each member method definition starts with the name of the class to which it applies, followed by `::`
- This example provides one constructor that accepts either zero or one parameter
- Also note the use of `return` when the member function is not `void`

## Creating instances of classes

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

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

- This instance will be created on the stack
- The member methods for the 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_value = 3000;
my_account.deposit(gift_value);
```

## 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
  - The Java heap is automatically garbage collected by the JVM

## Memory management

- In C++, variables can be created on the stack or in the heap
  - The objects we have created to date have all been created on the stack
  - 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)
    - So, at the end of the code:

```
A = B;
B = B+1;
```

    - A and B will have different values
  - Because Java works in the heap, only the value of the reference variable is copied, i.e. A and B will have the same value, as A and B will refer to the same object.

## Heap vs stack

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

```
account my_account(100.0);
```

  - In the heap (dynamically using pointers)

```
account* my_account = new account(100.0);
```

  - Or using a reference variable

```
account& my_account = some_existing_account;
```

- Objects created on the stack have an automatic garbage collection
- Variables are deleted when they go out of scope. No memory leak!
- Objects created in the heap do NOT have automatic garbage collection. If the pointer is not deleted, the object will remain there, occupying memory space → memory leak.
- Why stack, why heap? Stack is small. Heap is large, all RAM plus hard disks.

## Heap vs stack

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

```
account my_account(100.0);
```

  - In the heap (dynamically using pointers)

```
account* my_account = new account(100.0);
```

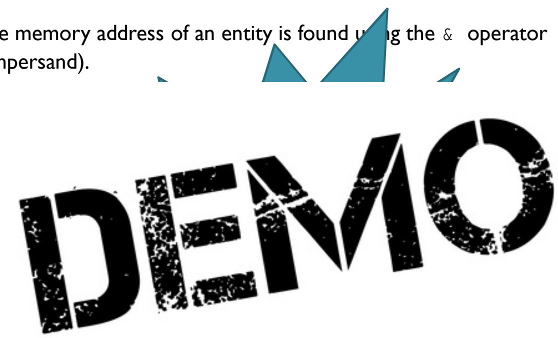
  - Or using a reference variable

```
account& my_account = some_existing_account;
```

- A *reference variable* can be defined as an alias for an object
  - These are initialised when they are defined and cannot be changed
  - An automatic and invisible de-reference occurs when a reference variable is used
- 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

## Heap vs stack

- The memory address of an entity is found using the `&` operator (ampersand).




## Reference Variables vs. Pointers

- 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(30.0);
```

would result in:

- Creation of an instance of class `account` called `acc1`
- Creation of an alias to `acc1` called `acc2`
- Adding 30 to the balance of the `account` object `acc2`

- 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 to it, which means less overhead when dealing with large objects.

## Reference Variables vs. Pointers

- Why not pass a pointer to the object as a parameter?
  - Because pointers should be used only when they will point to different objects throughout their lifetime
  - If a pointer will always point to the same thing, it should not be a pointer, it should be a reference variable.

See page 658 of the textbook!

## Multiple inclusion of the same class

- Good Software Engineering involves *re-use* of software components
- 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, all within the same project.
  - In such a situation the compiler would fail with a *duplicate class definition* error
- This problem is avoided using a so-called *macro guard*

## 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 skips everything after the `#ifndef` statement.

## Parameters and persistence

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

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

using

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

The output would be `5 : 7` .That is, the original parameter values have not been swapped.

## Parameters and persistence

- In Java, all non-primitive parameter calls result in changes to object parameters being persistent
  - Because the copied parameters are *references* to the target object
- In C++ this *is not* the case
  - Instances of classes are copied into the function
  - To avoid that overhead, and if we want the actions on the parameter object to be persistent, we use a *reference parameter.*


## 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;
```


## See you next week!