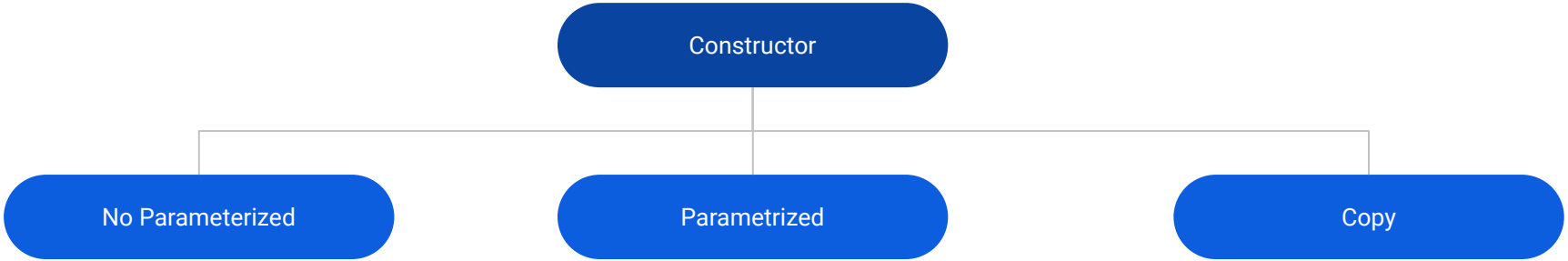
EE538: Computing and Software for Systems Engineers

Lecture 4: A Tour of the C++ Language

University of Southern California

Instructor: Arash Saifhashemi

Constructor




```
Point()                 Point(int i, int j)     Point(const Point& r)

Point p;                Point p(1, 3)           Point p1 = p2;
```

- **No parameterized** and **copy constructor** will be provided by compiler if we don't define ANY constructor
  - We call them **default constructor** and **default copy constructor.**
  - If you write ANY constructor, the default constructor is not created.
  - Default constructor does not necessarily initialize member variables.

Constructor

See src/class/constructor_main.cc

```
class Point {
public:
 Point() {
   i_ = 5;
   j_ = 5;
   std::cout << "NO PARAMETERIZED constructor." << std::endl;
 }
private:
 int i_;
 int j_;
};
```

Constructor Initialization List

```
class Point {
public:
// PARAMETERIZED Constructor Version 1
 Point(int i, int j) {
   std::cout << "**PARAMETERIZED constructor." << std::endl;
   i_ = i;
   j_ = j;
   z_ = new int;
 }

 // PARAMETERIZED Constructor Version 2
 Point(int i, int j) : i_(i), j_(j) {
   std::cout << "**PARAMETERIZED constructor." << std::endl;
   z_ = new int;
 }

private:
 int i_;
 int j_;
};
```

Constructor Initialization List

```
class Point {
public:
// PARAMETERIZED Constructor Version 1
 Point(int i, int j) {
   std::cout << "**PARAMETERIZED constructor." << std::endl;
   i_ = i;
   j_ = j;
   z_ = new int;
 }

 // PARAMETERIZED Constructor Version 2
 Point(int i, int j) : i_(i), j_(j) {
   std::cout << "**PARAMETERIZED constructor." << std::endl;
   z_ = new int;
 }

private:
 int i_;
 int j_;
};
```

- **Initialization list:**
  - Initializing member variables **before** the constructor's body**.**
- Initialization in the **initialization list vs the body**:
  - In the initialization list, the member variable gets **constructed** and **initialized at the same time**.
    i. Copy constructor
  - In the body, the member variable gets **constructed first**, **then assigned to**:
    i. Constructor
    ii. Copy assignment

Copy Constructor

See src/pointers/shallow_copy_main.cc

- What you need to know
  - Gets executed when the object is **copied**.
  - Will be created by default but you can overload it.
  - The default copy constructor performs **shallow** copy (i.e. members on dynamic memory won't be copied)
  - If you have **dynamic memory**, you should provide **deep copy**
  - STL containers: copy constructor copies all items
    i. E.g. Copying a vector might have a huge cost

```
Point(const Point &p2) {
   std::cout << "COPY constructor." << std::endl;
   i_ = p2.GetI();
   j_ = p2.GetJ();
 }
```

Destructor

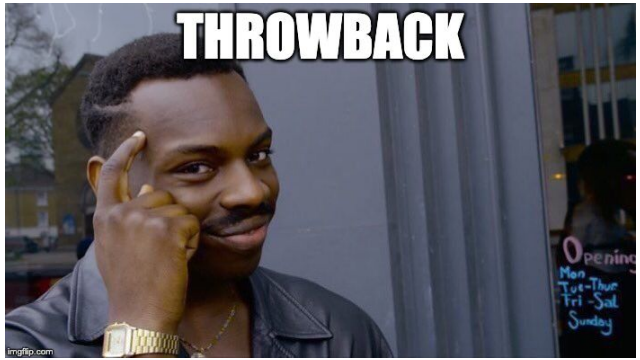
`~Point()`

- What you need to know
  - Gets executed when the object is **destroyed**.
  - If you don’t write one, the compiler generates a default one.
  - Very common for dynamic memory allocation
    i. Use **delete** in destructor

```
~Point() { std::cout << "DESTRUCTOR." << std::endl; }
```

When does a variable get destructed?

- Destruction of an object
  - **End** of **program**
  - **End** of **function**
  - Variable goes **out of scope**
  - **Delete** operator
- In most cases you only need a destructor only if you are using dynamic memory allocation.
  - The destructor can the delete the dynamic memory.



Throwback

- When you see a pointer, check for misuse:
  - Is it **deleted** correctly?
    i. Memory leak
  - Is it **initialized**?
    i. Can crash
  - Is the pointer value itself **modified**?
    i. Can crash
- What if the pointer goes out of scope?
- What if the variable that the pointer is pointing to goes out of scope or deleted?

```
void F(Point local_p) {
 //
 std::cout << "Inside F. Pass by value" << std::endl;
 std::cout << "local_p.i_: " <<
    local_p.i_ << ", local_p.j_: " << local_p.j_
            << std::endl;
}
```

- The **copy constructor** and **destructor** will be called for any local variable inside this function, including the ones that are **passed by value.**

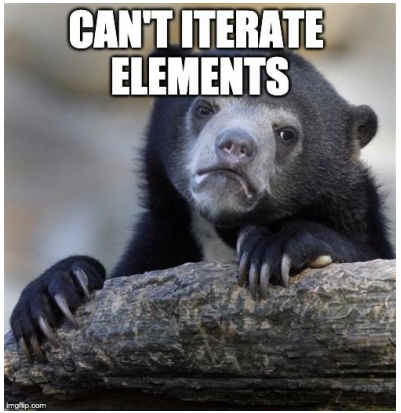
```
void F_reference(Point &p) {
 std::cout << "Inside F. Pass by reference" << std::endl;
}
```

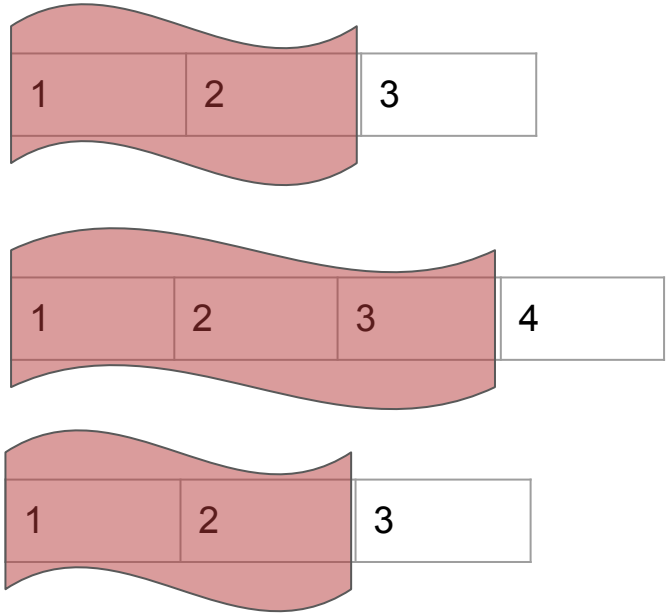
- The copy constructor and destructor will not be called for pass-by-reference parameters!

Other STL Containers

std::stack

- Conceptually, stack is like a vector, but we can only access its last element.
  - LIFO ordering
  - We can’t iterate all of its elements.
  - That means there is no **begin** and **end**!
- It provides these methods:
  - empty()
  - size()
  - top()
  - push()
  - pop()
- Homework:
  - Find time complexity of the above functions
  - Write a function to print the stack



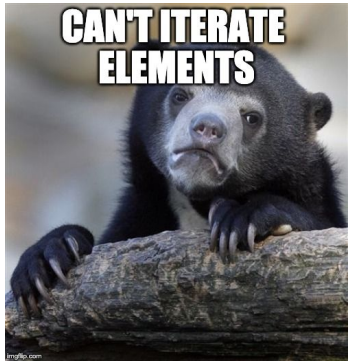


std::stack

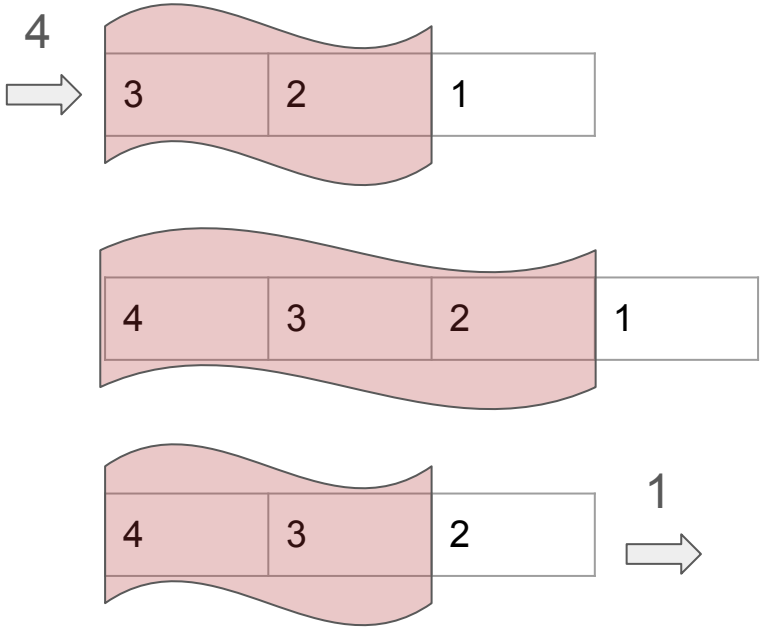
- What should you be worried about?
  - Don’t top() or pop() when the stack is empty
  - Don’t push when the stack is full

```
std::stack<int> s;
 int r = s.top();  // Seriously?
 s.pop();          // Don't do this either!

 // Do this instead
 if(!s.empty()){
   s.pop();
 }
```

std::queue



- Conceptually, queue is like a vector, but we can always only access its **first(front)** and **last(rear)** elements.
  - FIFO
  - We can’t iterate all of its elements.
  - That means there is no **begin** and **end**!
- It provides these methods:
  - empty()
  - size()
  - front(): read front
  - push(): push into rear
  - pop(): pop from front
- Homework:
  - Write a function to print a queue



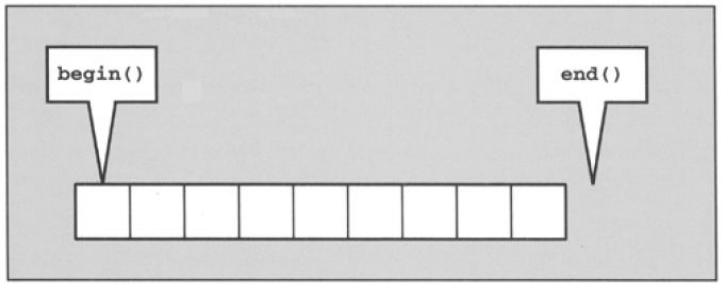
std::list

- Like vector, list is a collection of items, but:
  - No **contiguous memory locations**, therefore
    - No indexing operator
    - Slow indexing: O(n) -> Vector was O(1)
    - Fast insert/delete (after indexing):
      - O(1) -> Vector was O(n)
    - You should almost never use std::list
      - Because of cache
- It **CAN** be iterated
- It provides these methods:
  - empty()
  - size()
  - insert(), erase()
  - front(), back()
  - push_front(), pop_front()
  - push_back(), pop_back()

1	2	3	4

std::list indexing

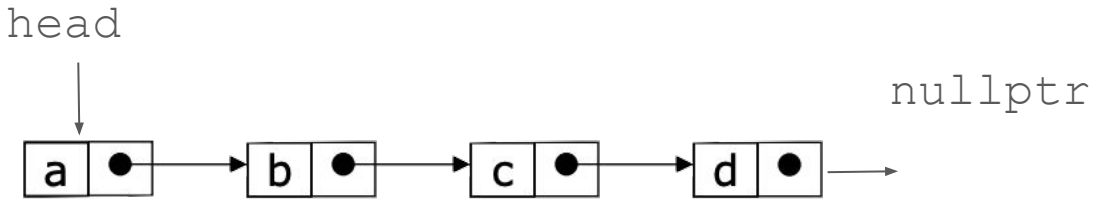
- Using iterators, we only have access to the **front** and **back** of the list
  - Front: **begin**()
  - One after back: **end**()
- So index i would be:
  - it=begin(); it++ (i times)
- We can’t directly add to iterators
  - STL provides these functions:
    - **std::advance**
    - **std::next**

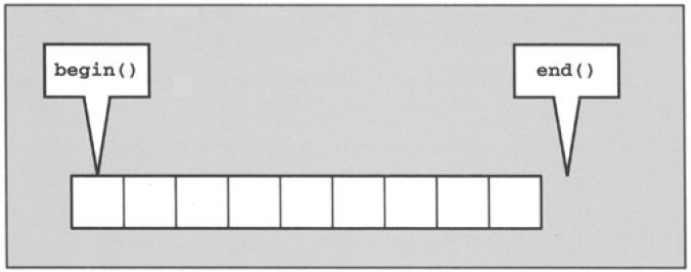


Linked List Implementation

Linked Lists

- A sequence of **Nodes**
- Each node has:
  - Value
  - Next pointer
- Each node points to the next one
- Last node points to nothing (**nullptr)**
- First node is in **head**

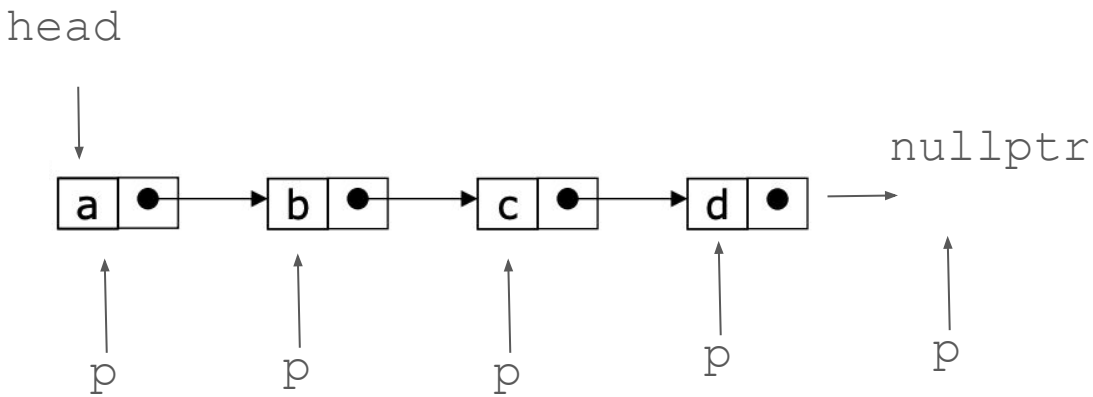





```
struct ListNode {
 int val;
 ListNode *next;
 ListNode(int x) : val(x), next(nullptr) {}
};
```

Iterating a List

- Usually, all we have is the **head** pointer
  - How to find the last node?

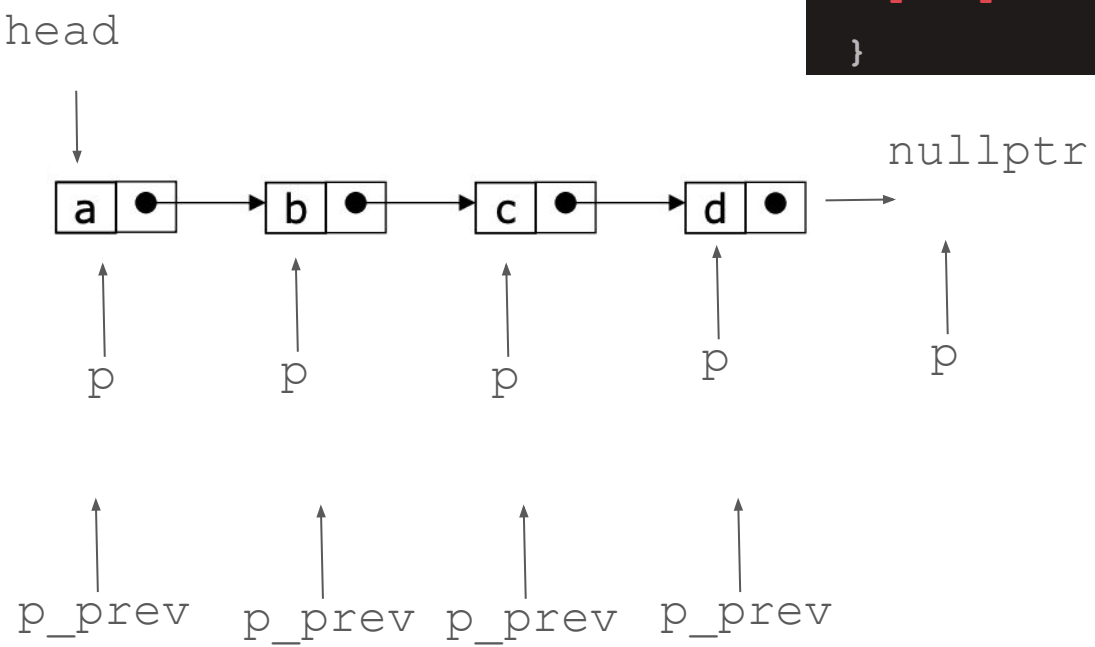



```
p = head;
while (p != nullptr) {
   p = p->next;
 }
```

Iterating a List

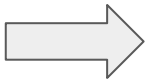
- Usually, all we have is the **head** pointer
  - How to find the last node?

```
while (p != nullptr) {
   p_prev = p;
   p = p->next;
 }
```

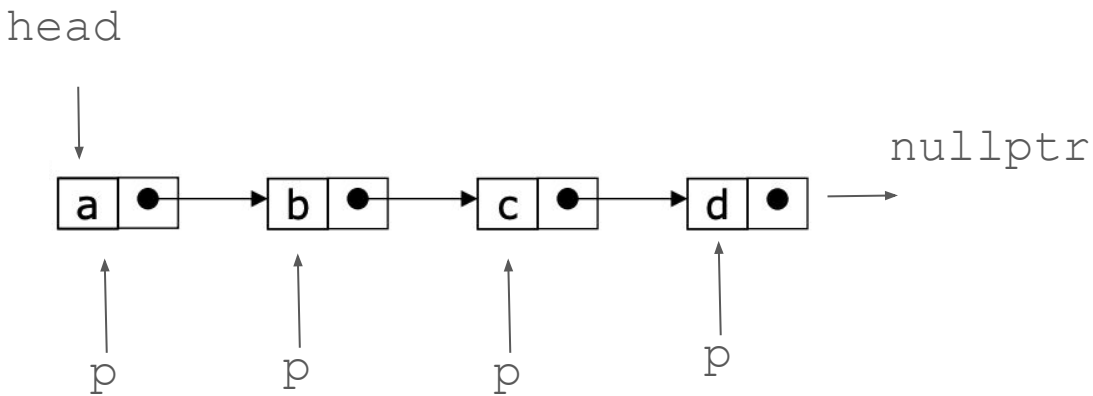


Getting the Tail Pointer (Optimized)

```
while (p != nullptr) {
   p_prev = p;
   p = p->next;
 }
```

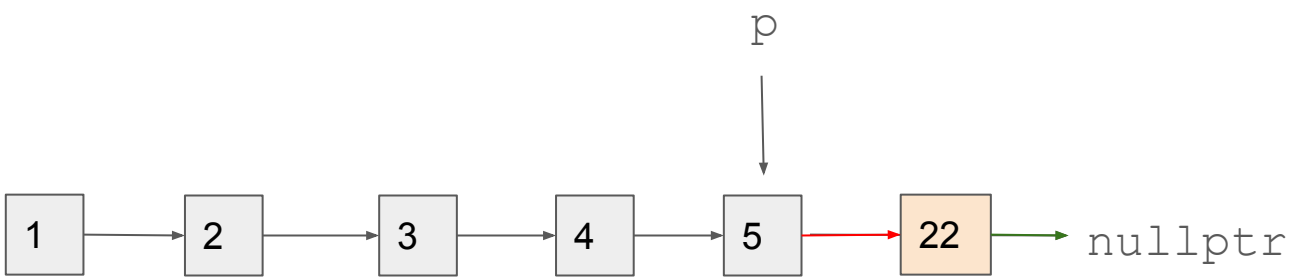


```
while (p->next != nullptr) {
   p = p->next;
 }
```



push_back

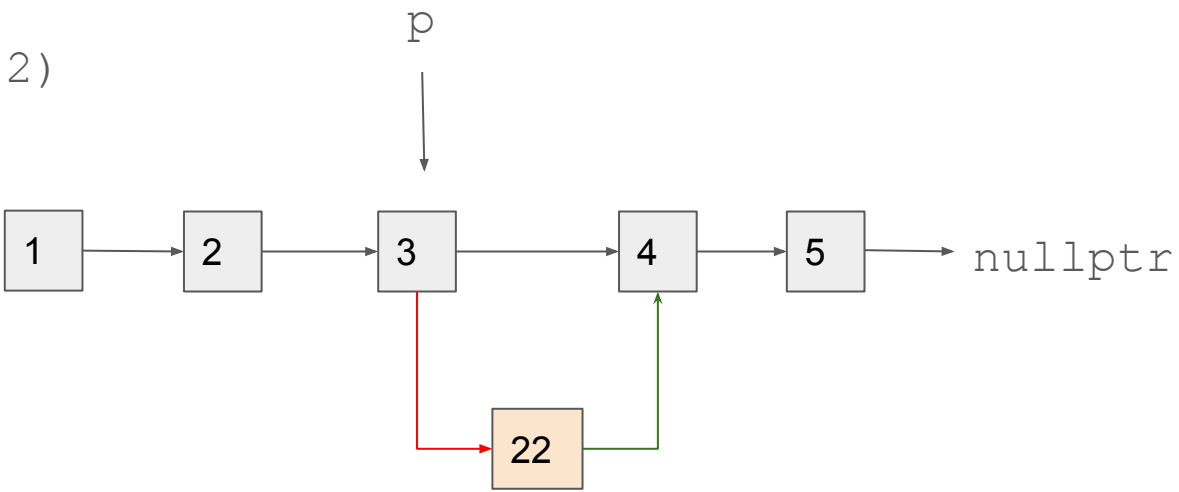
```
push_back(22)
```




```
p = GetBackPointer();

ListNode* newNode = new ListNode;

P -> next = newNode;
```

insert_after

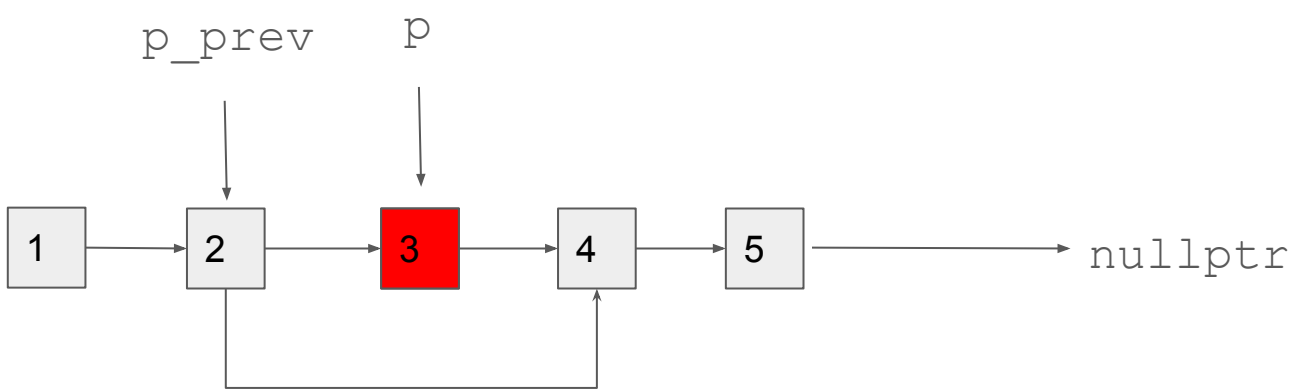
```
insert_after(p, 22)
```




```
ListNode* newNode = new ListNode;

newNode -> next = p -> next;

P -> next = newNode;
```

erase

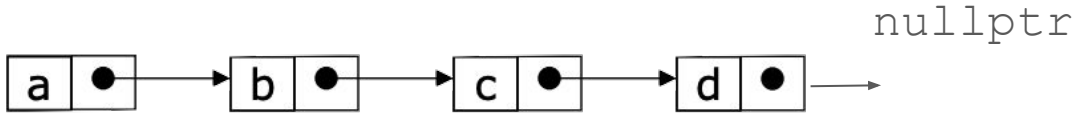
```
erase(p)
```




```
p_prev -> next = p->next;

delete p;
```

There is a famous interview question that requires you to do this without p_prev. Homework :)

Activity




```
class SinglyLinkedList {
public:
 SinglyLinkedList() { }
 ~SinglyLinkedList() { }
 ListNode *head_;
 bool empty();
 int size();
 void push_back(int i);
 void pop_back();
 int back();
 ListNode *GetBackPointer();
 ListNode *GetIthPointer(int i);
 void print();
};
```

Operator Overloading

Operator overloading

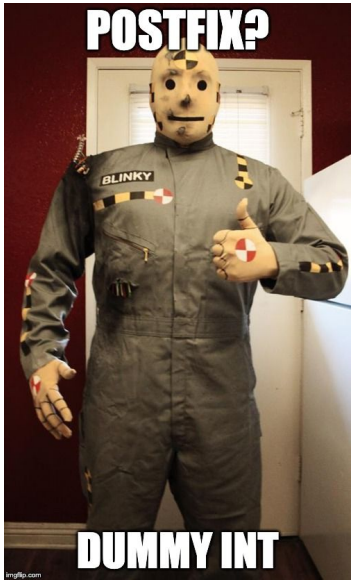
- Most operators can be overloaded
  - Examples: +, -, =, ++, ...

```
Point operator+(const Point &rhs) {
   Point res;
   res.SetI(i_ + rhs.GetI());
   res.SetJ(j_ + rhs.GetJ());
   return res;
 }
```

Overloading pre and postfix unary operators(++, --)

```
// Prefix overload
 // ++p;
 Point& operator++() {
   i_++;
   j_++;
   return *this;
 }
```

```
// Postfix overload
// p++;
Point operator++(int) {
  Point temp = *this;
  i_++;
  j_++;
  return temp;
}
```



```cpp
class Point {
public:
 // Copy assignment
 Point &operator=(const Point &rhs) {
   i_ = i_ + rhs.i_;
   j_ = i_ + rhs.j_;
   std::cout << "Copy assignment" << std::endl;
   return *this;
 }

 Point operator+(const Point &rhs) const {
   Point res;
   res.i_ = i_ + rhs.i_;
   res.j_ = j_ + rhs.j_;
   return res;
 }

 // Prefix overload, Point p2 = ++p1;
 Point &operator++() {
   i_++;
   j_++;
   return *this;
 }

 // Postfix overload, Point p2 = p1++;
 Point operator++(int) {
   Point temp = *this;  // 1.
   ++*this;             // 2
   return temp;         // 3.
 }
```

(Optional) Overloading << and >>

```
std::ostream &operator<<(std::ostream &os, const Point &m) {
 return os << "( " << m.GetI() << ", " << m.GetJ() << " )";
}

std::istream &operator>>(std::istream &is, Point &p) {
 std::cout << "Enter i ";
 is >> p.i_;
 std::cout << "Enter j ";
 is >> p.j_;
 return is;
}
```

Deep and Shallow Copy

Shallow Copy

```
class Student_shallow {
public:
 Student_shallow() { id = new int(0); }
 ~Student_shallow() {
   delete id;
   id = nullptr;
   cout << "Delete Student_shallow!" << endl;
 }

 int* id;
};

int main() {
 Student_shallow a;
 Student_shallow b = a;
 Student_shallow c;
 c = a;
 cout << *a.id << *b.id << *c.id << endl;
 *c.id = 1;
 cout << *a.id << *b.id << *c.id << endl;
}
```

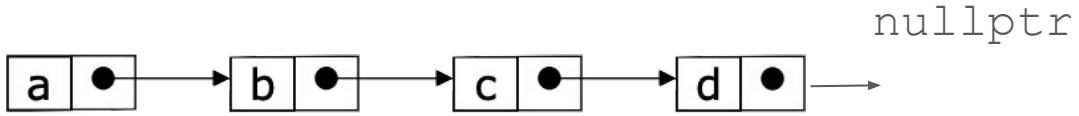
- Suppose a class has a pointer member variable
  - Shallow copy: only copy the pointer
  - Deep copy:

Shallow Copy

```
class Student_deep {
public:
 Student_deep() { id = new int(0); }
 Student_deep(const Student_deep& rhs) {
   id = new int(*(rhs.id));
 }

 Student_deep& operator=(const Student_deep& rhs) {
   id = new int(*(rhs.id));
   return *this;
 }
 ~Student_deep() {
   delete id;
   id = nullptr;
   cout << "Delete Student_deep!" << endl;
 }

 int* id;
};

int main() {
 Student_deep a;
 Student_deep b = a;
 Student_deep c;
 c = a;
 cout << *a.id << *b.id << *c.id << endl;
 *c.id = 1;
 cout << *a.id << *b.id << *c.id << endl;
}
```

- Suppose a class has a pointer member variable
  - **Shallow copy**: only copy the pointer
  - **Deep copy:** Allocate a new memory location and copy everything that the original pointer was pointing to.

Activity



- Write Deep Copy for the SinglyLinkedList class.

```
class SinglyLinkedList {
public:
 SinglyLinkedList() { }
 ~SinglyLinkedList() { }
 ListNode *head_;
 // ...
 };
```