MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

03 C++ Classes

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

January 13, 2022

Administration

- I will publish a poll after the class for the mid-term1's date

Dynamic Memory Allocation (struct in C)

- Operator new dynamically allocates storage for an object.
- Access members using:
  - pointer name**->**member
  - (*pointer name)**.**member

```
enum MealType { NO_PREF, REGULAR, LOW_FAT, VEGETARIAN };

struct Passenger {
    string name;
    MealType mealPref;
    bool isFreqFlyer;
    string freqFlyerNo;
};
```

```
Passenger *p;
p = new Passenger;          // p points to the new Passenger
p->name = "Pocahontas";     // set the structure members (access using ->)
cout << p->name << endl;    // outputs Pocahontas
(*p).name = "John";         // access using .
cout << p->name << endl;    //outputs John
cout << (*p).name << endl;  // outputs John

delete p;
```

- Arrays can be allocated using **new** and must be deleted as well

```
char* buffer = new char[500];
buffer[3] = 'a';
delete [] buffer;
```

- If an object is allocated with new, it should eventually be deallocated with delete. Otherwise: Memory Leak!

A Simple C++ Class

- A Counter Class
- Declaration vs definition
  - scoping operator ::
- Constructor: initializes member variables
- getter
- access specifier (interfaces)
  - public
  - private (default)
    - access is limited
  - [protected]

```
class Counter {  // a simple counter
        public:
            Counter();  // initialization
            int getCount();  // get the current count
            void increaseBy(int x);  // add x to the count
        private:
            int count;  // the counter's value
    };

    Counter::Counter() { count = 0; }  //constructor
    int Counter::getCount() { return count; }   // get current count
    void Counter::increaseBy(int x) { count += x; }  // add x to the count
```

```
Counter ctr;  // an instance of Counter
cout << ctr.getCount() << endl;  // prints the initial value (0)
ctr.increaseBy(3);  // increase by 3
cout << ctr.getCount() << endl;  // prints 3
ctr.increaseBy(5);  // increase by 5
cout << ctr.getCount() << endl;  // prints 8
cout << ctr.count << endl;   // ILLEGAL - count is private!!!
```

Passenger Class

- Simple Passenger Class
- Declaration vs definition
- access or modify private member

is not allowed from outside.

```
class Passenger {
    public:
        Passenger();
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}
```

```
Passenger pass; // pass is a Passenger
// ...
if ( !pass.isFrequentFlyer() ) {       // not already a frequent flyer?
    pass.makeFrequentFlyer("392953");  // set pass's freq flyer number
}
pass.name = "Joe Blow";   // ILLEGAL! name is private
```

Passenger Class

- Simple Passenger Class
- In-line definition!

```
class Passenger {
    public:
        Passenger();
        //bool isFrequentFlyer() const;
        bool isFrequentFlyer() const { return isFreqFlyer; }
        void makeFrequentFlyer(const string& newFreqFlyerNo);

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
bool Passenger::isFrequentFlyer() const {
    return isFreqFlyer;
}

void Passenger::makeFrequentFlyer(const string& newFreqFlyerNo) {
    isFreqFlyer = true;
    freqFlyerNo = newFreqFlyerNo;
}
```

```
Passenger pass; // pass is a Passenger
// ...
if ( !pass.isFrequentFlyer() ) {       // not already a frequent flyer?
    pass.makeFrequentFlyer("392953");  // set pass's freq flyer number
}
pass.name = "Joe Blow";   // ILLEGAL! name is private
```

Constructors

- Simple Passenger Class
- Constructors:
  - default
  - copy
- default argument

```
class Passenger {
    public:
        Passenger();  //default constructor
        Passenger(const string& nm, MealType mp, const string& ffn = "NONE");
        Passenger(const Passenger& pass); // copy constructor
        bool isFrequentFlyer() const;
        void makeFrequentFlyer(const string& newFreqFlyerNo);

    private:
        string name;
        MealType mealPref;
        bool isFreqFlyer;
        string freqFlyerNo;
};
```

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}
```

Passenger Class

- Simple Passenger Class
- Constructors:
  - default ()
  - copy (=)

```
Passenger::Passenger() {  // default constructor
    name = "--NO NAME--";
    mealPref = NO_PREF;
    isFreqFlyer = false;
    freqFlyerNo = "NONE";
}

Passenger::Passenger(const string& nm, MealType mp, const string& ffn) {
    name = nm;
    mealPref = mp;
    isFreqFlyer = (ffn != "NONE");  // true only if ffn given
    freqFlyerNo = ffn;
}

Passenger::Passenger(const Passenger& pass) {
    name = pass.name;
    mealPref = pass.mealPref;
    isFreqFlyer = pass.isFreqFlyer;
    freqFlyerNo = pass.freqFlyerNo;
}
```

```
Passenger p1; // default constructor
Passenger p2("John Smith", VEGETARIAN, 293145); // 2nd constructor
Passenger p3("Pocahontas", REGULAR);  // not a frequent flyer
Passenger p4(p3);  //copied from p3
Passenger p5 = p2;   //copied from p2
Passenger* pp1 = new Passenger;  //default constructor
Passenger* pp2 = new Passenger("JoeBlow", NO_PREF); // 2nd constr.
Passenger pa[20];  //default constructor
```

Vect Class

- Simple Vector Class
- Destructor:
  - needed when class allocates memory dynamically!
  - define using ~classname
- Avoid shallow copy
  - copy constructor

```
Vect a(100);  // a is a vector of size 100
Vect b = a;   // initialize b from a (DANGER!)
Vect c;       // c is a vector (default size 10)
c = a;        // assign a to c (DANGER!)
```

```
class Vect {
    public:
        Vect(int n);
        ~Vect();
// ... other public members omitted
    private:
        int* data; int size;
};

Vect::Vect(int n) {
    size = n;
    data = new int[n];
}
Vect::~Vect() {
    delete [] data;
}
```

```
Vect::Vect(const Vect& a) { // copy constructor from a
    size = a.size;  // copy sizes
    data = new int[size];  // allocate new array
    for (int i = 0; i < size; i++) {  // copy the vector contents
        data[i] = a.data[i];
    }
}

Vect& Vect::operator=(const Vect& a) {  // assignment operator from a
    if (this != &a) {  // avoid self-assignment
        delete [] data;  // delete old array
        size = a.size;   // set new size
        data = new int[size];  // allocate new array
        for (int i=0; i < size; i++) {  // copy the vector contents
            data[i] = a.data[i];
        }
    }
    return *this;
}
```

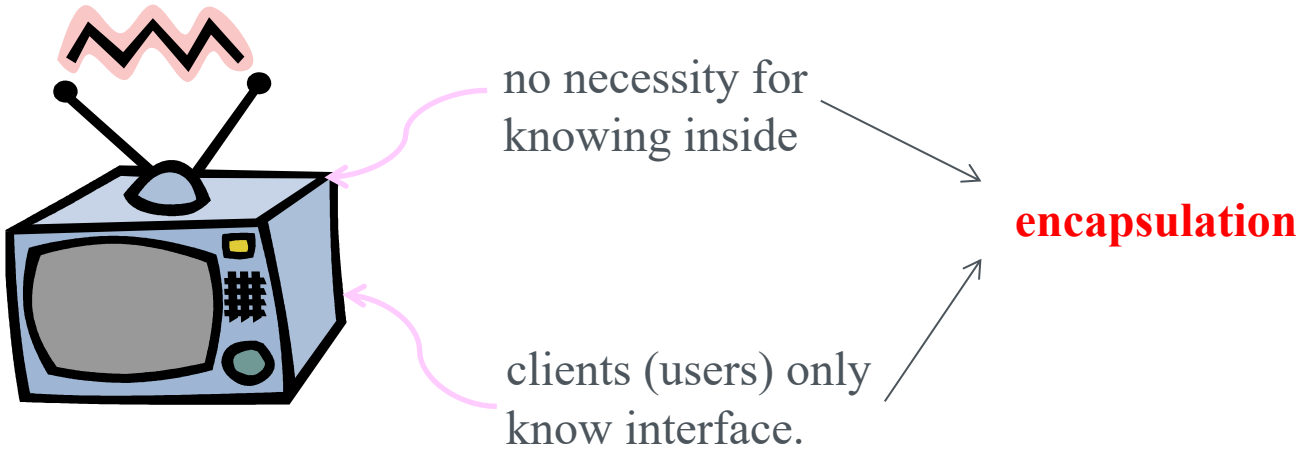
Classes with Dynamically-Allocated Members

- Every class that allocates its own objects using **new** should:
  - Define a **destructor** to free any allocated objects.
  - Define a **copy constructor**, which allocates its own new member storage and copies the contents of member variables.
  - Define an **assignment operator**, which deallocates old storage, allocates new storage, and copies all member variables.

```
Vect a(100);  // a is a vector of size 100
Vect b = a;   // initialize b from a (DANGER!)
Vect c;       // c is a vector (default size 10)
c = a;        // assign a to c (DANGER!)
```

Encapsulation

- **Encapsulation** conceals the functional details defined in a class from external world (clients).
  - Information hiding
    - By limiting access to member variables/functions from outside
  - Operation through **interface**
    - Allows access to member variables through interface
  - Separation of **interface from implementation**
    - declaration vs definition



Questions?