Class Definitions
Visibility Modifiers

- Permissions for data members and member functions:
  - `private`: Can only be accessed by that class
  - `protected`: Can be accessed by subclasses
  - `public`: Can be accessed by anyone
- Class members are `private` by default
- Cannot be applied to the whole class:

```java
default class A;  // Don't do this!
default protected class B;  // Or this!
```
Example

class Box // Class name
{
    public: // Public members section

    Box( int w ) { weight = w; }
    int getWeight( ) const { return weight; }

    private: // Private members section

    int weight;
}; // Notice the semicolon
Example in UML

- Always 3 sections
  - Name
  - Data members
  - Member functions

- Visibility modifiers
  - Public (+)
  - Private (-)
  - Protected (#)

Box
- weight : int
+ getWeight() : int
Inline Methods

- A method that is implemented inside the class definition is called an *inline method*.
- The compiler may choose to expand the body of the method at the point of call.
  - The compiled code executes faster since it avoids the overhead of a function call.
  - Inlining can make the compiled code larger and more complex (usually not desirable properties).
- Use inlining only for very short methods.
- Never use them with loops or recursive calls.
Class Interface

- Usually the class definition is in an *interface* (or *header*) file, and the implementation in an *implementation* (or *source*) file.
  - Interface files usually have a .h extension.
  - Implementation files can have a .cpp, .c++, or .C.
  - The filename does not have to match the class name.
- A `#include` statement is used to include the class definition into the implementation file:
  
  ```
  #include "myclass.h"
  ```
Fully Qualified Names

• Use a `#ifndef ... #define ... #endif` in the header file to avoid including the class definition more than once.

• Methods implemented in the source file use a *fully qualified* function name.
  • This avoids conflicts with other classes that have a method with the same name.
  • A fully qualified name consists of the class name, a double colon, and the method name:

    ... ClassName::methodName ...
Example

• box.h

```c
#ifndef BOX_H
#define BOX_H

class Box
{
    public:
    Box( int w );
    int getWeight( ) const;

    private:
    int weight;
};

#endif
```

• box.c

```c
#include "box.h"

Box::Box( int w )
{
    weight = w;
}

int Box::getWeight( ) const
{
    return weight;
}
```
Forward Declaration

- A class must be defined before it is used
  - If a class name is used in another class definition, the first class must be defined prior to the use
  - This could be a problem if the first class also uses the name of the second class
- A *forward declaration* is used to declare the name of a class
  - Permits pointers to the class to be declared
  - Cannot invoke methods in the class (since they're not defined yet)
Example 1

class Chicken;

class Egg
{
    public:
    Chicken * parent;
};

class Chicken
{
    public:
    Egg * children;
}
Example 2

class Link;

class List {
    public:
        ...
        private:
            Link * head;

};

class Link {
    Public:
        int value;
        Link * next;
        Link * prev;

    Link( int v,
            Link * n,
            Link * p );

    void addBefore( int val,
                        List * );

};
Constructors

- Constructors serve two purposes: they create and initialize an object.
- A constructor is a method with the same name as the class, and does **not** have a return type.
- There are three types of constructors:
  - A **default** constructor takes no arguments.
  - An **ordinary** constructor has some arguments.
  - A **copy** constructor is used to make copies (clone).
Copy Constructor

• A copy constructor is used to make a copy of an object value.
  • It takes an instance of the same class as a constant reference argument:
    \[
    \text{Box( const Box & b );}
    \]
  • A copy constructor is often called implicitly, such as when passing by value:

\[
\text{Box a; }\quad \text{// Default constructor gets called implicitly, too.}
\]
\[
\text{doStuff( a ); }\quad \text{// Copy constructor called.}
\]
Example

class Box {
public:
    Box( ) // Default constructor
    { weight = 0; }

    Box( int w ) // Ordinary constructor
    { weight = w; }

    Box( const Box & b ) // Copy constructor
    { weight = b.weight; }

private:
    int weight;
};
Initializers

- Data members can be initialized by an assignment in the constructor, or by an *initializer*:

  ```cpp
class Box {
    public:
    Box( ) : v( 0 ) { }
    Box( int v ) : val( v ) { }
    ...
  };
```

- Use initializers whenever possible to avoid initializing a value twice (first by the default constructor, then by the ordinary constructor).
Double initialization

class Box {
    public:
    Box( int w )
    { weight = w; }

    private:
    int weight;
};

• The default constructor for weight is called before the function body of the constructor
• Then weight is changed
Order of Initialization

- Class members are initialized in the order they are declared in the class body rather than in the order of the initializers

```cpp
// This class is broken
class Order {
    public:
        Order( int i ) : one( i ), two( one ) { }
        int test( ) const { return two; }

    private:
        int two;  // initialized first
        int one;  // initialized second
}
```
Example

• Correct class definition:

```cpp
class Order {
    public:
    Order(int i) : one(i), two(one) {} 
    int test() const { return two; }

    private:
    int one; // initialized first
    int two; // initialized second
};
```
Combining Constructors

• It is not allowed to call one constructor from another constructor:

```cpp
class Box {
  public:
  Box( int a ) : val1( a ) { }
  Box( int a, int b ) : val2( b )
  {
    Box::Box( a ); // This will not work!
  }
  private:
  int val1, val2;
};
```
Solution 1

• Use *default arguments*:

```cpp
class Box
{
    public:
        Box(int a, int b=7) : val1(a), val2(b) { }

    private:
        int val1, val2;
};
```

• Even though only one constructor is defined, it can be used with one or two arguments.
Solution 2

- Put the common initialization code in a separate private function:

```cpp
class Box {
    public:
    Box(int a) { initialize(a); }
    Box(int a, int b) {
        initialize(a);
        ...
    }
    private:
    int initialize(int c);
};
```
Destructors

• The *destructor* is implicitly called when an object is deleted
  • Object may have been explicitly deleted using delete
  • An object could also be automatically deleted at the end of a function if the object is stack-resident
  • The destructor is never called directly

• The destructor is defined using a tilde followed by the class name and takes no arguments:

  ```cpp
  ~Box();
  ```
Destructors cont.

- The destructor usually deletes any heap-resident memory the object may have allocated:

```cpp
class Storage {
public:
  Storage( int s ) { space = new int[s]; }
  int & operator[]( int i )
  { return space[i]; }

~Storage( ) { delete [] space; }

private:
  int * space;
};
```
The keyword **this**

- Every method has a pointer named **this** which points to the object the method was invoked on.

```cpp
class Box {
public:
    Box( int w ) : weight( w ) { }

    Box & doStuff() {
        this->weight = 73;
        return *this;
    }

private:
    int weight;
};
```
Nested Classes

- One class can be defined within another class.
  - If the nested class is defined in the private section, only the outer class will know it exists.
  - To access a nested class from outside the outer class, a fully qualified name must be used (suppose Link is public):

```cpp
class List
{
    private:
    class Link
    {
        int value;
        Link * next;
    };
    Link * head;

    public:
    ...
};

List::Link * l;
```
Friends

A class can have *friends* that are allowed to access its private data members and functions:

```cpp
class Box {
public:
    Box( int w ) : weight( w ) { }

    // Allow access for global function operator<<
    friend ostream & operator<<( ostream & out);

    // Allow class Crate to access val
    friend class Crate;

private:
    int weight;
};
```