Classes

Enhanced structures, the foundation of Object Oriented Programming
A class allows the user to define objects with associated functions and operators.

```
// An elementary implementation of type String
#include <iostream.h>
#include <string.h>

const int maxLen = 255;

class String {
    char s[maxLen + 1];
    int len;
    public:
    void assign (char const *st) {
        strcpy(s, st); len = strlen(st); }
    int length () { return len; }
    void print () { cout << s << endl; }
};
```

Objects are instances of classes:

String s1, s2;
Properties of Classes

- Enhanced **struct**: Member fields can be data, functions, and operators (functions implementing unary/binary operators)
- Information hiding: **public** and **private** (more types later)
- Private members hide (encapsulate) implementation details, and are accessible only by other members of the class
- **public** fields provides the interface for users, and are accessible by everybody
- By default, **class**’s members are private (while **struct**’s members are public)
- In fact, **struct** and **class** differ only by their default field type:
  
  \[
  \text{struct} = \text{class} + \text{public} \\
  \text{class} = \text{struct} + \text{private}
  \]
#include <iostream.h>
#include "my_string.h"    // Testing the class String

static char const kg[] = "My name is Kurt Gödel";

int main () {
    String one, two;    // Defining two objects

    one.assign ("My name is Alan Turing.");
    two.assign (kg);

    // Print the shorter of one and two
    if (one.length () <= two.length ())
        one.print();
    else
        two.print();

    return 0;
}
**Remarks about Classes**

- Member functions have one extra ("hidden") parameter over the visible ones:
  
  ```cpp
  int length();    // declaration inside class
  one.length()    // usage with an object of the class
  ```

- Each member function has an implicit parameter, which is an object of that class: the object which the function "is working on"

- At a function call, an argument for this parameter MUST be an object of the same class: No conversions are allowed

- The keyword `this` points to this parameter (usage given later)

- A member function may, and should, declare if it does not modify its hidden parameter:
  
  ```cpp
  int length() const { return len; }
  void print() const { cout << s << endl; }
  ```
Construction of Objects

The programmer may specify what happens when an object is “born” by using a constructor function:

```cpp
#include <iostream>
#include <cstring>
using std::cout;
using std::endl;

class String {
  public:
    enum {maxLen=255};
    // static const int maxLen=255;
    String (char const *st) // A constructor.
      { strncpy(s,st,maxlen); len=strlen(st); }
    int  length () const { return len; }
    void print () const { cout << s << endl; }
  private:
    char s[maxLen + 1];
    int len;
};
```

A better way for integral constants

This method is not only for ints
Using Constructors

```cpp
#include <iostream.h>
#include "my_string.h"
static char const kg[] = "My name is Kurt Godel";

int main() {
    String one("My name is Alan Turing");
    const String two(kg);

    // Print the shorter of one and two
    if (one.length() <= two.length())
        one.print();
    else
        two.print();

    ....
    return 0;
}
```

If `print()` were not a `const` function, then there would be syntax errors
A constructor may be called explicitly, returning an object of the class:

```java
String one("abc");  // Constructor is applied to one
```
or

```java
String one = String("abc");  // Constructor creates
// a temporary object, which is then copied to object one
```

The programmer may implement more than one constructor (differing by their signatures)

If no constructors are implemented, the compiler supplies a default constructor, having no arguments and doing nothing

If any constructor is implemented, then the Def. Ctor disappears

When a class must have an argument-less constructor (see later) and the Def. Ctor disappears, the programmer has to supply one

We will talk later about copy constructors
The programmer may specify what happens when an object “dies” by using a destructor function:

class String {
  public:
    String (char const *st) { // A constructor
      len = strlen(st);
      s = new char[len+1];    // Allocate space as needed
      strcpy(s, st);
    }
    ~String () {        // A destructor
      delete[] s;        // Free the allocated space
    }
    int length () const { return len; }
    void print () const { cout << s << endl; }

  private:
    int len;          // A pointer to variable length string
    char *s;
};
Explicit calls to destructors are *very rare*. In practically all cases, an implicit call is automatically created at the end of the `{ }` scope.

```cpp
static char const kg[] = "My name is Kurt Godel";

int main() {

    String one ("My name is Alan Turing");
    String two (kg); // Constructors are invoked.

    ...

    return 0; // one.~string() and two.~string() // are automatically called here as both one // and two go out of scope

}
```
As we already know, the :: operator indicates **scope** for name spaces. But it can also indicate a scope of a class.

**Example:**

```cpp
String::print()  // The full name of the function is the print() member of class String.
```
String s1(“abc”);

cout << s1.s; // Error: Inner structure is private
s1.print(); // OK: Interface dictates the protocol
Member functions may be defined in the body of the class declaration *(implicitly inline)*:

```cpp
class String {
    public:
        enum {maxLen = 255};
        void print() const
            { cout << s << endl; }
    private:
        char s[maxLen + 1];
};
```

Or may be defined separately:

```cpp
class String {
    public:
        enum {maxLen = 255};
        void print() const; // Declaration
    private:
        char s[maxLen + 1];
    };

    // Definition (note the scope)
    inline void
    String::print() const
        { cout << s << endl; }
```

Can be accessed by `String::maxLen`.

**Note the direct access to data-members (of this) from member functions.**
Levels of visibility of names referenced in a member function of a class C:

- Local to the member function: according to the hierarchy of scopes within the function
- Data members of C
- Outside C:
  - If C is defined inside a function, then according to the hierarchy of scopes within that function
  - Global in the compilation unit
Suppose we want to build a class for complex numbers.

```cpp
class Complex {
    public:
        double real, imag;
    Complex (double r = 0, double i = 0) {
        real = r;
        imag = i;
    }
};
```

With this setting, users access the data members directly:

```cpp
#include "Complex.h"
Complex c;
do_something_with (c.real);
```
Exposing the inner structure of objects results in unstable code and maintenance hardship.

A better way:

```cpp
#include <cmath>
using std::sqrt;

class Complex {
    public:
        Complex (double r = 0, double i = 0) :
            real(r), imag(i) {}
        double real_part() const { return real; }
        double imag_part() const { return imag; }
        double abs() const
            { return sqrt(real*real+imag*imag); }
    private:
        double real, imag;
};
```

The **private** part is not of real interest for the user. The compiler must see it; the user can see it but cannot refer to it.
Using Classes

The client code will look like this:

```
#include "Complex.h"

Complex c;
do_something_with(c.real_part());
```

Since `Complex::real_part()` is inline, there is no runtime function-call overhead.
Modifying Classes

• Suppose that we want to reimplement class Complex and use internally polar coordinates.

• In the first version, the user will need to change any piece of code that refers to the internal representation.
  • Lots of work!
  • Prone to errors!

• In contrast, the second version require no change of user code because implementation details are hidden behind a well-defined implementation-independent interface.

Encapsulation!
#include <cmath>
using std::sqrt; using std::atan2;

class Complex {
  public:
    Complex (double re = 0, double im = 0) {
      r = sqrt(re*re+im*im);
      theta = !re ? (im>=0?PI/2:-(PI/2)) : atan2(im,re);
    }

    double real_part() const { return r*cos(theta); }
    double imag_part() const { return r*sin(theta); }
    double abs() const { return r; }

  private:
    double r, theta;
};

The client code remains the same (but must be recompiled).

Look for Handle Class, Cheshire Cat Pattern, or Opaque Pointer

Not the final version!

Only because the members are inline.
Using accessors - *setters* and *getters* – are in general (not always) an indication of a bad design. It is preferred *not* to ask an object to give you some internal information, but to ask it to do some job for you. The right way (in general) is:

```cpp
#include <cmath>
using std::sqrt;

class Complex {
  public:
    Complex (double r = 0, double i = 0) : real(r), imag(i) {};
    Complex add (const Complex & ) const;
    Complex conjugate (const Complex & ) const;
    double abs () const
    { return sqrt(real*real+imag*imag);}
  private:
    double real, imag;
};
```

The private part is not of real interest for the user.
C++ provides the capability of having static members in classes:

- static data members are unique and belong to the whole class.
- static member functions do not have a this parameter, hence, they can access directly only static data members.
- A static member a in a class C can be referred to either by using any object o of the class (o.a) or by using the scope operator (C::a).

This way we can modify class Complex better than before:

```cpp
class Complex {
public:
    Complex (double re = 0, double im = 0) :
        r(sqrt(re*re+im*im)), theta(arg(re,im)) {}
    static double arg (double x, double y) {
        return x != 0. ? (y >= 0 ? PI/2 : -(PI/2)) : atan2(y,x);}
private:
    double r, theta;
};
```

```cpp
double arg = Complex::arg(x,sqrt(2));  // Usage
```
Example of Static Data Members

```cpp
#include <cmath>
#include <stdio.h>

class SomeObject {
public:
    SomeObject() { ObjCount++; CrntObjects++; }
~SomeObject() { CrntObjects--; }

    static void PrintStatus(int i) {
        fprintf(stderr, "%d SomeObjects were allocated, currently active %d\n", i, ObjCount, CrntObjects);
    }

private:
    static int ObjCount, CrntObjects;
};

void func(void) {
    SomeObject o1, o2, *o3,
        *o4 = new SomeObject;
    SomeObject::PrintStatus(1);
    o3 = new SomeObject;
    delete o4;
    SomeObject::PrintStatus(2);
}

// Only in C++ file (not header)
int SomeObject::ObjCount = 0;
int SomeObject::CrntObjects = 0;

int main() {
    func();
    SomeObject::PrintStatus(3);
    return 0;
}
```

1) 3 SomeObjects were allocated, currently active 3
2) 4 SomeObjects were allocated, currently active 3
3) 4 SomeObjects were allocated, currently active 1

Method Prototypes & Overloading

Like non-member functions, member functions can be overloaded. It’s not always simple to identify which member function the compiler will call.

class Overload {
public:
    Overload (...) {...} // Constructor
    // void f (int i, int j = 0) {...}
    // void f (int i, int j) {...}
    void f (int i, double d) {...}
    void f (double d, int i) {...}
    void f (int i, int j, int k = 17) {...}
};
Overload obj(...);
obj.f (2,3); // Which function “f” will be called?

Morale: Avoid confusingly-similar function calling sequences, although the compiler may issue an function-call ambiguity message.