An Introduction to Programming

through C++

with

Manoj Prabhakaran

Lecture 19

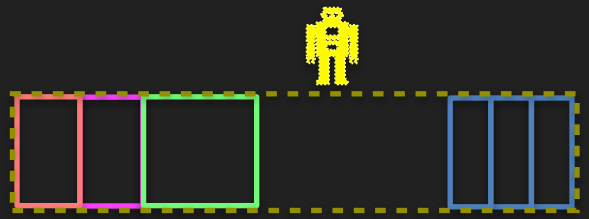
Classes

Glimpses of Object-Oriented Programming

Based on material developed by Prof. Abhiram G. Ranade

Object-Oriented Programming

- Recall our model of program execution
  - A single agent (processor) executing instructions and reading/updating memory



- But we may think of different functions as different agents, one calling up another and waiting for a response
  - Static variables allow each of these agents to have memory (state) across calls
  - But only one such state per function

- In OOP, we think in terms of dynamically created agents (possibly many of the same kind) with their own states, interacting with each other

Structs a.k.a. Classes

- In C, *objects* that carry complex data have a `struct` type
- C++ provides more support for "Object-Oriented Programming"
  - Objects not only carry data, but also "know" how to compute on that data
    - Note: The "box" associated with an object stores its data
  - The type of such an object is traditionally called its *class*
- In C++, keywords `struct` and `class` both refer to the same idea
  - Going forward, we'll mostly use `struct` for a `class` without functions

Structs a.k.a. Classes

```
struct node {
  int val;
  node* next = nullptr;
};
```

```
struct queue {
  node* head = nullptr;
  node* tail = nullptr;
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
};
```

These members are "private."

Visible in member functions of the same class, but nowhere else in the program.

Encapsulation:

Keep implementation details private. Expose only the interface publicly.

```
struct node {
  int val;
  node* next = nullptr;
};
```

```
class queue {
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
};
```

Structs a.k.a. Classes

```
struct node {
  int val;
  node* next = nullptr;
};
```

```
struct queue {
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
private:
  node* head = nullptr;
  node* tail = nullptr;
};
```

These members are "private."

Visible in member functions of the same class, but nowhere else in the program.

Can do the same in a struct too. But the default is public.

```
struct node {
  int val;
  node* next = nullptr;
};
```

```
class queue {
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
};
```

Classes

- Coming up: Some features of classes to support OOP
  - Encapsulation features
  - Constructors and destructors
  - Operator overloading
- Will not cover some features which are important to OOP
  - Inheritance: "Deriving" a class from a "Base class"
    - And many features supporting inheritance

More Encapsulation

- Keeping implementation details inaccessible outside of the class ensures that the entire class can be reimplemented without affecting code that uses objects of the class

Private members are not visible outside of the class

Classes/structs can be nested

As the struct node is an implementation detail of the class queue, its definition can be moved within the class queue

(Note that the class queue doesn't have any member of the type node.)

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
};
```

Destructor

```
void queue_demo() {
  queue Q;
  for(int i=0; i < 10000000; i++)
    Q.enqueue(1);
  Q.clear();
}
```

Recall: Even though `Q` is a local variable that is automatically destroyed, if `clear()` not called here, memory leak!

Can define a **destructor** function in the class, which will be automatically invoked the object's life ends (when, e.g., goes out of scope or delete called).

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
};
```

Destructor

```
void queue_demo() {
  queue Q;
  for(int i=0; i < 10000000; i++)
    Q.enqueue(1);
  Q.clear();
}
```

`Q.~queue()` is called here.

Can define a **destructor** function in the class, which will be automatically invoked the object's life ends (when, e.g., goes out of scope or delete called).

Special syntax: *~classname()*. No return type.

Typically public.

After the destructor code finishes, destructor of each member (if present) is invoked.

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
  ~queue() { clear(); }
};
```

Constructors

Can also define one or more **constructor** functions in the class, which will be automatically invoked when the object's life begins (when, e.g., comes into scope or new is called).

Special syntax: *classname*(*arguments*)
No return type. Typically public.
Can be overloaded.

Before the constructor is executed, all the members are initialised (using their constructors), in the order in which they appear in the class

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
  ~queue() { clear(); }
  queue() {}
  queue(int v) {enqueue(v);}
};
```

Constructors

```
// these call the constructor without args
queue Q1;
queue* q1 = new queue;

// these call the one with an int arg.
queue Q2(3);
queue* q2 = new queue(4);
```

OK to have no constructor. If so, compiler implicitly adds a "default" constructor (with empty arguments and empty body).

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
  ~queue() { clear(); }
  queue() {}
  queue(int v) {enqueue(v);}
};
```

Member Initialisation

- A constructor can specify how to initialise the members before its own code is invoked
  - By providing the inputs to their (default or user-defined) constructors
  - Otherwise, their default initialisation (if any) will be used
    - Otherwise, zero-initialised

```
class box {
  const int h = 100, w = 100;
public:
  box(int a) : h(a) {
    // when this code starts
    // h, w already inited
    // h==a and w==100
  }
};
```

Before the constructor is executed, all the members are initialised (using their constructors), in the order in which they appear in the class

Copy Constructor

- Often a new object needs to be constructed by copying an existing object
  - E.g., passing arguments by value, or returning a value
- We can explicitly specify how it should work
  - E.g., a "deep copy" for queue
- If none specified, the compiler adds a default copy constructor which copies all members (using their copy constructors)

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
  ~queue() { clear(); }
  queue() {}
  queue(int v) {enqueue(v);}
  queue(const queue&);
};
```

Copy Constructor

Deep copy

The default copy constructor would have just copied head, tail

```
queue::queue(const queue& q) {
  for(node* n = q.head; n; n = n->next)
    enqueue(n->val);
}
```

Being a function in the class, all members (including the private members) are visible here.

So, can access the private members of other objects of the same class.

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
  ~queue() { clear(); }
  queue() {}
  queue(int v) {enqueue(v);}
  queue(const queue&);
};
```

Copy Constructor

```
queue::queue(const queue& q) {
  for(node* n = q.head; n; n = n->next)
    enqueue(n->val);
}
```

```
queue Q1;
...
// invoking the copy constructor
queue Q2 = Q1; // not an assignment
queue* q = new queue(Q2);
```

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
public:
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
  ~queue() { clear(); }
  queue() {}
  queue(int v) {enqueue(v);}
  queue(const queue&);
};
```

Operators

```
int main() {
  queue Q1, Q2;             // two empty queues
  Q1 << 1 << 2 << 3;        // enqueue 1, 2, 3 into Q1
  Q2 = Q1;                  // copy (assign) contents of Q1 to Q2
  int x, y;
  Q2 >> x >> y;             // dequeue into x and then into y
  Q2 += Q1;                 // append contents of Q to the end of Q.
  cout << "Q1: " << Q1 << " Q2: " << Q2 << endl; // print the queues
}
```

- How do we implement such operators?
  - Operators are in fact functions which take their operands as inputs
  - E.g., `a << b` will invoke either `a.operator<<(b)` or `operator<<(a,b)`, whichever is available

Operators

```
queue::queue& operator<< (int v) {
  enqueue(v);
  return *this;
}

queue::queue& operator>> (int& v) {
  dq_err = dequeue(v);
  return *this;
}
```

```
class queue {
  struct node {
    int val;
    node* next = nullptr;
  };
  node* head = nullptr;
  node* tail = nullptr;
  bool dq_err = false;
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
public:
  ~queue() { clear(); }
  queue() {}
  queue(int v) {enqueue(v);}
  queue(const queue&);
  queue& operator<< (int v);
  queue& operator>> (int& v);
  bool fail() { return dq_err;}
};
```

Operators

```
queue& queue::operator+= (const queue& q) {
  queue tmp = q;  // make a copy of q
  // attach the new nodes at our tail
  ( tail ? tail->next : head ) = tmp.head;
  if(tmp.tail) tail = tmp.tail;
  tmp.head = tmp.tail = nullptr;
  return *this;
}
```

Without this, when tmp goes out of scope and its destructor is called, the newly created nodes will be deleted

```
class queue {
  struct node;
  node* head = nullptr;
  node* tail = nullptr;
  bool dq_err = false;
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
public:
  ~queue() { clear(); }
  queue() {}
  queue(int v) {enqueue(v);}
  queue(const queue&);
  queue& operator<< (int v);
  queue& operator>> (int& v);
  bool fail() { return dq_err;}
  queue& operator+= (const queue&);
};
```

Operators

```
cout << "Q1: " << Q1 << endl;
```

We need this operator<< to take an `ostream` object as its first operand. But we can't modify the `ostream` class!

Will be a function outside either class (takes 2 operands). But it needs to access private members of queue. Will declare it as a `friend` in queue.

```
ostream& operator<< (ostream& out, const queue& q) {
  for(node* p = q.head; p; p = p->next)
    out << p->val << (p->next?" ":"");
  return out;
}
```

```
class queue {
  struct node;
  node* head = nullptr;
  node* tail = nullptr;
  bool dq_err = false;
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
public:
  ~queue() { clear(); }
  queue() {}
  queue(int v) {enqueue(v);}
  queue(const queue&);
  queue& operator<< (int v);
  queue& operator>> (int& v);
  bool fail() { return dq_err;}
  queue& operator+= (const queue&);
  friend ostream& operator<<
          (ostream&, const queue&);
};
```

Operators

```
(Q? cout << Q : cout << "empty") << endl;
```

Cast operators can also be defined. Return type is implicit (should be the same as what the object is cast into)

```
class queue {
  struct node;
  node* head = nullptr;
  node* tail = nullptr;
  bool dq_err = false;
  void enqueue(int v);
  bool dequeue(int& v);
  void clear();
public:
  ~queue() { clear(); }
  queue() {}
  queue(int v) {enqueue(v);}
  queue(const queue&);
  queue& operator<< (int v);
  queue& operator>> (int& v);
  bool fail() { return dq_err;}
  queue& operator+= (const queue&);
  friend ostream& operator<<
          (ostream&, const queue&);
  operator bool() { return head; }
};
```

Demo