**Topic: Defining C++ classes**

- C++ permits member functions
- Encapsulation includes data hiding
- Members may be public or private
- Access functions
- Initialize members using constructors
- Arrays are safer inside classes
- Copy constructors add power to copying
- Constructors may have member initializers
- Destructors clean up
- `new` and `delete` trigger steps

**Passing a structure to a function**

```cpp
struct employees { char name[40]; int hours; };  
void employee_display(employees e);  
void main()  
{  
   employees emp1 = {"Dale",49}, emp2 = {"Lin",35"};  
   employee_display(emp1);  
   employee_display(emp2);  
}  

void employee_display(employees e)  
{  
   cout << e.name << " worked " << e.hours << " hrs.\n";  
}
```

**Output:**

Dale worked 49 hours.
Lin worked 35 hours.

- Function `employee_display` implements an operation on an instance of struct type `employees`. 
Classes and objects

Class: a structure type defined by data attributes and operations

```cpp
struct employees {
    char name[40];
    int hours;
    void display();
};
```

Object: an instance of a class

```cpp
employees emp1, emp2;
```

Using a class

```cpp
#include <iostream.h>
struct employees {
    char name[40];
    int hours;
    void input();
    void display();
};

void main() {
    employees emp;
    emp.input();
    emp.display();
}
```

Sample I/O:
Enter name, hrs: Musa 40
Musa worked 40 hours.
Member functions

- A call to a member function must name an object, using dot notation.
  
  ```
  employees emp;
  emp.input();
  ```

- Member functions of an object’s class (methods) have access to that object’s members.
  ```
  void input() { cin >> name >> hours; }
  ```

- Member function header includes class name and scope resolution operator (::).
  ```
  void employees::input()
  ```

An implicit parameter to all member functions

- The object that calls a method is an implicit parameter
- It need not be named inside the member function to access object’s members
- The object’s address is always stored in the standard ID this.
- Example: in a member function of class employees, ID is equivalent to this->ID.
Message passing means calling member functions

- **Example:** An employee-roster object might pass a message to an employee object to display itself:
  
  ```cpp
  rosters::display(int subscript).
  {
    employee[subscript].display()
  }
  ```

Encapsulation hides data

- The keyword `class` may replace `struct`.
- With `class`, members are inaccessible to client code unless declared `public`.
  ```cpp
  class employees
  {
    public:
      void input();
      void display();
    private:
      char name[40];
      int hours;
  };
  ```

```cpp
void main()
{
  employees emp;
  emp.hours = 40;
  emp.input();
}
```
Controlled access to data

• Members are public, private, or protected
• Class members default to private (hidden from client)
• Public members are class interface
• Private members and definitions of member functions are implementation
• Client code may reach private data through access functions
• protected members are hidden

Accessors and mutators

• An accessor member function does not change the state of an object
• It is considered best to declare accessors with the qualifier const
• A mutator member function changes the value of member data items.

```cpp
class employees
{
public:
    employees();
    employees(char* nm, int I);
    char* get_name() const;
    void set_name(char* nm);
    void display() const;
private:
```
Constructors may initialize member data

class employees
{
    public:
        employees()
        {
            name[0] = hours = 0;
        }
        employees(char nm[], int hrs)
        {
            strncpy(name, nm, hours = hrs);
        }
        void input();
        void display();

    private:
        char name[40];
        int hours;
};

void main()
{
    employees e;
    e.display();

Constructors

- Execute automatically when an object is created, on stack or heap
- Take name of class: employees::employees()
- Should assign values to all member data
- Have no return value
- May take parameters:
  employees emp("Dass", 43);
- May be overloaded:
  employees();
  employees(char nm[], int hrs);
- A call to a constructor returns an anonymous object:
  strings str1 = strings("Hello");
Using member initializers

- Any data member may have an initializer, in constructor header, to set its value
- Example:
  \[ \text{employees}(\text{int} \ I, \text{int} \ S) : \text{ID}(I) \]
- Member initializers are useful for bringing attention to parameters that are to be assigned to members
- If a class has a \textit{const} member, a member initializer in the constructor must set its value
- Value of a \textit{const} member may not be changed once initialized

Destructors

- A class may have one destructor
- It is automatically called when an object goes out of scope or is deallocated with \textit{delete}
- Name is tilde, followed by class name:
  \[ \text{employees}:\sim \text{employees}(); \]
- Cleans up after object, to close file, deallocate heap memory, etc.
- No parameters, no return value
If the *strings* class used dynamic allocation...

- A string object could use exactly as much memory as needed
- The character-content data member would be a pointer
- A constructor could use *new* to get space for a character array on the heap
- The *strings* destructor should call *delete[]* to free the space occupied by the character contents

When a function returns an object

class widgets
{
    public:
        widgets() { cout << "default constructor" << endl; }
        widgets(int I): ID(I) { cout << "constructor 2" << endl; }
    ~widgets() { cout << "destructor" << endl; }
    private:
        int ID;
    }

    widgets buy_widget()
    {
        static qty = 0;
        qty++;
        cout << "Buying widget #" << qty << endl;
        widgets w(qty);
        return w;
    }

    void main()
    {
        widgets my_widg;
        my_widg = buy_widget();
    }

Output:
default
    constructor
Buying widget #1
constructor 2
    destructor
destructor
Copy and conversion constructors

Two ways to use copy constructor:

```cpp
#include "strings.h"
#include "strings.cpp"

void main()
{
    strings string1("string1"),
    string2 = "string2";
}
```

Definitions of `strings` class and copy constructor:

```cpp
class strings
{
    public:
        strings(); // Conversion constructor
            // from <char*>
    private:
        enum{STR_MAX = 256};
        char content[STR_MAX];
}
```

Deep copy vs. shallow copy

- When an object is copied member by member (shallow copy), the result may be two pointers each pointing to the same third data item

  - `emp1`
    ```
    | ID | city |
    |----|------|
    | 5  | Boston
    ```

  - `emp2`
    ```
    | ID | city |
    |----|------|
    | 5  | Boston
    ```

  - This means trouble; e.g., changing `emp1`’s city to “Chicago” also changes `emp2`’s city

- Solution: deep copy, not copying pointer members but rather what they point to

  - `emp1`
    ```
    | ID | city |
    |----|------|
    | 5  | Boston
    ```

  - `emp2`
    ```
    | ID | city |
    |----|------|
    | 5  | Boston
    ```
Copy constructors may use deep copy

```cpp
class employees {
public:
    employees(int I, char* C) : ID(I) {
        strcpy(city = new char[strlen(C) + 1], C);
    }
    employees(employees& e) {
        ID = e.ID;
        city = new char[strlen(e.city)+1], e.city;
        strcpy(city, e.city);
    }
    void dump_city_addr() {
        cout << "Address of city: " << hex << (long)city;
    }
private:
    int ID;
    char* city;
};

void main() {
    employees emp1(1234, "NYC");
    employees emp2 = emp1;
    emp1.dump_city_addr();
    emp2.dump_city_addr();
}
```

Output:
- Address of city: 76025c
- Address of city: 760e1c

A collection of integers

```cpp
class score_lists {
enum {MAX_SCORES = 6};
public:
    score_lists() { size = 0; }
    bool append(int new_element);
    void input();
    void display();
private:
    int element[MAX_SCORES];
    int size;
};

void main() {
    score_lists list;
    list.append(100);
    list.input();
    list.display();
}
```

Constant member
- Inline constructor
- Range from 0 to MAX_SCORES
Preventing array boundary errors

```cpp
bool score_lists::append(int new_element)
// Adds <new_element> to collection.
{
    if (size < MAX_SCORES)
    {  element[size++] = new_element;
        return TRUE;
    }
    else
    {  cout << "score_lists::append: "
        "collection already full" << endl;
        return FALSE;
    }
}
• If this append is used to build a collection, out-of-bounds errors will not occur
```

Constant members and member initializers

```cpp
class employees
{
    const int ID;
    int salary;
public:
    employees(int I,int S) : ID(I)
    { salary = S; }
    void display()
    { cout << "ID=" << ID << " Salary=" << salary << endl; }
};

void main()
{
    employees e(4321,45000);
    e.display();
}

Output: 
ID=4321 Salary=45000
```

Member initializer
sets value of ID
Value passed to member initializer
[constmem.cpp]
Static members

• All instances of a class share the same value for a static member of the class
• `class runners`{
  public:
    runners();
  private:
    static double world_record;
    double latest,best;
};
• Initialize like global variable, e.g.:
  `runners::world_record = 100000;`
• Static member exists independent of class instances

Advantages of using classes

• Code is natural and close to problem
• Interface with client is separated from internal implementation; good for maintenance
• Functions that use related data have automatic access to it
• Programmer may automate initialization and validation of data