Lecture 14: Object Oriented Programming and Classes
About These Slides

• Based on Chapter 18 of the book
  *An Introduction to Programming Through C++*
  by Abhiram Ranade *(Tata McGraw Hill, 2014)*

• Original slides by Abhiram Ranade
  – First update by Varsha Apte
  – Second update by Uday Khedker
Main Recommendations From The Previous Chapter

• Define a struct to hold information related to each entity that your program deals with

• Define member functions corresponding to actions/operations associated with the entity
Outline

- Constructors
- Copy Constructors
- Destructors
- Operator overloading
- Overloading the assignment operator
- Access control
- Classes
- Graphics and input/output classes
Motivational Example: The Queue Struct in Taxi Dispatch

```c
const int N=100;

struct queue{
    int elements[N], nwaiting, front;
    bool insert(int v){
        ...
    }
    bool remove(int &v){
        ...
    }
};
```

- Once the queue is created, we expect it to be used only through the member functions, insert and remove.
- We do not expect elements, nWaiting, front to be directly accessed.
Main Program Using Queue

```c
int main()
{
    Queue q;
    q.front = q.nWaiting = 0;
    while(true)
    {
        char c; cin >> c;
        if(c == 'd')
        {
            int driver; cin >> driver;
            if(!q.insert(driver))
                cout << "Q is full\n";
        }
        else if(c == 'c')
        {
            int driver;
            if(!q.remove(driver))
                cout << "No taxi.\n";
            else cout << "Assigning <<
                driver<< endl;
        }
    }
}
```

- Main program does use q through operations insert and remove
- However, at the beginning, q.front and q.nWaiting is directly manipulated
- This is against the philosophy of software packaging
- When we create a queue, we will always set q.nWaiting and q.front to 0
- C++ provides a way by which the initialization can be made to happen automatically, and also such that programs using Queue do not need to access the data members directly
- Just defining Queue q; would by itself set q.nWaiting and q.front to 0!
  - Next
In C++, the programmer may define a special member function called a constructor which will always be called when an instance of the struct is created.

A constructor has the same name as the struct, and no return type.

The code inside the constructor can perform initializations of members.

When q is created in the main program, the constructor is called automatically.

```cpp
struct Queue{
  int elements[N], front, nWaiting;
  Queue(){ // constructor
    nWaiting = 0;
    front = 0;
  }
  // other member functions
};

int main(){
  Queue q;
  // no need to set
  // q.nWaiting, q.front
  // to 0.
}
```
Constructors In General

- Constructor can take arguments
- The creation of the object `a` in `main` can be thought of as happening in two steps
  - Memory is allocated for `a` in `main`
  - The constructor is called on `a` with the given arguments
- You can have many constructors, provided they have different signatures
Another example: Constructor for V3

```c
struct V3{
    double x,y,z;
    V3(){
        x = y = z = 0;
    }
    V3(double a){
        x = y = z = a;
    }
};
int main();
V3 v1(5), v2;
```

- When defining `v1`, an argument is given
- So the constructor taking a single argument is called. Thus each component of `v1` is set to 5
- When defining `v2`, no argument is given. So the constructor taking no arguments gets called. Thus each component of `v2` is set to 0
Remarks

• If and only if you do not define a constructor, will C++ defines a constructor for you which takes no arguments, and does nothing
  – If you define a constructor taking arguments, you implicitly tell C++ that you want programmers to give arguments. So if some programmer does not give arguments, C++ will flag it as an error
  – If you want both kinds of initialization, define both kinds of constructor
• A constructor that does not take arguments (defined by you or by C++) is called a default constructor
• If you define an array of struct, each element is initialized using the default constructor
The Copy Constructor

• Suppose an object is passed by value to a function
  – It must be copied to the variable denoted by the parameter
• Suppose an object is returned by a function
  – The value returned must be copied to a temporary variable in the calling program
• By default the copying operations are implemented by copying each member of one object to the corresponding member of the other object
  – You can change this default behaviour by defining a copy constructor
struct Queue{
    int elements[N], nWaiting, front;
    Queue(const Queue &source){ // Copy constructor
        front = source.front;
        nWaiting = source.nWaiting;
        for(int i=front, j=0; j<nWaiting; j++){
            elements[i] = source.elements[i];
            i = (i+1) % N;
        }
    }
};
Copy Constructor in the Example

• The copy constructor must take a single reference argument: the object which is to be copied
• Note that the argument to the copy constructor must be a reference, otherwise the copy constructor will have to be called to copy the argument! This is will result in an unending recursion
• Member elements is not copied fully. Only the useful part of it is copied
  – More efficient
• More interesting use later
Destructors

• When control goes out of a block in which a variable is defined, that variable is destroyed
  – Memory allocated for that variable is reclaimed

• You may define a destructor function, which will get executed before the memory is reclaimed
Destructor Example

• If a queue that you have defined goes out of scope, it will be destroyed
• If the queue contains elements at the time of destruction, it is likely an error
• So you may want to print a message warning the user
• It is usually an error to call the destructor explicitly. It will be called automatically when an object is to be destroyed. It should not get called twice.
• More interesting uses of the destructor will be considered in later chapters.
Destructor Example

struct Queue{
    int elements[N], nWaiting, front;
    ...
    ~Queue(){      //Destructor
        if(nWaiting>0) cout << "Warning:"
                       << " non-empty queue being destroyed."
                       << endl;
    }
};
Operator Overloading

• In Mathematics, arithmetic operators are used with numbers, but also other objects such as vectors
• Something like this is also possible in C++!
• An expression such as $x @ y$ where $@$ is any “infix” operator is considered by C++ to be equivalent to $x.\text{operator@}(y)$ in which $\text{operator@}$ is a member function
• If the member function $\text{operator@}$ is defined, then that is called to execute $x @ y$
Example: Arithmetic on V3 objects

struct V3{
    double x, y, z;
    V3(double a, double b, double c){
        x=a; y=b; z=c;
    }
    V3 operator+(V3 b){ // adding two V3s
        return V3(x+b.x, y+b.y, z+b.z); // constructor call
    }
    V3 operator*(double f){ // multiplying a V3 by f
        return V3(x*f, y*f, z*f); // constructor call
    }
};
int main(){
  V3 u(1,2,3), a(4,5,6), s;
  double t=10;
  s = u*t + a*t*t*0.5;
  cout << s.x << ' ' << s.y << ' ' << s.z << endl;
}

Using V3 Arithmetic
Remarks

- Expression involving vectors can be made to look very much like what you studied in Physics
- Other operators can also be overloaded, including unary operators (see the book)
- Overload operators only if they have a natural interpretation for the struct in question
- Otherwise you will confuse the reader of your program
The this pointer

• So far, we have not provided a way to refer to the receiver itself inside the definition of a member function.
• Within the body of a member function, the keyword this points to the receiver i.e. the struct on which the member function has been invoked.
• Trivial use: write this->member instead of member directly.

```c
struct V3{
    double x, y, z;
    double length(){
        return sqrt(*this.x * *this.x
                     + *this.y * *this.y
                     + *this.z * *this.z);
    }
}
```
• More interesting use later.
**Overloading The Assignment Operator**

• Normally if you assign one struct to another, each member of the rhs is copied to the corresponding member of the lhs

• You can change this behaviour by defining member function `operator=` for the struct

• A return type must be defined if you wish to allow chained assignments, i.e. \( v_1 = v_2 = v_3 \); which means \( v_1 = (v_2 = v_3) \);
  – The operation must return a reference to the left hand side object
Example

struct Queue{

  ...

  Queue & operator=(Queue &rhs){
    front = rhs.front;
    nWaiting = rhs.nWaiting;
    for(int i=0; i<nWaiting; i++){
      elements[i] = rhs.elements[i];
      i = (i+1) % N;
    }
    return *this;
  }

};

// only the relevant elements are copied
Access Control

- It is possible to restrict access to members or member functions of a struct.
- Members declared public: no restriction.
- Members declared private: Can be accessed only inside the definition of the struct.
- Typical strategy: Declare all data members to be private, and some subset of function members to be public.
Access Control Example

```cpp
struct Queue{
private:
  int elements[N], nWaiting, front;
public:
  Queue(){ ... }  // Constructor
  bool insert(int v){  // Insert function
    ..
  }
  bool remove(int &v){  // Remove function
    ..
  }
};
```
Remarks

- **public**, **private**: access specifiers
- An access specifier applies to all members defined following it, until another specifier is given
- Thus elements, nWaiting, front are private, while Queue(), insert, remove are public
Remarks

• The default versions of the constructor, copy constructor, destructor, assignment operator are public

• If you specify any of these as private, then they cannot be invoked outside of the struct definition

• Thus if you make the copy constructor of a struct X private, then you will get an error if you try to pass a struct of type X by value

• Thus, as a designer of a struct, you can exercise great control over how the struct gets used
Classes

• A class is essentially the same as a struct, except:
  – Any members/member functions in a struct are public by default
  – Any members/member functions in a class are private by default
Classes

• Example: a Queue class:

class Queue{
    int elements[N], nWaiting, front;
public:
    Queue(){...}
    bool remove(int &v){...}
    bool insert(int v){...}
};

• Members elements, nWaiting and front will be private.
Example

```cpp
struct V3{
  double x, y, z;
  V3(double v) {
    x = y = z = v;
  }
  double X() {
    return x;
  }
};
```

```cpp
struct V3{
  double x, y, z;
  V3(double v) {
    double X();
  }
  //implementations
  V3::V3(double v) {
    double X();
    x = y = z = v;
  }
  double V3::X() {
    return x;
  }
};
```
### Input Output Classes

- **cin, cout**: objects of class `istream, ostream` resp. predefined in C++
- **<<, >>**: operators defined for the objects of these classes
- **ifstream**: another class like `istream`
  - You create an object of class `ifstream` and associate it with a file on your computer
  - Now you can read from that file by invoking the `>>` operator!
- **ofstream**: a class like `ostream`, to be used for writing to files
  - Must include header file `<fstream>` to use `ifstream` and `ofstream`
Example of file i/o

#include <fstream>
#include <simplecpp>

int main() {
    ifstream infile("f1.txt");
    // constructor call. object infile is created and associated
    // with f1.txt, which must be present in the current directory
    ofstream outfile("f2.txt");
    // constructor call. Object outfile is created and associated
    // with f2.txt, which will get created in the current directory
Example of file i/o

repeat(10){
    int v;
    infile >> v;
    outfile << v;
}

// f1.txt must begin with 10 numbers. These will be read and
// written to file f2.txt

}
Concluding Remarks

• The notion of a packaged software component is important.
• Making data members private: hiding the implementation from the user
• Making some member functions public: providing an interface using which the object can be used
• Separation of the concerns of the developer and the user
• Idea similar to what we discussed in connection with ordinary functions
  – The specification of the function must be clearly written down (analogous to interface)
  – The user should not worry about how the function does its work (analogous to hiding data members)