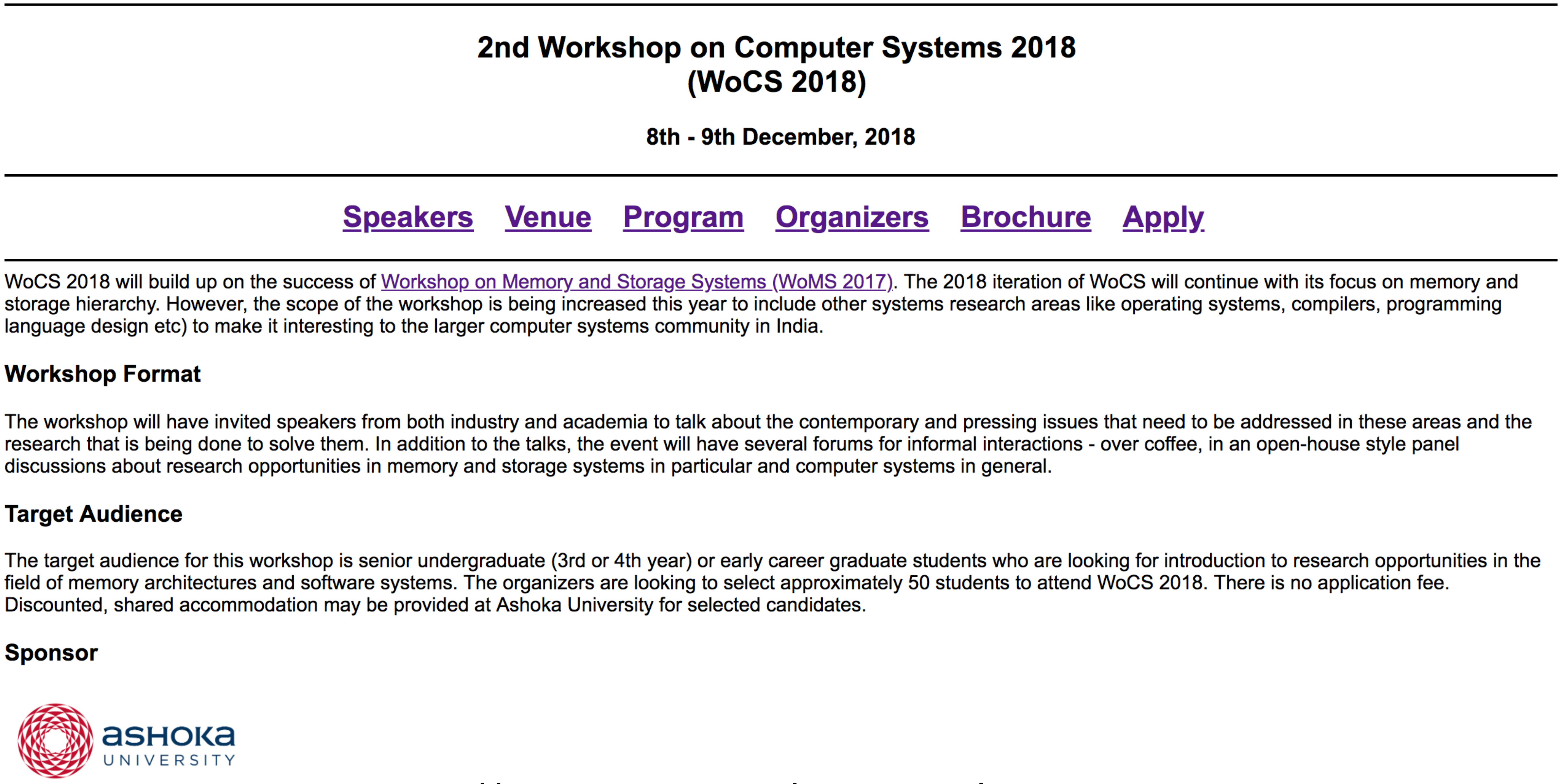
CS 206

Lecture 19 – Object Oriented Programming



2nd Workshop on Computer Systems 2018 (WoCS 2018)

8th - 9th December, 2018

Speakers Venue Program Organizers Brochure Apply

WoCS 2018 will build up on the success of Workshop on Memory and Storage Systems (WoMS 2017). The 2018 iteration of WoCS will continue with its focus on memory and storage hierarchy. However, the scope of the workshop is being increased this year to include other systems research areas like operating systems, compilers, programming language design etc) to make it interesting to the larger computer systems community in India.

Workshop Format

The workshop will have invited speakers from both industry and academia to talk about the contemporary and pressing issues that need to be addressed in these areas and the research that is being done to solve them. In addition to the talks, the event will have several forums for informal interactions - over coffee, in an open-house style panel discussions about research opportunities in memory and storage systems in particular and computer systems in general.

Target Audience

The target audience for this workshop is senior undergraduate (3rd or 4th year) or early career graduate students who are looking for introduction to research opportunities in the field of memory architectures and software systems. The organizers are looking to select approximately 50 students to attend WoCS 2018. There is no application fee. Discounted, shared accommodation may be provided at Ashoka University for selected candidates.

Sponsor

ashoka UNIVERSITY

http://cs.ashoka.edu.in/wocs2018/

Why Object Oriented Programming?

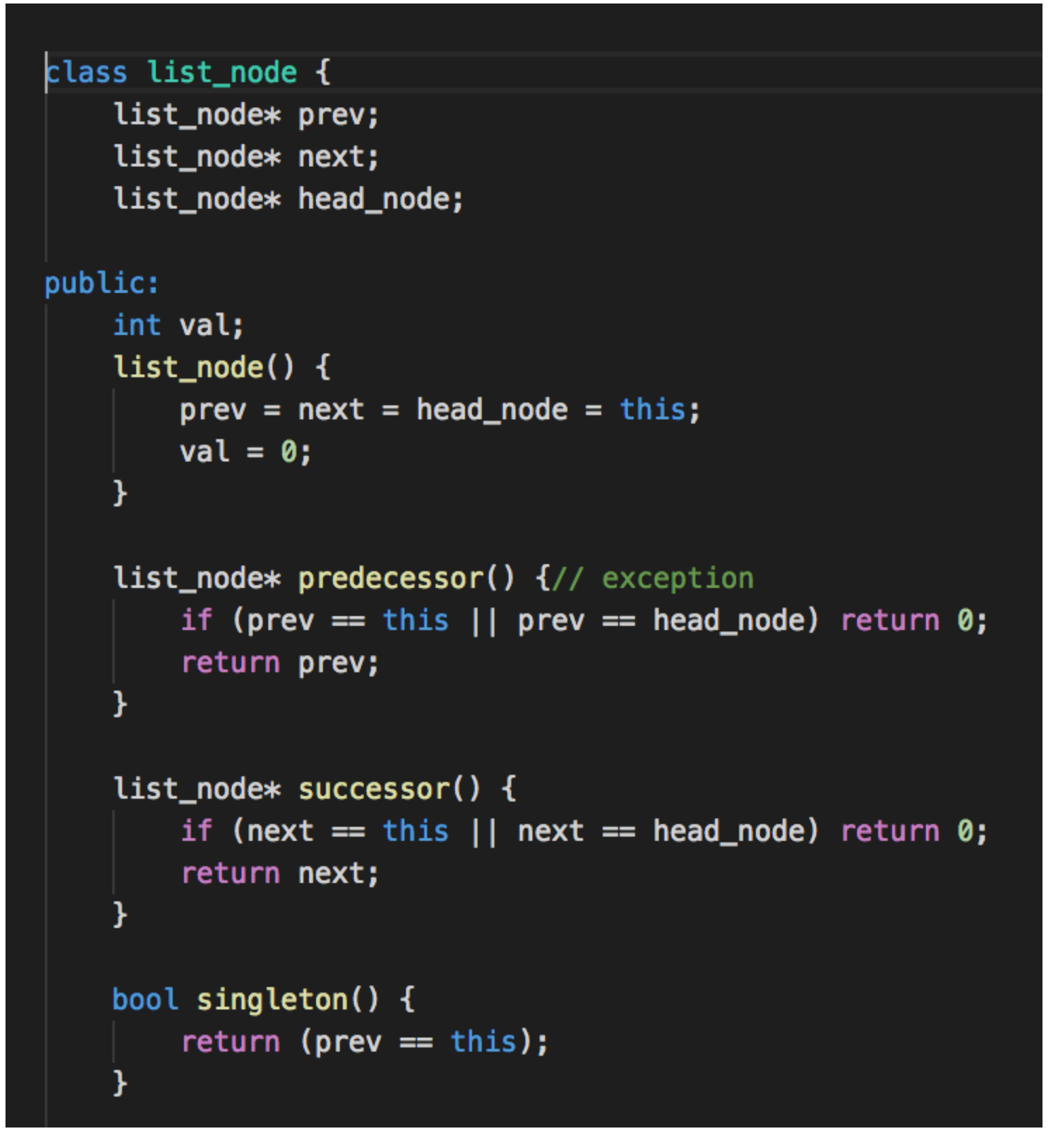
- Conceptual load reduction
  - Minimizes the amount of detail that the programmer must think about at one time
- Fault containment
  - preventing the programmer from using a program component in inappropriate ways
- Code modularity and programmer efficiency
  - easier to assign their construction to separate individuals
  - enhance opportunities for code reuse

Members and Methods

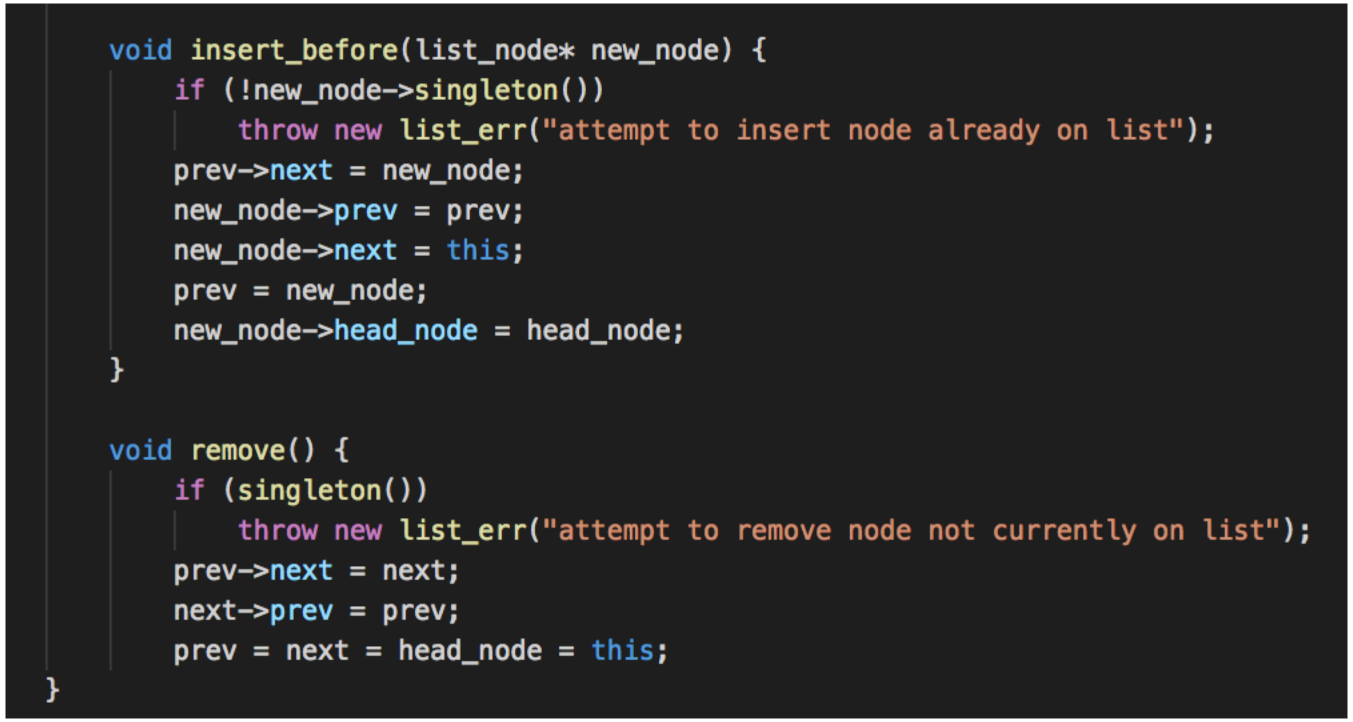
- Members
  - Can be data or subroutines
  - data members are called *fields*

- Methods
  - Subroutine members are also called methods in OO world

List Node

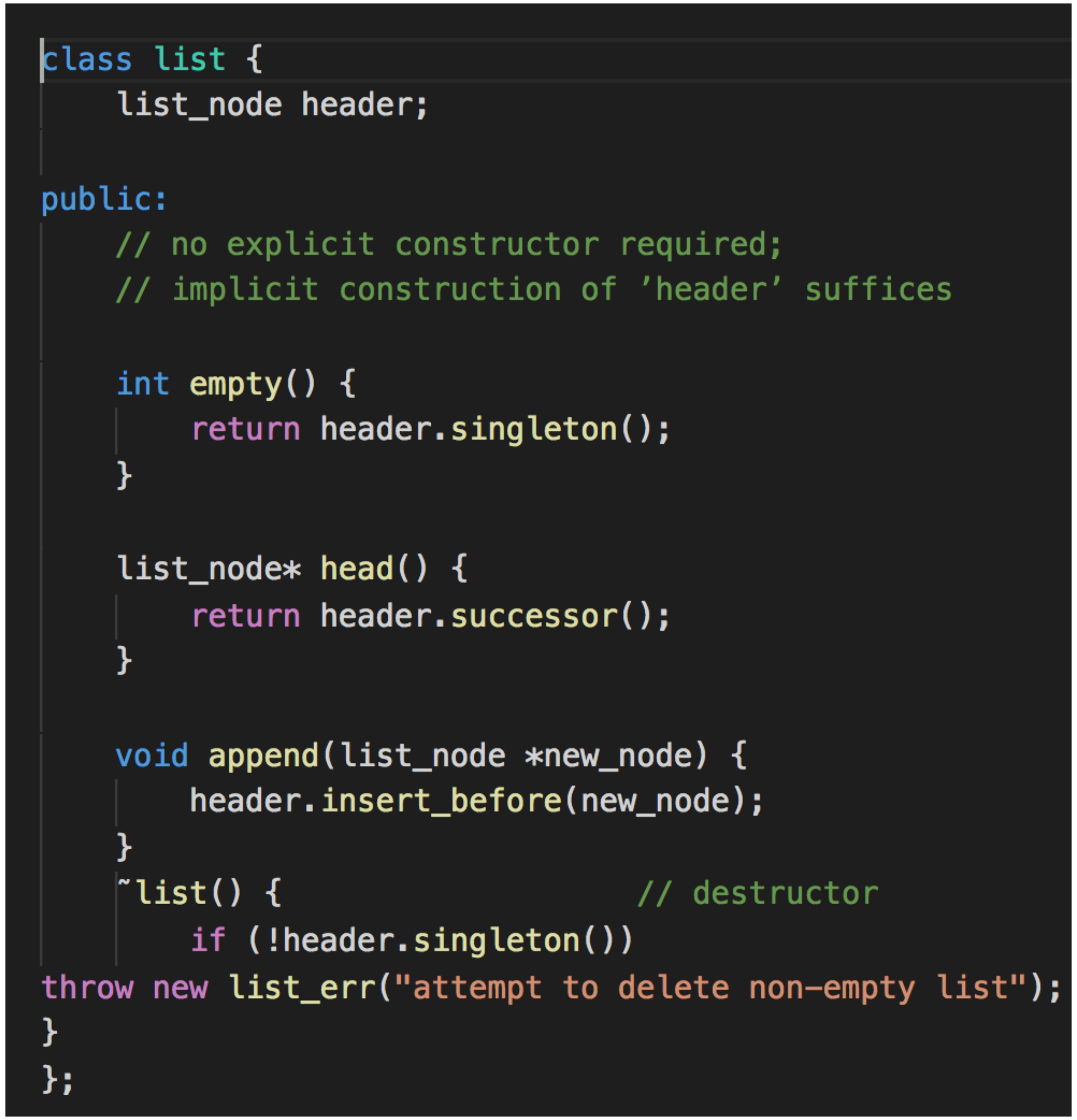


```
class list_node {
    list_node* prev;
    list_node* next;
    list_node* head_node;

public:
    int val;
    list_node() {
        prev = next = head_node = this;
        val = 0;
    }

    list_node* predecessor() {// exception
        if (prev == this || prev == head_node) return 0;
        return prev;
    }

    list_node* successor() {
        if (next == this || next == head_node) return 0;
        return next;
    }

    bool singleton() {
        return (prev == this);
    }
```

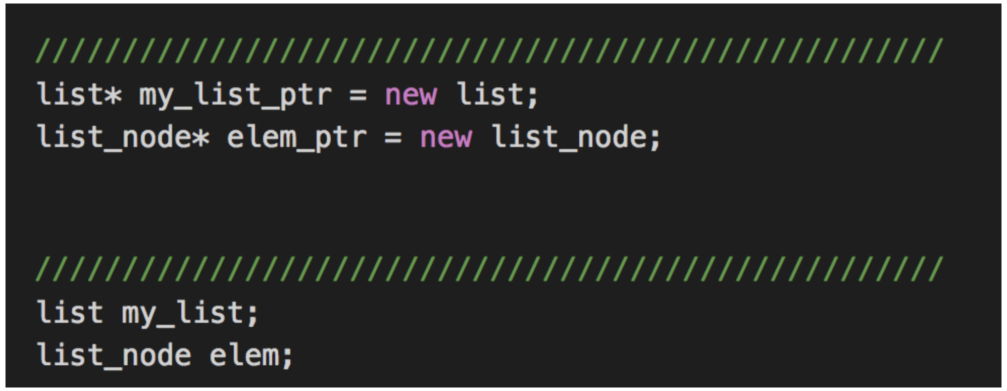


```
        void insert_before(list_node* new_node) {
            if (!new_node->singleton())
                throw new list_err("attempt to insert node already on list");
            prev->next = new_node;
            new_node->prev = prev;
            new_node->next = this;
            prev = new_node;
            new_node->head_node = head_node;
        }

        void remove() {
            if (singleton())
                throw new list_err("attempt to remove node not currently on list");
            prev->next = next;
            next->prev = prev;
            prev = next = head_node = this;
    }
```

List Class



```
class list {
    list_node header;

public:
    // no explicit constructor required;
    // implicit construction of 'header' suffices

    int empty() {
        return header.singleton();
    }

    list_node* head() {
        return header.successor();
    }

    void append(list_node *new_node) {
        header.insert_before(new_node);
    }
    ~list() {                   // destructor
        if (!header.singleton())
throw new list_err("attempt to delete non-empty list");
}
};
```



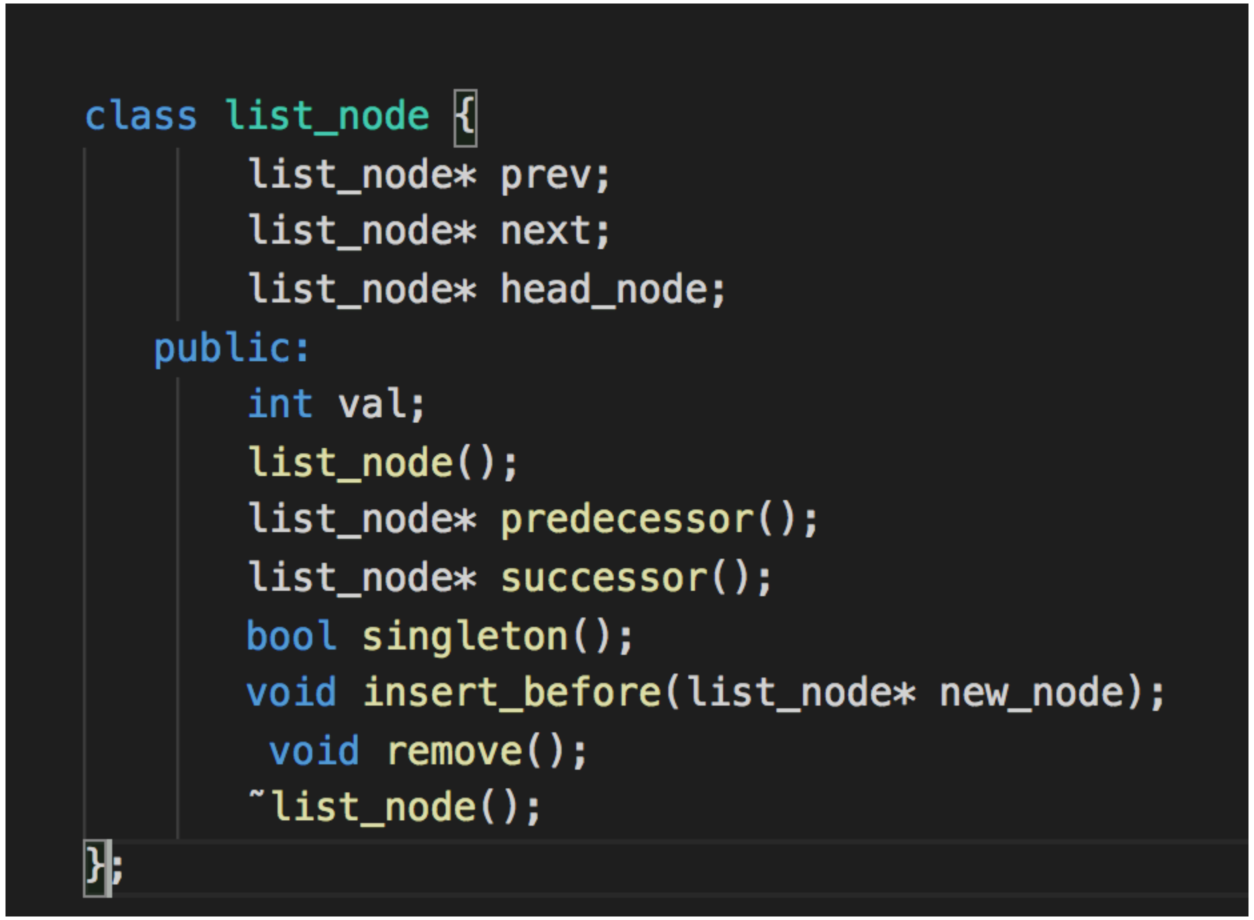
```
////////////////////////////////////////////////
list* my_list_ptr = new list;
list_node* elem_ptr = new list_node;


////////////////////////////////////////////////
list my_list;
list_node elem;
```

How are the two different?

- When created with **new**, an object is allocated in the heap
- When created via declaration it is allocated statically or on the stack, depending on lifetime

Public and Private Members

- Public
    - members that appear after the public label are exported from the class
    - members that appear before the label are not
- Private
    - members are only accessible within the class defining them
    - Not visible to any class derived from base class.



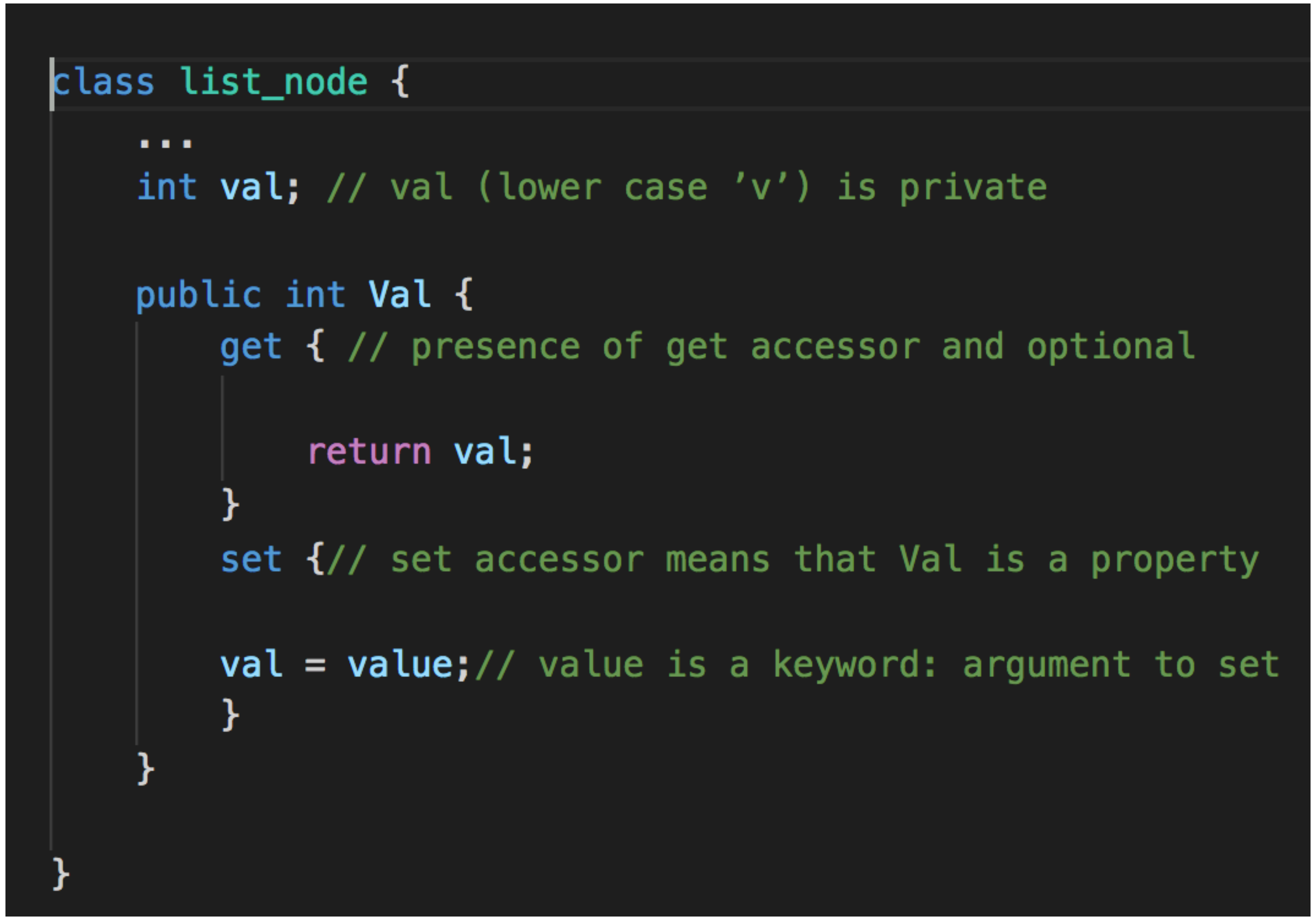
```
class list_node {
        list_node* prev;
        list_node* next;
        list_node* head_node;
    public:
        int val;
        list_node();
        list_node* predecessor();
        list_node* successor();
        bool singleton();
        void insert_before(list_node* new_node);
         void remove();
        ~list_node();
};
```

Constructors and Destructors

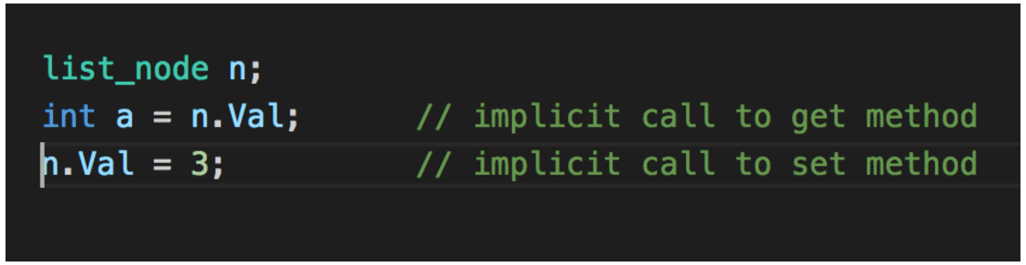
- Constructor method
  - Creation of a class object causes the invocation of a programmer-specified initialization routine called the constructor
  - Same name as name of class
- *Destructor* method
  - invoked automatically when an object is destroyed, either by explicit programmer action or by return from the subroutine in which it was declared
  - Same name as name of class, preceded by ~

Accessor Mechanisms in C#

- C# provides a *property* mechanism
  - facilitates the declaration of methods to “get” and “set” values

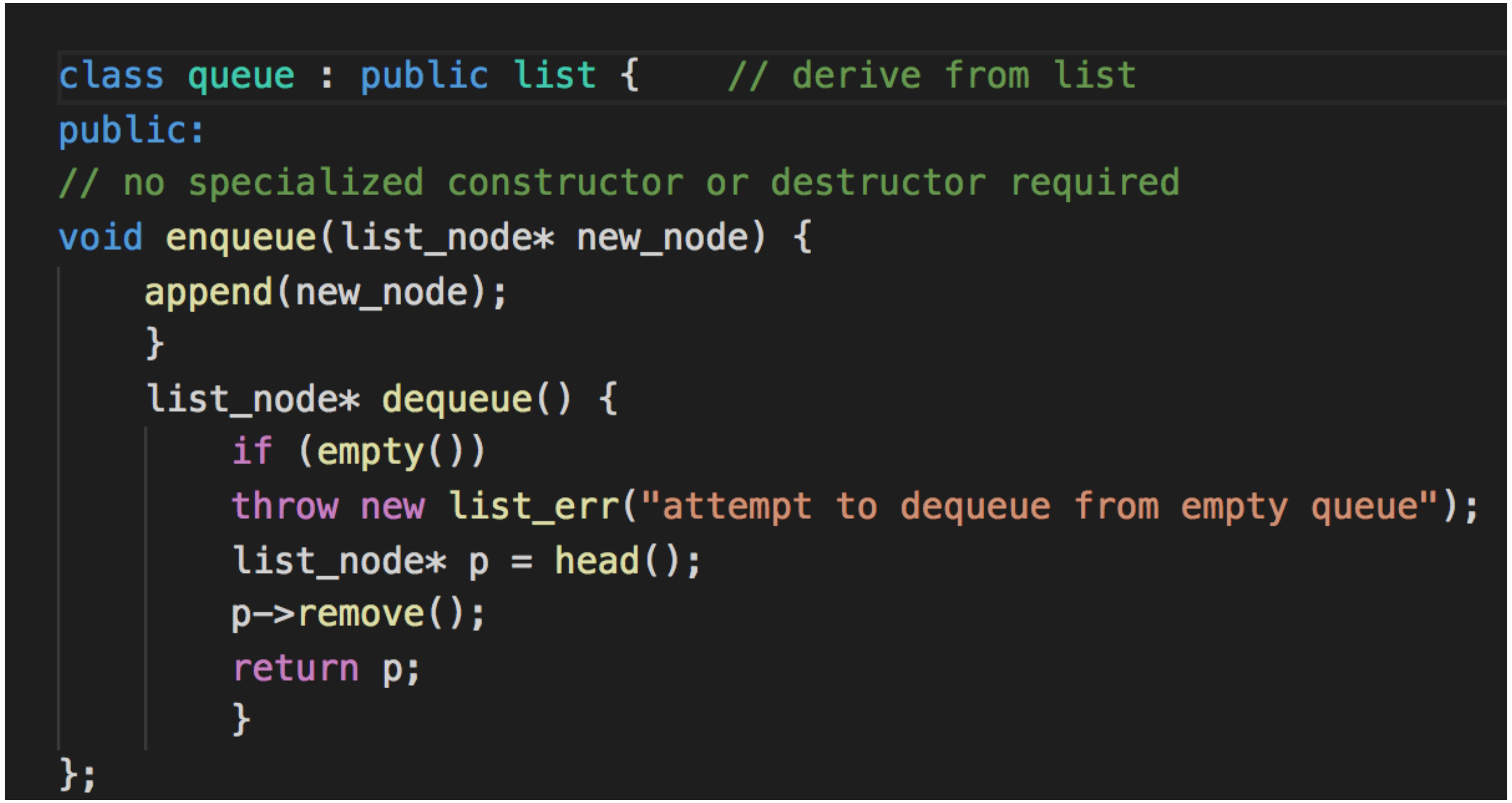


```
class list_node {
    ...
    int val; // val (lower case ’v’) is private

    public int Val {
        get { // presence of get accessor and optional

            return val;
        }
        set {// set accessor means that Val is a property

        val = value;// value is a keyword: argument to set
        }
    }

}
```



```
list_node n;
int a = n.Val;          // implicit call to get method
n.Val = 3;              // implicit call to set method
```

Derived Classes



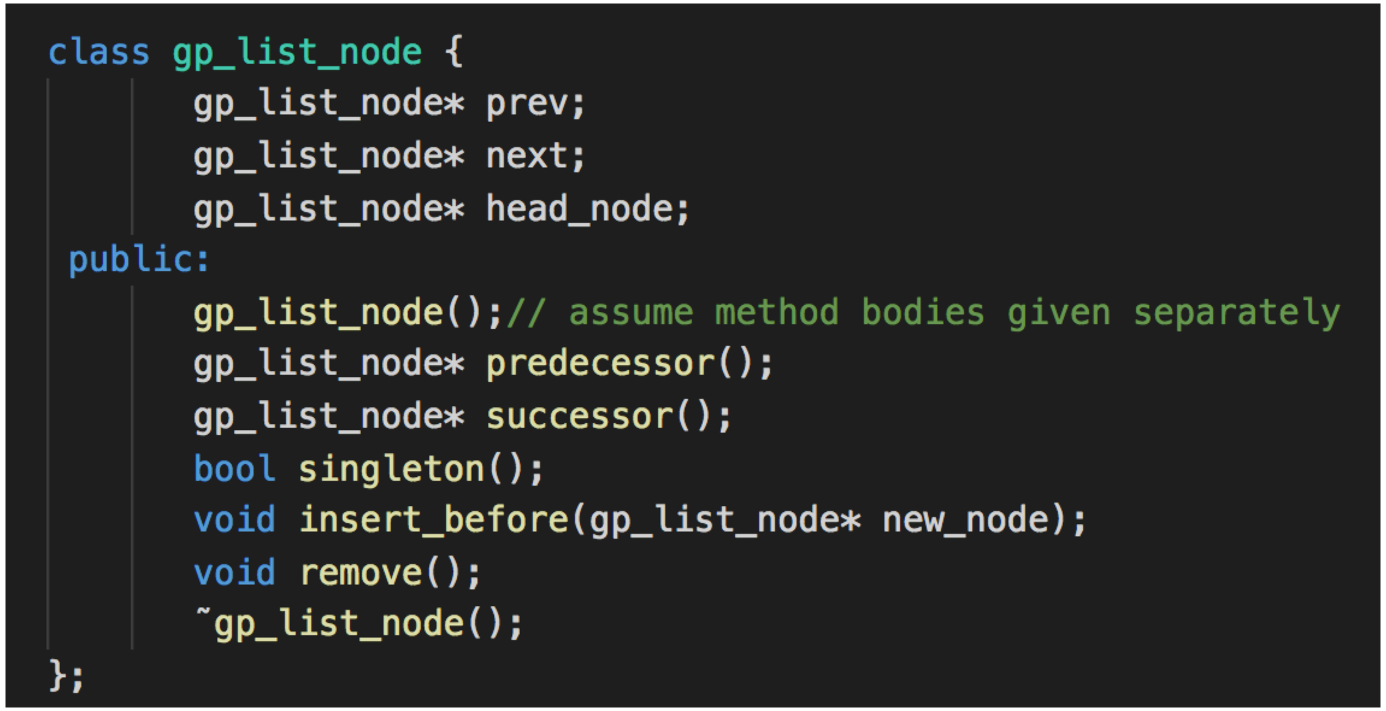
```
class queue : public list {    // derive from list
public:
// no specialized constructor or destructor required
void enqueue(list_node* new_node) {
    append(new_node);
    }
    list_node* dequeue() {
        if (empty())
        throw new list_err("attempt to dequeue from empty queue");
        list_node* p = head();
        p->remove();
        return p;
        }
};
```

- What if we wanted to create a queue? Use already existing ”list”

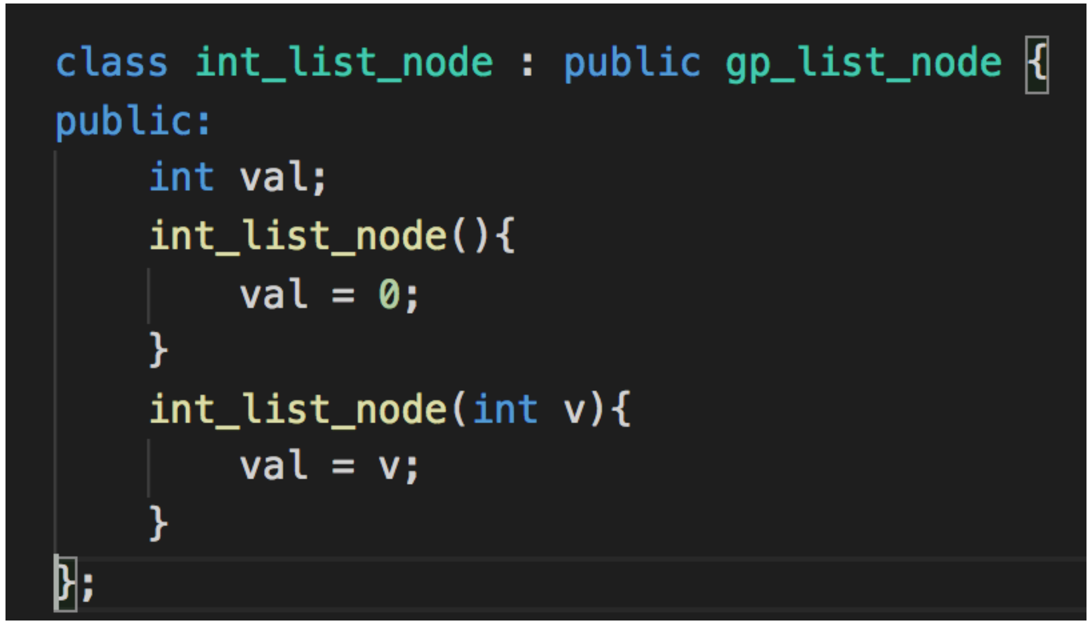
- Derived class?
- Base class?
- What has been added?

General Purpose Base Class

- Using inheritance, can create a general-purpose element base class that contains only the fields and methods needed to implement list operations



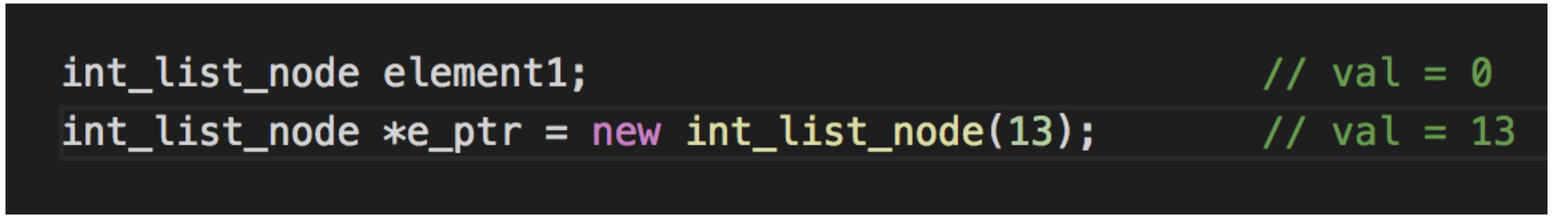
```
class gp_list_node {
        gp_list_node* prev;
        gp_list_node* next;
        gp_list_node* head_node;
 public:
        gp_list_node();// assume method bodies given separately
        gp_list_node* predecessor();
        gp_list_node* successor();
        bool singleton();
        void insert_before(gp_list_node* new_node);
        void remove();
        ˜gp_list_node();
};
```



```
class int_list_node : public gp_list_node {
public:
    int val;
    int_list_node(){
        val = 0;
    }
    int_list_node(int v){
        val = v;
    }
};
```

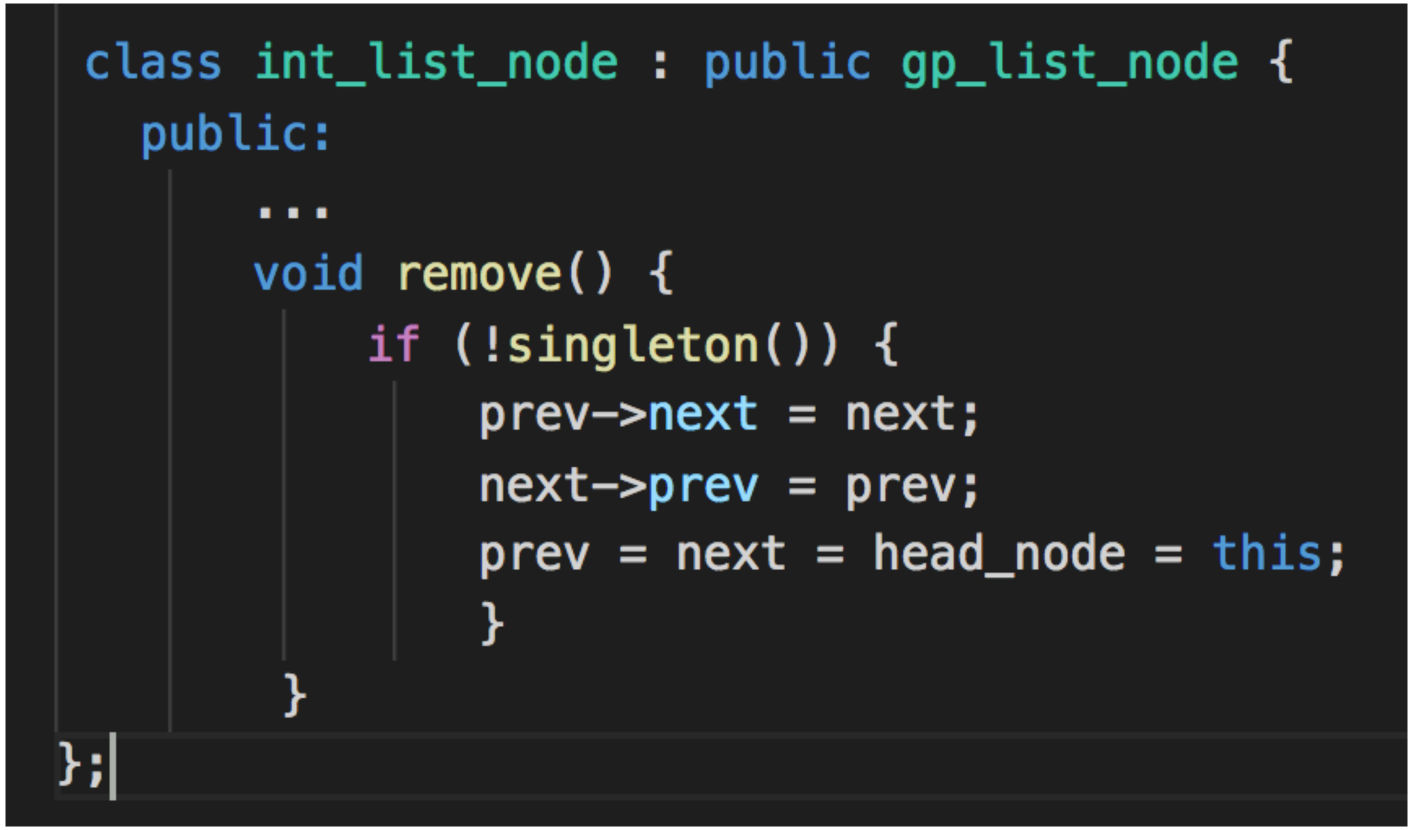
Constructor Overloading

- Can use overloading to provide two alternative implementations of constructor



```
int_list_node element1;                                  // val = 0
int_list_node *e_ptr = new int_list_node(13);           // val = 13
```

Base Class Method Modification



```
class int_list_node : public gp_list_node {
  public:
      ...
      void remove() {
          if (!singleton()) {
              prev->next = next;
              next->prev = prev;
              prev = next = head_node = this;
              }
       }
};
```