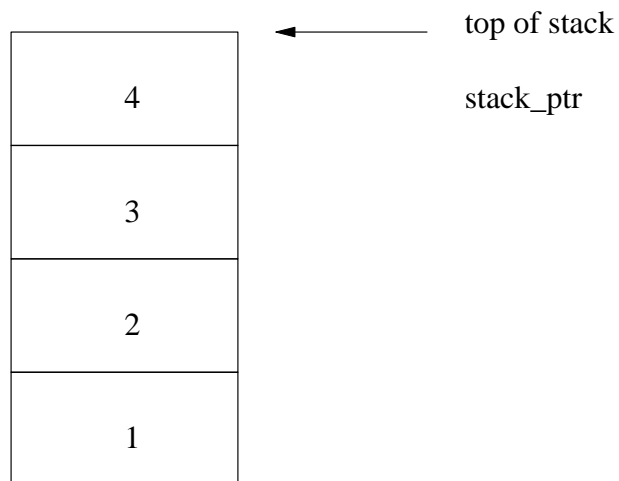


Stacks

- the behaviour of stack methods are to:
 - push to the top of a stack
 - remove from the top of a stack, via pop



Stacks

- here we have

- ```
push(1)
push(2)
push(3)
push(4)
```

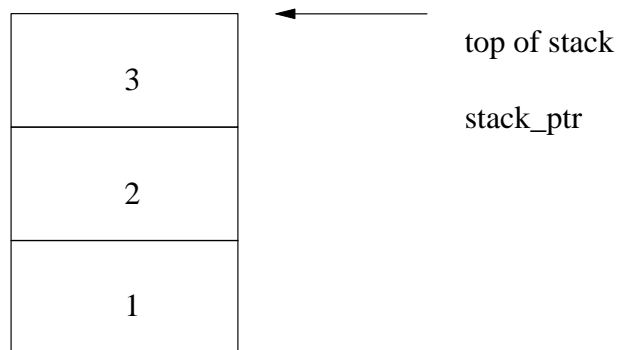
- and then executing

- ```
i = pop()
```

- yields the value 4 in i

Stacks

- and the stack now looks like this:



- we note that stacks and lists are isomorphic

Stacks

- for example if we needed a stack of integers we could use our lecture 2 implementation of a single linked list
- `push(i)` is equivalent to `l.cons(i)`
- `i = pop()` is equivalent to `i = l.head(); l = tail();`
- we might be tempted to conclude here :-)
 - however stacks are often used right at the center of many systems and performance can be critical

Stacks

- it should be noted that the operations `push` and `pop` are expected to be used very frequently

Stack definition

examples/c++/stacks/int/v1/stack.h

```
class element
{
public:
    element *next;
    int      data;
};

class stack
{
private:
    element *head_element;
    element *duplicate_elements (element *e);
    element *delete_elements (void);
    friend std::ostream& operator<< (std::ostream& os, const stack& l);
};
```

Stack definition

■ [examples/c++/stacks/int/v1/stack.h](#)

```
public:
    stack (void);
    ~stack (void);
    stack (const stack &from);
    stack& operator= (const stack &from);

    stack empty (void);
    bool is_empty (void);
    stack push (int i);
    int top (void);
    int pop (void);
};
```

Stack definition

- notice its similarity to the single linked list class

Stack definition

examples/c++/stacks/int/v1/stack.cc

```
/*
 * push - push i to stack.
 *       pre-condition:  none.
 *       post-condition: returns the stack which has i at its head
 *                       and the remainder of contents as, stack.
 */

stack stack::push (int i)
{
    element *e = new element;

    e->data = i;
    e->next = head_element;
    head_element = e;
    return *this;
}
```

Stack definition

■ [examples/c++/stacks/int/v1/stack.cc](#)

```
/*
 * pop - opposite of cons. Remove the head value and return it.
 *     pre-condition:  non empty stack.
 *     post-condition: remove and return value from top of stack.
 */

int stack::pop (void)
{
    element *e = head_element;
    int value = e->data;

    assert (! is_empty());
    head_element = head_element->next;
    if (debugging)
        printf ("wanting to delete 0x%p0, e);
    else
        delete e;
    return value;
}
```

Version 2: Stack

- notice that if `push` and `pop` are used many times when we will have many calls to `new` and `delete`
 - these last two functions may be very costly, as they are generic for any data type
 - probably using complex memory management algorithms

- given that `push` and `pop` occur so frequently we will maintain our own free list

Version 2: Stack

■ [examples/c++/stacks/int/v2/stack.h](#)

```
class stack
{
private:
    element *head_element;
    element *free_list;
    element *duplicate_elements (element *e);
    element *delete_elements (element *h);
    friend std::ostream& operator<< (std::ostream& os, const stack& l);
    element *new_element (void);
    void      delete_element (element *e);
```

Version 2: Stack

■ [examples/c++/stacks/int/v2/stack.h](#)

```
public:
    stack (void);
    ~stack (void);
    stack (const stack &from);
    stack& operator= (const stack &from);

    stack empty (void);
    bool is_empty (void);
    stack push (int i);
    int top (void);
    int pop (void);
};
```

Version 2: Stack

<examples/c++/stacks/int/v2/stack.cc>

```
/*
 * delete_elements - delete all elements of stack.
 *                   pre-condition : none.
 *                   post-condition: all elements are deleted
 *                   and zero is returned.
 */
element *stack::delete_elements (element *h)
{
    while (h != 0) {
        element *t = h;
        h = h->next;
        if (debugging)
            printf ("wanting to delete 0x%p\n", t);
        else
            delete t;
    }
    return 0;
}
```

Version 2: Stack

■ [examples/c++/stacks/int/v2/stack.cc](#)

```
/*  
 * delete_element - pre-condition : e, must not be on the stack.  
 *                  post-condition: places, e, onto the free_list.  
 */  
  
void stack::delete_element (element *e)  
{  
    e->next = free_list;  
    free_list = e;  
}
```

Version 2: Stack

examples/c++/stacks/int/v2/stack.cc

```
/*
 * new_element - pre-condition : none.
 *               post-condition: return an element either
 *               from the free_list or from the heap.
 */

element *stack::new_element (void)
{
    element *e;

    if (free_list == 0)
        e = new element;
    else
    {
        e = free_list;
        free_list = free_list->next;
    }
    return e;
}
```


Version 2: Stack

examples/c++/stacks/int/v2/stack.cc

```
/*  
 * push - push i to stack.  
 *       pre-condition:  none.  
 *       post-condition: returns the stack which has i at its head  
 *                       and the remainder of contents as, stack.  
 */  
  
stack stack::push (int i)  
{  
    element *e = new_element ();  
  
    e->data = i;  
    e->next = head_element;  
    head_element = e;  
    return *this;  
}
```

Version 2: Stack

■ <examples/c++/stacks/int/v2/stack.cc>

```
/*
 * pop - opposite of cons. Remove the head value and return it.
 *     pre-condition:  non empty stack.
 *     post-condition: remove and return value from top of stack.
 */

int stack::pop (void)
{
    element *e = head_element;
    int value = e->data;

    assert (! is_empty());
    head_element = head_element->next;
    if (debugging)
        printf ("wanting to delete 0x%p\n", e);
    else
        delete_element (e);
    return value;
}
```

Version 2: destructor

examples/c++/stacks/int/v2/stack.cc

```
/*  
 * ~stack - destructor, releases the memory attached to the stack.  
 *           pre-condition:    none.  
 *           post-condition:   stack is empty.  
 */  
  
stack::~~stack (void)  
{  
    head_element = delete_elements (head_element);  
    free_list = delete_elements (free_list);  
}
```

Version 2: copy

examples/c++/stacks/int/v2/stack.cc

```
/*
 * copy operator - redefine the copy operator.
 *               pre-condition : a stack.
 *               post-condition: a copy of the stack and its elements.
 */

stack::stack (const stack &from)
{
    head_element = duplicate_elements (from.head_element);
    free_list = 0;
}
```

Version 2: assignment

examples/c++/stacks/int/v2/stack.cc

```
/*
 * operator= - redefine the assignment operator.
 *             pre-condition : a stack.
 *             post-condition: a copy of the stack and its elements.
 *                               We delete 'this' stacks elements.
 */

stack& stack::operator= (const stack &from)
{
    if (this->head_element == from.head_element)
        return *this;

    head_element = delete_elements (head_element);
    head_element = duplicate_elements (from.head_element);
    free_list = 0;
}
```

Version 2: constructor

examples/c++/stacks/int/v2/stack.cc

```
/*  
 * stack - constructor, builds an empty stack.  
 *       pre-condition:  none.  
 *       post-condition: stack is created and is empty.  
 *                       free_list is empty.  
 */  
  
stack::stack (void)  
    : head_element(0), free_list(0)  
{  
}
```