CSCI 2270
Data Structures and Algorithms
Lecture 18

Elizabeth White
elizabeth.white@colorado.edu
Office hours: ECCS 128
Wed 1-2pm
Fri 2-3pm

Administrivia

Thursday 2-3 office hours have moved to Friday 2-3

HW2 will post on Tuesday (part 1)

linked list implementation of a biiig integer in any base

Lab this week: review for test

One bug in doubly linked list code: posted correction

Binary search trees

insert

size

remove

remove_max

Exam programming questions…

Be able to modify the singly linked list code from hw1

Be able to modify the doubly linked list code from class

Be able to modify any of the stack and queue code from class

Get the frequency of an item in a binary tree

Get the frequency of an item in a binary search tree

Write contains for a binary tree

Write contains for a binary search tree

Reverse a binary tree (mirror image)

Make a binary search tree from a binary tree

Compute the height of a binary tree

Structs to classes

You know all about structs, now.

```
        struct binary_tree_node
        {
                int data;
                binary_tree_node *left;
                binary_tree_node *right;
        };
void print(binary_tree_node* node_ptr, unsigned int depth);
void tree_clear(binary_tree_node*& root_ptr);
binary_tree_node* tree_copy(const binary_tree_node*
root_ptr);
unsigned int tree_size(const binary_tree_node* node_ptr);
```

Structs to classes

Structs stick all the relevant variables together:

```
struct binary_tree_node
{
        int data;
        binary_tree_node *left;
        binary_tree_node *right;
};
```

And then we write methods that pass the structs around and change these variables:

```
void print(binary_tree_node* node_ptr, unsigned int depth);
void tree_clear(binary_tree_node*& root_ptr);
```

Classes

Classes stick all the relevant variables together and then stick the methods in there too.

Variables are generally kept private (except to class functions)

    Protects data from evil users: no access

Functions are generally left public so we can call them

    Many function calls are automatic

big_number.h:

Classes

```
class big_number
{
	public:
		// constructors (work like init() methods)
		big_number();
		big_number(int i);
		big_number(const big_number& m);
		big_number(const string& s, unsigned int base);
		// assignment operator
		big_number& operator=(const big_number& m);
		// destructor (works like destr() method)
		~big_number();				…
```

Classes

```
class big_number
{          ....
           private:
                      node* head_ptr;
                      node* tail_ptr;
                      unsigned int digits;
                      bool positive;
                      unsigned int base;

                      // helper functions can go here

};
```

Constructor lets us initialize new big_numbers (like init() methods)

```
        big_number();
        big_number(int i);
        big_number(const big_number& m);
        big_number(const string& s, unsigned int base);

int main()
{
        big_number q;
        big_number r(56);
        big_number s(r);
        big_number t(“5643637729127639378738789389”, 10);
}
```

Destructor lets us avoid leaks (like destr() method)

```
        ~big_number();


int main()
{
        big_number q;
        big_number r(56);
        big_number s(r);
        big_number t(“5643637729127639378738789389”, 10);
}       <-- destructor runs automatically here and frees up the
memory
```

Assignment operator

```
        big_number& operator=(const big_number& m);

int main()
{
        big_number q;
        big_number r(56);
        big_number s(r);
        big_number t(“5643637729127639378738789389”, 10);
        q = t;    <-- assignment runs here
}
```

Input/output

```
        friend std::ostream& operator<<(std::ostream& os,
                const big_number& bignum);
        friend std::istream& operator>>(std::istream& is,
                big_number& bignum);


int main()
{
        big_number r(56);
        cout << r << endl;       <-- output operator
        big_number x;
        cin >> x;                <-- input operator
}
```

Comparisons

```
        friend bool operator==(const big_number& a,
                const big_number& b);
        friend bool operator!=(const big_number& a,
                const big_number& b);
int main()
{
        big_number r(56); big_number s(78);
        if (r == s)
                cout << r << “ and “ << s << “ are equal “ << endl;
        if (r != s)
                cout << r << “ and “ << s << “ are not equal “ <<
                        endl;
}
```

Members and non-members

Member functions: non-friend functions defined inside class brackets {} in big_number.h

Each of these is called by a particular big_number in your program:

```
big_number p(58);           // run int constructor on p
big_number q("1", 10);      // run string constr. on q
q += p;                     // run += on q
++p;                        // run ++ on p
```

Members and non-members

Nonmember functions: marked friend in big_number.h

None of these are called by a particular big_number in your program:

```
big_number p(58);              // run int constructor on p
big_number q(“1”, 10);         // run string constr. on q
big_number r;
r = q + p;                     // add q + p
if (q == r)                    // compare q and r
        cout << “q and r are equal” << endl;
```

These non-members don’t get called via a big number in the program. We just pass big_numbers in to them.

Members and this

Each of these is called by a particular big_number in your program.

Each of these members has the address of the big_number calling it. That address is called *this*.

```
big_number p(58);           // run int constructor on p
big_number q(“1”, 10);      // run string constr. on q
q += p;                     // this is address of q
++p;                        // this is address of p
```

Members and *this

Each of these is called by a particular big_number in your program.

Each of these members has the address of the big_number calling it. That address is called *this*.

Some of these members want to return a big_number by reference (operator ++, for instance). In this case, we dereference the this pointer (*this) to get the big_number calling it instead:

```
            return *this;
```

Recall the dangling reference

```
int& no_no_nanette() {
        int answer = 9;
        return answer;
}

int main {
        cout << no_no_nanette << endl;        <-- seg fault
}
```

Why doesn’t this dangle?

```
big-number& operator =(const big_number& m) {
        …
        return *this;
}

int main {
        big_number a(98);
        big_number b;
        b = a;                          <-- safe; b is still alive here
        cout << b << endl;
}
```

Why is this nice?

```
big-number& operator =(const big_number& m) {
        …
        return *this;
}
int main {
        big_number a(98);
        big_number b;
        big_number c(“1026473838392839239”, 10);
        big_number d(17);
        a = b = c = d;           <-- chained assignment
        cout << a << endl;       <-- prints 17
}
```

What does operator = do?

Not leak memory: clear any linked list in *this already

```
this->clear_list(head_ptr, tail_ptr);
(*this).clear_list(head_ptr, tail_ptr);
clear_list(head_ptr, tail_ptr);
```

Then copy new list, and set digits, positive, and base

Pitfall: What does operator = do now?

Not leak memory: clear any linked list in *this already

```
this->clear_list(head_ptr, tail_ptr);
(*this).clear_list(head_ptr, tail_ptr);
clear_list(head_ptr, tail_ptr);
```

Then copy new list, and set digits, positive, and base

What happens if we self-assign with this operator;

```
big_number a(87);
a = a;
```

Pitfall: What does operator = do now?

Self assignment check

```
        if (this == &anothernumber)   // address comparison
```

If this is true, we have 2 big_numbers in the same pair of underpants—and no work to do. Just return *this in this case.

```
        else
        {
                // copy all the things
        }
```