Chapter - 20
Advanced Pointers
Pointers, Structures, and Classes

public:

};
new operator

The new operator creates a new variable from a section of memory called the heap and returns a pointer to it.

```cpp
public:

};
```
**delete operator**

The “delete” operator returns the storage to the heap. Only data allocated by “new” can be returned this way.

Normal variables:

```cpp
delete pointer;  // Where pointer is a pointer to
                // a simple object

pointer = NULL;
```

Array variables:

```cpp
delete pointer[];  // Where pointer is a
                   // pointer to an array

pointer = NULL;
```
Example

{  
  /*
   */
  delete[] data_ptr;
  data_ptr = NULL;
}

What would happen if we didn’t *free* the memory?
Linked List
Linked List

public:

    public:

    private:

            friend class linked_list;

};

public:

    linked_list_element *first_ptr; // First element

// Initialize the linked list
linked_list(void): first_ptr(NULL) { }

// ... Other member functions

};
Adding an element

```c
new_ptr = insert_ptr->next;
insert_ptr->next = new_ptr;
```
C++ code to add an element

```cpp
{

}
```
Finding an element in a list

Note: The following two statements are equivalent:

```c++
(*current_ptr).data = value;
current_ptr->data = value;
```
The C++ code

{  
/
  */
  /*  
   break;
   break;
  }
Double Linked List

- head_ptr
- next_ptr
- previous_ptr
- tail_ptr
- insert_ptr
Double Linked List

private:

    public:

    private:

    friend class double_list;
};

public:

double_list(void) { head_ptr = NULL; }

// ... other member functions
Adding an element

```c
new_ptr->next_ptr = insert_ptr
new_ptr->previous_ptr = insert_ptr->previous_ptr
insert_ptr->previous_ptr->next_ptr = new_ptr;
insert->previous_ptr = new_ptr;
```
Adding an element

{  
/*  
 */  
  break;  
  break;  
}
Trees
Trees

class tree {
    private:

    public:

    private:

    friend class tree;
};

public:

tree(void) { root = NULL; };

// ... other member function
};
Tree Search

(root node)

apple

Ø

lemon

right

left

pear

Ø

grape

orange

plum

orange > lemon
search right

orange < pear
search left

orange = orange
success
Tree Insert Rules

The algorithm for inserting a word in a tree is:
1. If this is a null tree (or sub-tree), create a one-node tree with this word.
2. If this node contains the word, do nothing.
3. Otherwise, enter the word in the left or right sub-tree, depending on the value of the word.

Does this follow the two rules of recursion?
Adding a node

```cpp
void tree::enter(char *word) {
    enter_one(root, word);
    return;
}
```
Printing a Tree

The printing algorithm is:
1. For the null tree, print nothing.
2. Print the data that comes before this node (left tree).
3. Print this node.
4. Print the data that comes after this node (right tree).

```cpp
def print_tree(node):
    if node is None:
        return
    std::cout << std::endl;
    std::cout << node.value << ' ' << std::endl;
    print_tree(node->left);
    std::cout << std::endl;
    print_tree(node->right);
```