C++ Definitions:

Class Definition

Defining a CLASS in C++ is much like defining a structure, for example:

```cpp
class Fraction {
    int numerator
    int denominator
};
```

numerator and denominator are said to be **data members**. Note use of Initial case for ClassName (just a convention, not a requirement of C++). Now we are free to use the new class (type) called **Fraction**:

```cpp
Fraction f1, f2;
```

f1 and f2 are **instances** of the Fraction class. An instance of any class is also known as an **object**.

Unlike structures we **cannot** access these elements with

```cpp
f1.numerator // this is illegal, private member
bottom = f2.denominator // is also illegal
```

To be accessible the **private** data members need "public" operators. By default data members are private.

Ignoring the possible existence of public data members, let us now consider how public access functions are declared.

**Member Function Definition**

If data members are private within a class how then can they be updated or even inspected? The answer: via access functions (operators) which are associated with the class. These are called **member functions**.

```cpp
let us expand our example:

class Fraction {
    public:// here we have two access functions
        void create (int, int);
        void print();
    private:
        int numerator;
        int denominator;
};
```

`create` and `print` are prototypes of two functions that access the private members. One possible set of operator declarations is:

```cpp
void Fraction :: create (int num, int denom)
    { // this is an update access function
        numerator = num;
        denominator = denom;
    }
void Fraction :: print ()
    { cout << ((float) numerator) / ((float) denominator);
    }
```

Then the two commands:

```cpp
f1.create(1, 2);
f1.print();
```

first set `f1.numerator` and `f1.denominator`, and then print `0.5`

As with all functions in C++, it is possible and useful to provide default values for the parameters to the member, for example:

```cpp
class Fraction {
    public:
        void create (int num = 0, int denom = 1);
        void print();
    private:
        int numerator;
        int denominator;
};
```

```cpp
Fraction F1;
F1.create (5,3); // set numerator = 5 and denominator = 3
F1.create (5); // sets numerator = 5 and denominator = 1
F1.create (); // set numerator = 0 and denominator = 1
F1.create ( ,3); // how about this one?
```

March 19, 2001
Here we have been using the initializer member function “create” to set the values of the Fraction F1. Of course we could have built constructors to do the same thing at object creation time. For example:

```cpp
Fraction F1 (5,3);
Fraction F2 (5);
Fraction F3 ();
Fraction :: Fraction (int num = 0, int denom = 1) {
    Numerator = num;
    Denominator = denom;
}
```

A second example using constructors and destructors

Dynamic storage allocation and deallocation is a place where constructors and destructors are useful. Consider the case where we want to create a genuine String datatype (class), which retains the string length data member.

A String object could contain strings of arbitrary length, instead of being held in some fixed size array.

The String length could be remembered, thus reducing use of the `strlen()` function, which searches the entire string for the terminating NULL character. We can add special string manipulation operators, instead of being restricted to what `<string.h>` supplies.

```cpp
Our constructor function could be defined as:
String :: String (const char* s)
{   len = strlen(s);
    text = new char[len+1]; // new never fails
    strcpy(text, s);
}
```

after computing the length of the string that s points to, the constructor uses it to allocate enough space for the new string, before finally moving it into place.

To give back the space we need a destructor. If we don't have one, then consider the problems of using the String class within a procedure

```cpp
void procedure ()
{   String S1("abc");  
}
```

when procedure is called the object S1 is created by the constructor, but when the procedure returns the fields text and len of S1 are removed, but the string "abc" itself (that text points to) remains. This failure to return all the memory acquired during procedure execution is called a memory leak.

A destructor, a function that is called automatically when an object ceases to exist, saves the day. Constructors and destructors are counterbalancing pairs. Thus a destructor is a member function just like a constructor and has the same name, but with a tilde (~) pre-pended.

Let us assume that string objects can be declared as:

```cpp
String S1("abc"), S2("pqrs"); // C++
// The C++ case is more general than this
String S1 = "abc", S2 = "pqrs"; // C
```

Anyway, we want S1 and S2 to have these values initially, but to be changed later. Clearly our String class is going to need a member function (constructor operator) that allocates space to our strings and initializes them appropriately:

```cpp
class String {
    String (const char* ); // constructor prototype
    ~String () { delete [] text; }  // destructor

    private:
        char* text; // pointer to a string
        int len; // length of string
}
```

For example:

```cpp
class String {
public:
    String (const char* s);

    ~String () { delete [] text; } // destructor

    private:
        char* text; // pointer to a string
        int len; // length of string

    void procedure ()
    {   String S1("abc");  
    }
}
```

The ~String member function, specified completely above as “inline” code, releases the space pointed to by text.

**Overloading Definition**

In C++, two functions in the same scope may have the same name. When functions are overloaded this way, the C++ compiler determines which one is needed by examining the function's arguments. For example, suppose that two prototypes of the function `foo` exist in the same scope

```cpp
void foo (int);
void foo (double); // now we are overloading
```
Now
foo (1);    // generates a call to foo (int)
foo (1.0);   // generates a call to foo (double)

By this means functions which do the same operation, but with parameters of
different type may use the same name. This cuts down on the number of names
we need to create.

An obvious example might be

int power (int x, int y);
double power (double x, double y);
or
sort (int N, char* s);
sort (int N, int* i);
sort (int N, double* r);  // well, perhaps not!

Because of this overloading of the identifier sort we will need three
constructors and three destructors.

Overloading is also useful for initialization purposes in classes.

Consider

class String {
public:
    String (const char* s);  // constructor
    String () { text = ""; len = 0; }  // overloading
    ~String () { delete [] text; }
    ..........
private:
    char* text;
    int len;
};

with the general constructor declared as:
String :: String (const char* s)
{
    len = strlen(s);
    text = new char[len+1];
    strcpy(text, s);
}

A couple of points. Why not text = NULL; (or text = 0) since we clearly want
the null string in the overloaded function? No good reason, just a common
practice.

Here we have formed the new String constructor called the default
constructor, because it has no arguments, which will be invoked when String
objects are declared without a specified value.
String S;    // default constructor is invoked

In addition to function overloading, C++ also supports operator overloading,
where the same operator serves two different purposes depending on the
context.
cin >> N;    // in C/C++ >> is also right-shift bitstring

Similarly, in C++ the << represents not only an output operation, but also a left
shift operation on bit strings, as it does in C. The compiler recognizes from the
context the intended usage. Different purposes saves on the need to create
more symbols.

There are more advanced examples that are useful in ADT applications, but we
will leave these for now.

Object Oriented Programming

For a language to be object-oriented it must at least include the following three
capabilities

Encapsulation
The ability to define a new type and a set of operations on that type, without
revealing the representation of the type. C++ classes support encapsulation by
restricting access to private data members.

Inheritance
The ability to create new types that inherit properties from existing types. C++
support inheritance through a mechanism known as class derivation.

Polymorphism
The ability of objects that belong to related classes to respond differently to
the same operation. In C++ virtual functions support polymorphism.

Thus we now need to look at class derivation and virtual functions.
Derivation

In C++ we can derive a class from a previously defined one. For example we might need Circle, Square and Triangle objects which can be derived from some general class called Shape. If every shape has in common a colour and an x-y position, and if every shape can change its position and colour, then we could start with:

```cpp
class Shape {
public:
    void change_colour (int new_colour);
    void move (int x_change, int y_change);
    ......
private:
    int x, y; // coordinates of the origin
    int colour; // current colour
    ......
};
```

Shape is said to be the base class, while Circle, Square and Triangle are derived classes. Our aim is to make it possible to re-use code on a grand scale.

If we want to add, say, a Pentagon we need only code the part that is different from Shape.

This whole process can be generalized to libraries of objects, as illustrated in the following relationship diagram.

Each class has all the properties of its base class, plus others that are unique to the object. For example, the objects in the magnitude class have a common need for a relational operator for comparison purposes.

Let us now look at some specifics to explain how Circle inherits the members of Shape, along with its constructors and destructors. The basics:

```cpp
class Circle: public Shape {
    ......
    // Circle will be based on Shape
};
```

Thus radius is in addition to the colour and origin members that it inherits from Shape.

When one class is derived from another, then C++ allows a base pointer to identify an instance of a derived class: For example a variable of type Shape* can point to a Circle or Triangle object.

```cpp
Circle c; // define a circle
Shape* p = &c; // p points to a Circle
```

Thus a parameter of type Shape* can match any actual argument that points to a derived member. Although it appears that the following function requires a Shape argument, it can in fact receive a Circle, Square or Triangle object.

```cpp
void add_to_list (Shape& s) {
    ......
}
```

Hence add_to_list is a highly versatile function that can handle different kinds of shape arguments.