Classes & Objects

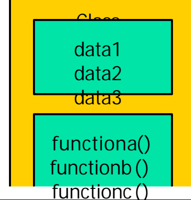
- classes
- member data and member function
- access control and data hiding
- instantiation of objects
- class constructor and destructor
- objects as function parameters
- constant member functions
- static member data
- self-reference

Classes

- The C++ class mechanism allows the programmer to define her own data types (user-defined types) that can be used like the built-in data types.
- A program that provides classes that closely match the concepts of the application is easier to understand and makes it more concise.
- Classes are typically used to define abstractions that do not map naturally to the built-in data types: e.g. complex numbers, date, time, vector, circle.
- The class mechanism allows it to separate the details of the implementation from the interface that specifies how to use and interact with objects of a class.

Classes

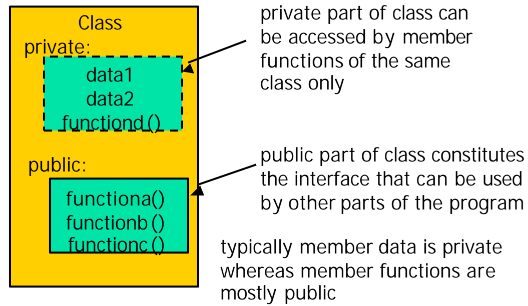
- A class is a user-defined prototype from which one can *instantiate* objects of a particular type in the same way one generates integer variables using the built-in data type int.
- A class contains data (member data) and functions (membership functions).



Class Definition (Interface)

```
class Date                              // declares class name
{
    private:                            // not visible outside the class
      int day, month, year;             // member data
    public:                             // visible interface
      void init(int d, int m, int y);   // initialize date
      void add_year(int n);             // add n years
      void add_month(int n);            // add n months
      void add_day(int n);              // add n days
      void show_date();                 // displays date
};                                      // do not forget the ; here !!!
```

Access Control



private part of class can be accessed by member functions of the same class only

public part of class constitutes the interface that can be used by other parts of the program

typically member data is private whereas member functions are mostly public

Data Hiding

- *Data hiding* is mechanism to protect data from accidentally or deliberately being manipulated from other parts of the program
- The primary mechanism for *data hiding* is to put it in a class and make it private.
- Private data can only be accessed from within the class
- Public data or functions are accessible from outside the class

Class Implementation

```
void Date::init(int d, int m, int y)
{
    day=d;
    month=m;
    year = y;
}

void Date::add_month(int n)
{
   month+=n;
   year+= (month-1)/12;
   month = (month-1) % 12 + 1;
}
```

Class Implementation

```
#include <iostream.h>

void Date::show_date()     // Date:: specifies that show_date is a
                           // member function of class Date
{
    cout << day << "." << m << "." << y << endl;
}
void Date::add_year(int n)
{
    year+=y;
}

void Date::add_day(int n)
{
    ....
}
```

Instantiation of an Object

```
class Date  // class definition
{ … };
void main ()
{
    Date d1;   // instantiate  a variable/object d1 of class Date
    Date d2;   // instantiate a second object d2 of class Date
   d1.init(15,3,2001);   // call member function init for object d1
   d1.show_date();      // member function show_date for d1
   d2.init(10,1,2000);  // call member function init for object d1
   d2.show_date();      // member function show_date for d1
}
```

Member Function

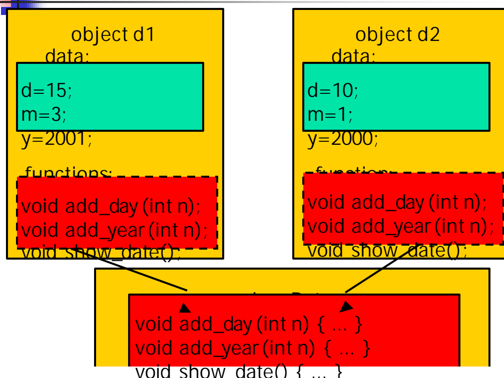
- In conventional non-member functions there is no explicit connection between the data type and the function

```
class Date { public: int day, int month; int year, }; // class definition
void init(Date& today, int d, int m, int y);  // non-member function
Date today;
init(today,15,3,2001);   // call to non-member function
```

- Such a connection can be established by declaring the function as a member function
- Member functions can be only invoked in conjunction with an object of the class for which they are defined

```
class Date { int day, int month; int year,
void init(int d, int m, int y);  // member function
};
Date today;
today.init(15,3,2001);   // the dot operator makes the connection
                         // between the object and the function explicit
```

Objects & Classes



Class Member Functions

- A class member function can only be called in association with a specific object of that class using the dot operator (period)

```
Date d1;  // object of class d1
d1.init(12,3,2001);  //  call member function init for d1
init(13,2,2000);  // illegal, no object associated to call init
```

- A member function always operates on a specific object not the class itself.
- Each objects has its own data but objects of the same class share the same member function code
- In other OO languages member function calls are also called *messages*.

Constructor

- A constructor is a member function that is automatically invoked when a new object of a class is instantiated.
- The constructor must always have the same name as the class and has no return type.

```
class Date
{
    Date();    // constructor for class Date
};
```

Constructor

```
class Date
{
public:
    Date();    // constructor for class Date
private:
  int day;
  int month;
  int year;
};

Date::Date() : day(1), month(1), year(1900)
// initialize int day with 1, int month with 1 etc.
{
//empty body
};
```

Member Initialization

- two types of member initialization in class constructor
  - by initialization : members are initialized before the constructor is executed (necessary for data members tha have no default constructor)
  - by assignment : members are created per default constructor first and then a value is assigned to them

```
Date::Date(int d, int m, int y) : day(d), month(m), year(y)
{
}

Date::Date(int d, int m, int y) // assignment initialization
{
  day=d;
  month=m;
  year=y;
}
```

Overloaded Constructors

```
class Date
{
public:
    Date();    // default constructor with standard date 1.1.1900
    Date(int d, int m, int y); // constructor with day, month, year
    Date(string date); // constructor with string
private:
  int day;
  int month;
  int year;
};
```

Overloaded Constructors

```
#include <stdlib>
#include <string>
Date::Date(int d, int m, int y)   : day(d), month(m), year(y)
{
}
Date::Date(string str) // constructor with string dd.mm.yyyy
{
   day=atoi(str.substr(0,2).c_str());
   month=atoi(str.substr(3,2).c_str());
   year=atoi(str.substr(6,4).c_str());
}
```

Overloaded Constructors

```
void main()
{
  Date d1; // default constructor  date 1.1.1900
  Date d2(12,9,1999);  // Date(int d, int m, int y) constructor
  Date d3("17.2.1998"); // Date(string date) constructor
}
```

Constructor with Default Values

```
class Date
{
public:
  Date(int d=1, int m=1, int y=1900); // constructor with
                              // day, month, year and default values
};
Date::Date(int d, int m, int y) : day(d), month(m), year(y) {}
void main()
{
  Date d1; // 1.1.1900
  Date d2(5);  //  5.1.1900
  Date d3(15,8); // 15.8.1900
  Date d4(12,10,1998); // 12.10.1998
}
```

Copy Constructor

- The copy constructor initializes on object with another object of the same class.
- Each class possesses a built-in *default copy constructor* which is a one-argument constructor which argument is an object of the same class
- The default copy constructor can be overloaded by explicitly defining a constructor Date(Date& d)

```
void main()
{
  Date d1(12,4,1997);
  Date d2(d1);  // default copy constructor
  Date d3=d1;  // also default copy constructor
}
```

Destructors

- Destructors are automatically called when an object is destroyed
- The most common use for destructors is to deallocate memory

```
class Date
{
public:
   Date();    // constructor
   ~Date();   // destructor
…
};

if …
   {
     Date d1;
     …
   } // destructor for d1 is invoked
```

Objects as Arguments

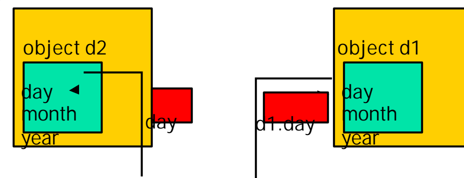
- Class objects can become arguments to functions in the same way as ordinary built-in data type parameters

```
class Date
{
   int diff(Date d);  // computes the number of days between two dates
};
int Date::diff(Date d)
{
    int n=day-d.day;    // d.day access member data day of object d
    n+= 30 * (month – d.month);  // approximately correct…
    n+= 365 (year – d.year);
    return n;
}
Date d1(14,5,2000);
Date d2(10,4,2000);
cout << d1.diff(d2) << endl;   // difference between d1 and d2
```

Objects as Arguments

```
class Date
{
   void add_days (Date d, int n);  // computes the date d + n days
};
void Date::add_days(Date d, int n)
{
   day=d.day+ n % 30;
   month = d.month + (n % 365) / 30;
   year = d.year + n / 365;
}
Date d1(14,5,2000);
Date d2;
d2.add_days(d1,65); // d2 set to 29.7.2000
```

Objects as Arguments



d2.add_days(d1,65);

Member function of d2 refers to its own data member day directly

Member function of d2 refers to data member day in object d1 using the dot operator

Returning Objects from Functions

```
class Date
{
   Date get_date (int n);  // returns the date + n days
};
Date Date::get_date(int n)
{
   Date temp;
   temp.day= day+ n % 30;
   temp.month= month+ (n % 365) / 30;
   temp.year = year + n / 365;
   return temp;
}
Date d1(14,5,2000);
Date d2=d1.get_date(65);   // d2 set to 29.7.2000
```

Constant Member Functions

- A member function that is declared as constant does not modify the data of the object

```
class Date
{
   int year() const;   // const year() does not modify data
   void add_year(int n);  // non-const add_year() modifies data
};

int Date::year() const   // defined as const
  { return year; }
int Date::add_year(int n)
  { year+=n; }
```

Constant Member Functions

- A const member function can be invoked for const and non-const objects, whereas a non-const member function can only be invoked for non-const objects

```
void f(Date& d, const Date& cd)
{
   int i=d.year();   // ok
   d.add_year(2); // ok
   int j=cd.year();  // ok
   cd.add_year(3); // error cannot change const object cd
}
```

Static Member Data

- Normally each object possesses its own separate data members
- A static data member is similar to a normal static variable and is shared among all objects of the same class
- There is exactly one copy of a static data member rather than one copy per object as for non-static variables

Static Member Data

```
class Foo
{
 public:
    Foo() { counter++; }
    ~Foo() { counter--; }
    void display()
    {
      cout << "There are " << counter << " Foo objects!";
    }
private:
  static counter;
};
```

Static Member Data

```
Foo f1;
Foo f2;
f1.display(); // 2 foo objects
{
    Foo f3;
    f1.display(); // 3 foo objects
} // f3 gets destroyed
f1.display(); // 2 foo objects
```

Static Member Data

```
class Date
{
    int days_per_month();
    int days_per_year();
    bool leap_year();
    static int days_in_month[12] =
       { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
};
```

Static Member Data

```
bool Date::leap_year() {
return((( year % 4  == 0)  && ( year % 100 != 0)) || ( year % 400 ==
    0));
}
 int Date::days_per_month() {
 if ((month==2) && leap_year())
   return 29;
   else
    return days_in_month[month-1];
}
int Date::days_per_year() {
If leap_year()
  return 366;
else
  return 365;
}
```

Self-Reference

- Each member function knows what object it was invoked for and can explicitly refer to this object using the pointer *this*.
- Self-reference is for example to return a reference to the updated object so that operations can be chained (in particular when using operators)
- Example: cout << "Hello " << "World" << endl; the operator << returns a reference to the cout stream object

Structures and Classes

- By definition a struct is a class in which members are by default public, whereas in a class data and function members are by default private.

```
struct s
{
…
};
 // is equivalent to
class s
{
  public:
  …
};
```

Recommended Reading

- Stroustrup: chapters 10
- Lafore : chapters 6
- Lippman : chapters 13, 14