MTH 4300, Lecture 21

Templates;
A Template Class for Linked Lists;
Stacks

E Fink

April 28, 2025

1. Templates

The core data in our `Nodes` are `strings`. But most of the interesting operations that we perform on the data have code that almost entirely does not rely on the fact that our data fields happen to be `strings`. In fact, you could take every `string` in the code, and replace it with, say, `double`, and it ought to work in approximately the same way.

But performing tons of replacements is tedious and error-prone; and moreover, what if we want to work with several different linked lists containing different types of data? Creating several photo-copy classes seems silly.

There is a simple way to overcome this problem: templates! And in fact, you already know a bit about templates, since you know how to declare, for example, `vectors`, e.g.

```
std::vector<int> x;
```

The data type in the angle brackets is a *type parameter*.

Templates

Before discussing class templates, we will apply templates to functions. These allow for the writing of functions where parameters, return types, and local variable types that would ordinarily have to be fixed to be replaced by type variables.

To create a function template, simply immediately precede your function's declaration or definition with

```
template<typename T>
```

(which is customarily placed on the line above the ordinary signature line). The symbol `T` can be replaced with any other variable name. It will represent the name of a type that will be utilized within the function body, and which will be different each time the function is called.

You can then call the function by following the function name with `<int>`, `<double>`, `<string>`, or whatever type you want `T` to be replaced with in any particular call.

L21x1_fn.cpp

(In fact, if the specification of the type is omitted, the compiler will deduce the type from the arguments, if it can.)

Templates

Now, we'll use templates in our struct/class definitions. This is most appropriate for *container* classes: e.g., vectors or linked lists, where the data structure is primarily intended to hold collections of data of a particular type.

To allow your classes to have type parameters, simply begin their declaration with

```
template<typename T>
struct MyType { ...
```

Then, create objects with a declaration of the form

```
MyType<actual_type> varname;
```

L21x2_struct.cpp

(Notice: when you refer to `Node` in the declaration of the class, you simply call the class `Node`. However, outside of the class declaration, you refer to the data type as, e.g., `Node<int>`.)

2. A Template Class for Linked Lists

Let's package our Linked Lists in a template class, which is closer to how you would use a Linked List in practice:

```
template<typename T>
class LinkedList {
private:
    Node<T> *head;
public:
    LinkedList();
    void print() const;

    void push_front(const T&);
    void pop_front();
    T get_front();

    ~LinkedList();
    LinkedList(const LinkedList&) = delete;
    LinkedList& operator=(const LinkedList&) = delete;
};
```

A Template Class for Linked Lists

To explain:

- The constructor will just set `head` to be `nullptr`. And the `print()` function just prints all the elements.
- While there are many ways to insert, delete and read from a linked list, we don't want to make ourselves crazy, so we will just implement functions that insert/delete/read from the *front*. Inserting data will involve dynamically allocating Nodes. Deleting data will involve deleting Nodes.
- To avoid memory leaks, we want to make sure that ALL Nodes are deleted when the List goes out of scope. Hence we have a destructor.
- But the rule of 3 says that in this case, we need to implement a copy constructor and copy assignment, otherwise chaos can reign! We cheat: we use `= delete`, which simply PROHIBITS either type of copying.
- In general, we should be very careful about assuming that the `head` Node is NOT `nullptr`. We will `throw std::runtime_error()` when necessary, which is a lazy way to halt the program when we are approaching danger.

A Template Class for Linked Lists

One annoyance is that if you implement methods outside of the class declaration, you have to reintroduce the template, and include that reference next to the name of the class, just like you write when you declare objects of that class.

Pay attention to the signature lines of the constructor and destructor when we implement them outside the class!

L21x3_linked.cpp

Consider the following problem:

> *You are building a Word processor. Each action performed – typing a character, deleting text, changing font style – changes the document. You want to keep track of all actions performed, in the order performed, so that you can make Ctrl-Z (undo action) work.*

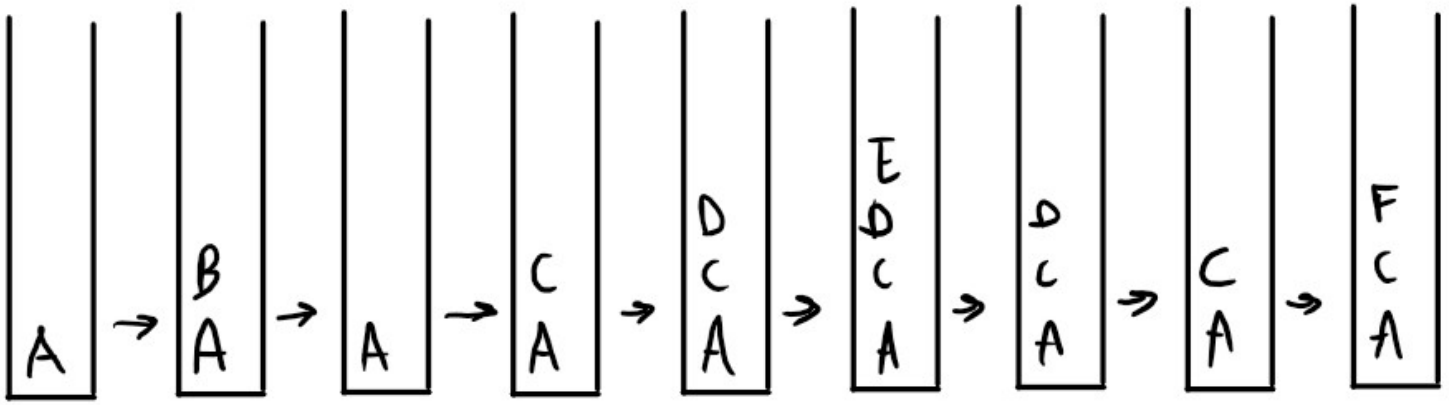
For example, if I

type A	type B	Ctrl-Z
type C	type D	type E
Ctrl-Z	Ctrl-Z	type F

what do I have on the screen at the end?

Stacks

You can visualize the "list" of actions as a pile, where the first actions are at the bottom, and further actions are added to the top – and older actions are removed from the top!



This is kind of like a list, but with a special, restricted set of operations we wish to perform on it. Note the "Last In, First Out" nature of the operations we perform.

Stacks

We can implement a class that is perfect to store actions for this type of problem: a *stack*. What a stack is can best be described presenting part of the class declaration:

```
template<typename T>
class Stack {
private: ????
public:
    Stack(): top{nullptr} {}
    void push(const T&);
    void pop();
    bool is_empty() const;
    T peek() const;
};
```

A stack is a data structure that supports the following operations:

`push()` should add a new element to the top of the stack.

`pop()` should remove the element from the top of the stack.

`peek()` should return the top element without removing it.

`is_empty()` should return `true` if there are no elements in the stack.

Stacks

So, how can we store the elements in a stack?

Option 1: a plain array. Big downside: absolutely fixed length.

Option 2: a vector. Small downside: periodic calls to `reserve()` are expensive.

Option 3: a linked list! (Downside: if you're not concerned about the amount of time `reserve()` takes, each typical non-resizing operation can be a little bit more expensive than with a vector)

The idea of the linked list implementation: the top of the stack is the FRONT of the linked list. After all, this element is pretty easy to add and remove. (Below is a depiction of adding the letter D to a stack already containing A, B and C.)