

Structures Revisited

- What's a structure? What's the keyword to define a structure?
- What are structures used for?
- Is a structure definition an executable statement? Can it be put in a header file?
- Why `including` a header file with a structures definition multiple times is a problem? How is this problem solved?
- What are `#define #ifndef #endif`?
- What is a structure variable?
- How are elements of a structure called?
- Do elements of the same structure (different structures) have to have unique names?
- How can structures be initialized? What happens when one structure variable is assigned the value of another?
- Can structures be passed as parameters? by value? by reference? Can a function return a structure?

Classes

What's Wrong with Structures?

- Structure is an *aggregate* construct providing *encapsulation*
- *encapsulation* - combining a number of items in a single entity
- consider implementing date:
 - as structure:

```
struct Date{
    int month;
    int day;
    int year;
};
```
 - and a set of functions manipulating dates:

```
void set_date(Date &d, int, int, int);
void add_year(Date &d, int n);
bool compare(Date &d1, Date &d2);
```
- problems:
 - there is no explicit connection between data type (structure) and these functions
 - it does not specify that the functions listed should be the only ones that access and modify date variables
 - if there is a bug in Date manipulation - it can be anywhere in the program
 - if modification of Date is needed - all program needs to be updated

Class Definition

- Class is an aggregate data type
- Class may contain *member* variables and *member* functions
- member variables and member function prototypes are declared within class definition
- member functions can manipulate member variables without accepting them as parameters

```
class Date { // class name
public: // ignore this for now
    void set(int, int, int);
    int getday();
    int month;
    int day;
    int year;
}; // don't forget the semicolon
```
- a variable of type class is called *object* (how is a variable of type structure called?)

```
Date mybday;
```
- each object has member variables and can call member functions of its class. Addressing the members is done using dot-operator:

```
mybday.set(10, 26, 68);
cout << mybday.day;
```

Public and Private Members

- `public/private` attributes control the way the class members are accessed (why do we want to do that?)
- public member can be accessed within member functions and directly (with dot operator)
- private member - can only be accessed within member functions

```
class Date { // class name
public:
    void set(int, int, int);
    int getday();
private:
    int month;
    int day;
    int year;
}; // don't forget semicolon
```
- make member variables private, make functions either public or private. This restricts manipulation of variables to member function which makes debugging and changes in class easier
- examples:

```
mybday.set(10, 5, 99); // good
mybday.year=99; // ERROR - private member
```

Member Function Definitions

- functions can be defined either inside or outside class definition
- outside definition - class name (called *type qualifier*) and *scope resolution operator* (`::`) precedes function name:

```
void Date::set(int m, int d, int y){
    month=m; // no dot with member variables
    day=d; // no declaration of member variables
    year=y;
}
```
- to define inside - replace prototype with definition

```
class Date {
public:
    void set(int, int, int);
    int getday(){return(day);} //still needs ";"
private:
    int month;
    int day;
    int year;
}; // don't forget semicolon
```
- function defined inside is called *in-line* function - at compilation the function code replaces every invocation - use only for small functions

Mutators and Accessors

- *accessor function* - member function that does not modify the state of an object (only returns the information about the object's state)
- *mutator function* - member function that modifies the object's state
- accessors should be marked with "const" so that compiler can find accidental object modification

- also remember to use "const" with parameters that are not modified

```
class Date {
public:
    // mutator
    void set(int, int, int); // mutator
    int getmonth() const; // accessor
    // accessor in-line
    int getday() const {return(day);};
private:
    int month;
    int day;
    int year;
};
```

- separate mutators and accessors - a function should not do both
- since variables are private they all need accessor
- are we missing an accessor in Date?

Constructors

- *constructor* - mutator that is invoked automatically when object is declared
- used to assign an initial state to the object (initialize object)

- constructor does not have a return value and it has the same name as class, do NOT put void as a return value of constructor

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

- outside constructor definition (can also be inlined)

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

- constructor is invoked at declaration: `Date mybday(10,26,99);`
or directly: `yourbday=Date(10,26,00);`

Multiple/Void Constructors

- class can have multiple constructors
- which constructor to call is determined by number and type of arguments
- it is bad style to define constructors that differ by type of arguments only - confusing
- if no constructors declared - object can be defined without initialization
- *void constructor* - a constructor that does not take arguments
- caveat: if at least one constructor declared - object has to be always initialized (need a void constructor)

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

- calling default constructor (if constructors defined): `Date mybday;`
- what is this? `Date mybday();`

Program Layout

- program with objects is laid out as follows:
 - header - class definition (inline function definitions), global constants and other non-executable constructs related to class
 - program file - member function definitions
- multiple related classes may be put in one header/program file pair