Classes (Book ch.: 10)

structs and classes are the same except for default access mechanism (public vs. private).

class Date {
    int d, m, y;
public:
    void init(int dd, int mm, int yy); // initialize
    void add_year(int n); // add n years
    void add_month(int n); // add n months
    void add_day(int n); // add n days
};

General rule: Data should be private.

General rule: Initialization should not be done by a separate function (such as init()) but by the constructor.
Classes (contd.)

class Date {
    int d, m, y;
public:
    Date(int dd, int mm, int yy);
    // ...
};

Date::Date(int dd, int mm, int yy) {
    d = dd ? dd : today.d;
    m = mm ? mm : today.m;
    y = yy ? yy : today.y;
    // assumes that a global today is available
    // accessing today.d etc. is okay here.
}

Date today = Date(13, 2, 2007);
Date tomorrow = Date(13, 2) // illegal;
Constructors (contd.)

Default argument values:
Date(int dd=0, int mm=0, int yy=0);

Or multiple constructors:
Date(int dd, int mm, int yy);
Date(int dd, int mm);
Date(int dd);
Date(); // Each must be defined

What is “in-lining”?
Static members

Consider:
Date::Date(int dd, int mm, int yy) {
    d = dd ? dd : today.d;
    m = mm ? mm : today.m;
    y = yy ? yy : today.y;
    // assumes that a global today is available

Not a good approach (using globals):
Instead use a static member.
static member: Belongs to the class.

class Date {
    int d, m, y;
    static Date today;
...
Date Date::today(13,2,2007);

Another example: Assigning unique numbers to bank accounts.
Constant member functions

class Date {
    int d, m, y;

public:
    Date(int dd, int mm, int yy);
    // ...
    int day() const {return d;} //in-lined
    int month() const {return m;} //in-lined
    int year() const; }

int Date::year() const {return y; }

const: fn. does not modify class state (compiler will catch violations).

void f(Date& d, const Date& cd) {
    int i = d.year(); // okay
    d.addYear(1); // okay
    int j = cd.year(); // okay
    cd.addYear(1); // Not okay
    ...
}
Self-reference

class Date {
    // ...
    public:
    Date& addYear(int n);
    Date& addMonth(int n);
    Date& addDay(int n);
    ...
    Date& Date::addYear(int n) {
        y += n; // or this->y += n;
        return *this;
    }
    addMonth and addDay similar.

    void f(Date& d) {
        d.addYear(1).addMonth(2).addDay(3);
        ...
    }
Operator overloading

Key point: Operators are just functions with a special name/syntax.

```cpp
bool operator==(Date a, Date b) {
    return ((a.day()==b.day()) &&
            (a.month()==b.month()) && (a.year()==b.year()));
}
```

Other possible operators:

```cpp
bool operator!=(Date a, Date b);
bool operator<(Date a, Date b);
bool operator>(Date a, Date b);
Date& operator++(Date a); //inc. by 1 day
Date& operator--(Date a);
Date& operator+=(Date a, int n); //add n days
Date& operator-=(Date a, int n); //subtract n
Destructors

A destructor is used to release memory (or other resources).

class Name {
    const char* s; ... };

class Table {
    Name* p;
    int sz;

class Table {
    Name* p;
    int sz;

    Table(int s) { sz = s; p = new Name[sz]; }

    ~Table() { delete [] p; }
    // not enough!

    Or may be it is!: 10.4.7: When an array is deleted, the destructor is invoked on each (constructed) element of the array.
**Default Constructor:**
This is a constructor that can be called without an argument:

class Table {
  ...
public:
  Table(int s=15) { ...}
  ...
}

struct Tables {
  int i;
  int vi[10];
  Table t1;
  Table vt[10];
};

Tables tt;

tt will be initialized using a (generated) default constructor: calls Table(15) for t1 and each element of vt[]. (i and vi[] are not initialized.)

Any class containing consts or references cannot be default-constructed unless a default constructor is explicitly defined (why?)
Class objects as members

class Club {
  string name;
  Table members;
  Table officers;
  Date founded;
public:
  Club(const string& n, Date fd);
};

Each member’s constructor will be invoked and may expect arguments: supply them in the \textit{initializer list}:

\begin{verbatim}
Club::Club(const string& n, Date fd)
  : name(n), members(), officers(), founded(fd)
  {
    ... 
  }
\end{verbatim}
Main point: When an object of type Club is constructed, its member objects are first constructed and then the Club object is constructed. Destructors in reverse order.

Member initializers are necessary for member objects that don’t have default constructors, for const members, and for reference members.

class X {
    const int i; Club c1; Club& rc;
public:
    X(int ii, const string& n, Date d, Club& c) : i(ii), c1(n,d), rc(c) {}
Copy constructor, assignment oper.:
We often want to create a new object as a copy of an existing object or assign an existing object to another:

Table t1;
Table t2 = t1; // member-wise copy
Table t3;
t3 = t2; // member-wise

This maybe a bad idea:
possible shared objects - aliasing.
Better approach: Define your own.

Table::Table(const Table& t) { // copy const.
    sz = t.sz; p = new Name[sz];
    for (int i=0; i<sz; i++) p[i]=t.p[i];
}

Table& Table::operator=(const Table& t){//assign
    if (this != &t) {
        delete[] p;
        sz = t.sz; p = new Name[sz];
        for (int i=0; i<sz; i++) p[i]=t.p[i];
    }
    return *this;
}

Reading: Chapter 10
(some of it is rather technical; focus on the parts we have talked about).