CS 162
Intro to Computer Science II

Lecture 22

Recursion

Template, STL

Linked List

3/8/24

Oregon State University

Odds and Ends

- Design exercise 4 & document due Sunday midnight via Canvas

Today's topic(s)

- Recursion

Exercise

- Write your own recursive *int pwr()* function that takes two integers as arguments and returns the integer result.
    - Prototype: int pwr(int base, int exp);

pwr(2, 10)

$2^{10} = 1024$

Demo…

Pros and Cons of Recursion

- Pros
  - Readable
  - Sometimes easier to conceptualize for problems that have many moving parts

- Cons
  - Efficiency
  - Memory usage
    - Each call to the function makes a new function stack frame (see previous slides)

Today's topic(s)

- Templates
- Standard Template Library (STL)
- Linked List

Templates

- How would you write a function to swap two ints?

```
void swap (int& a, int& b){
        int temp = a;
        a = b;
        b = temp;
}
```

- What if we also want to swap two floats?

```
void swap (float& a, float& b){
        float temp = a;
        a = b;
        b = temp;
}
```

- Two doubles? Two chars? Two strings? Two Animal objects?...

Function Templates

- Useful when have a general algorithm which doesn't change even if types change
- **Algorithm Abstraction**: expressing algorithms in a very general way so that we can ignore incidental detail and concentrate on the substantive part of the algorithm
- Classic example: swap
  - We can create a template function which can take any type

```
template <class T>
void swap (T& a, T& b){
        T temp = a;
        a = b;
        b = temp;
}
```

Function Templates

- template <class T>
  - Referred to as template prefix
  - Tells the compiler that the definition that follows is a template
  - T is a type parameter
- To call this function template, we can explicitly specify our template parameter using angle brackets:
  - swap<int>(i, j); // where i and j are ints
  - swap<float>(x,y); // where x and y are floats
  - swap<Animal>(a1, a2); //where a1 and a2 are Animals
- Since swap() takes parameters of the template type T, we don’t need to explicitly specify the template type, i.e. these also work:
  - swap(i, j); // where i and j are ints
  - swap(x,y); // where x and y are floats
  - swap(a1, a2); //where a1 and a2 are Animals

Function Templates

- We can write function templates that include any number of template parameters, e.g:

```
template <class T, class U>
void print_two_things(T first, U second){
    cout << first << second << endl;
}
```

And we can call it as before:

```
    print_two_things<string, int>("number: ", 1);
    print_two_things(2.5, 'e');
```

Note:

- The compiler generates a new implementation of the template for each type with which it is used.
    - This means concrete implementations of templates (i.e. int, float) are not created until compile time
- Therefore, we cannot explicitly compile template implementations into object files from .cpp files.
    - In fact, we can't separate template implementations into separate .cpp files at all
    - Instead, we need to write template implementations either in the same file in which they are used or else in a header (.h) file

Template Classes

- Work the same way as templated functions
- All functions within the class will operate on the provided types
- Scope with `ClassName<T>::functionname()`
- Each function needs the Template prefix

Today's topic(s)

- Template
- Standard Template Library (STL)

Standard Template Library (STL)

- C++ STL can be broken down into:
  - **Containers** – general purpose data structures (templates) for holding things
  - **Iterators** – special classes for traversing containers
  - **Algorithms** – sorting, searching, etc.

- Iterators make it possible to run the algorithm on the containers

- The STL is a great resource:
  - It contains a wide variety of very useful structures and algorithms
  - It is well-implemented, which means the structures and algorithms perform very efficiently
  - In general, it allows us to avoid re-inventing the wheel

Introducing STL Containers

- Predefined templates that can store any type of data
- The appropriate container will be dictated by the application requirements
- Example considerations:
  - Does the data need to be stored?
  - How will the data be accessed?
    - Front to back
    - Randomly?
  - Will additional data ever need to be added or removed?
- Careful planning will allow you to write clean, efficient code

Types of Containers

- Sequential containers (vector, deque, list)
  - Programmer controls the order of the elements
- Associative containers (map, set, multimap, multiset)
  - Position of elements is controlled by container
  - Elements are generally accessed by using a “key”
- Adapters (stack, queue)
  - Use an existing type of container to build other types
    - In this context, we call these “Abstract Data Types”

Examples of C++ Containers

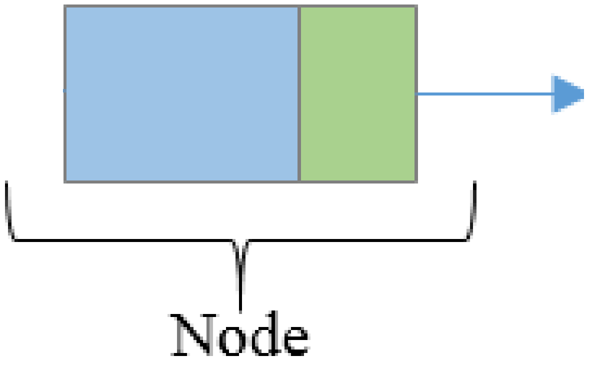
- <array> - stores a constant amount of data in contiguous memory
- <vector> - An array that can be resized
- <list> - Linked list that stores data in non-contiguous memory
- <set> - An ordered collection of items
- <queue> - Stores data & returns it in the order it was received
    - First in, first out
- <stack> - Stores data & returns it in the opposite order that it was received
    - First in, last out

- Generally, it is a good idea to refer to the STL documentation before starting a project

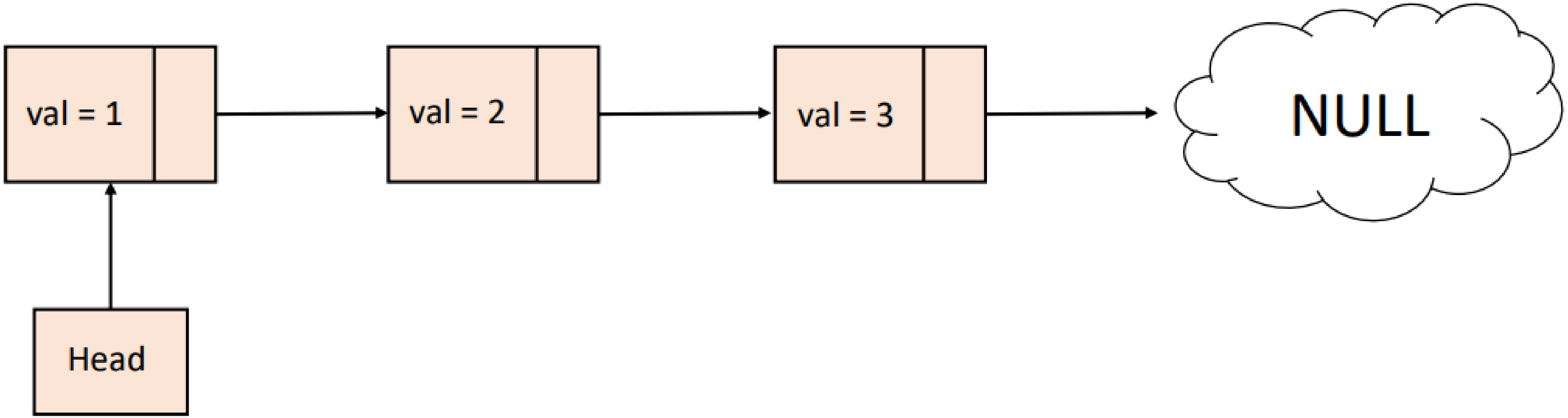
Linked List

- A list constructed using pointers
- Can grow and shrink easily while the program is running
- Not stored contiguously in memory
- Use structs to create

```
struct Node {
        int val;
        Node* next;
};
```



Singly Linked List



In class activity

- Use the code provided on Canvas, complete the following tasks:
  - Task 1: What does the code do? (Hint: Trace through the code by drawing the picture out)
  - Task 2: Write code to print the list you just created. Trace the code you wrote to verify
    - Hint: Use while loop and Node* current
  - Task 3: Delete the list you just created. Trace the code you wrote to verify
    - Hint: You might need another Node*

Pros and Cons of Singly Linked List

- Pros
  - Easy to implement
  - Insertion and deletion of elements can be done easily and doesn’t requires movement of all elements compared to an array
  - Can allocate or deallocate memory easily during its execution
- Cons
  - Uses more memory when compared to an array
  - No random access
  - Traversing in reverse is not possible for singly linked list