A simple program involving integers

```c
int main()
{   int a = 17;
    int b = 33;
    int c = a + b + 50;
    printf("%d\n", c);
    return 0;
}
```
Big, really big integers

“496982034007341215862794536196784621078462756513297”

Goal: write a library to deal with large numbers.

Concerns

- require dynamic memory management.
- need functions creating, destroying, reading, printing, assigning, and basic arithmetic.
- avoid naming conflicts, e.g. Create_BigInt(), Print_BigInt().
- programmers will need to know these names and the rules for calling them.
- programmers will need to explicitly initialize and destroy big numbers.
- will need to be careful when combining big numbers with other data types, like int.
- will have to “clean-up” unused memory by local big numbers when exiting functions.

A simple program involving BigInts

```c
#include "bigint.h"

int main()
{
    BigInt a, b, c;
    BigInt t;

    Create_BigInt( &a, "28587365452419232");
    Create_BigInt( &b, "6948672303927125");

    Assign_BigInt( &c, Add_BigInt(a, b));
    Convert_Int_to_BigInt(&t, 50);
    Assign_BigInt( &c, Add_BigInt(c, t));

    Print_BigInt(c);
    Destroy_BigInt(&a);
    Destroy_BigInt(&b);
    Destroy_BigInt(&c);
    Destroy_BigInt(&t);

    return 0;
}
```
A simpler C++ program involving BigInts

```cpp
int main()
{
    BigInt a = "29587365452419232";
    BigInt b = "6498672303927125";
    BigInt c = a + b + 50;
    c.print();
    return 0;
}
```

The C++ compiler needed to know how to:

- create new instances of BigInt.
- convert character strings and integers to BigInts.
- initialize the value of one BigInt with another.
- add two BigInts together.
- print BigInts.
- destroy BigInts when no longer needed.

Where did this information come from?
The BigInt C++ Class declaration

class BigInt  
{  
  private:
    // "private" variables are not accessible from the outside
    char *nd;
    int ndigits;

  public:
    // "public" variables and functions
    // can be accessed from the outside
    BigInt(const char *s);
    BigInt(int);
    // these are "constructors"; they define ways of initializing BigInts.
    // The first one builds BigInts from strings, the second one from ints.

    BigInt operator+(const BigInt &A);  // this is how we add BigInts
    BigInt& operator=(const BigInt &A); // this is how we assign BigInts

    void print();  // this is how we print BigInts
    'BigInt();   // this is how we destroy local BigInts  // when exiting functions
}

Integrating a BigInt library

- **Implementation**: BigInt.cc  
- **Specification**: BigInt.h  
- **Application**: app.cc
Some common objects

- complex numbers
- vectors & matrices
- strings
- sets
- lists
- algebraic groups, rings, fields
- and, yes, even stacks...

C++ stack objects

C++ classes are essentially C struct's bundled together with the corresponding functions that modify that data structure.

For example, here is the declaration for the stack example we used before, but now reformulated as a C++ class. This would typically be contained in its header file, e.g. “dstack.h”:

```cpp
// dstack.h -- Dynamic stack (DStack) declaration and function prototypes.
//
// Functions:
//
// S.init(int N) initialized stack of size N
// S.push(val) push new value on top of stack
// S.pop() returns (and removes) top value of stack
// S.num_items() returns number of items currently on stack
// S.size() returns max number of items stack can hold
// S.full() returns 1 if stack is full, 0 otherwise
// S.empty() returns 1 if stack is empty, 0 otherwise
// S.print() print stack contents

class DStack
{
private:
      float *bottom;
      float *top;
      int size;

public:
      DStack(int size=20);
      void push(float val);
      int num_items() const;
      float pop();
      int full() const;
      int empty() const;
      void print() const;
};
```
**Things to note...**

- A C++ class is basically a C struct that also allows functions as elements.
- Note that functions and variables are bundled together in one package.
- The `private` and `public` keywords clearly denote what items can and can’t be modified by external programs.
- A common convention (although completely optional) is to suffix the private variable names with an underscore ("_") to help identify them.
- Member functions are accessed just like a struct’s element:
  ```cpp
  A.push(1.3);        // used to be: push(&A, 1.3);
  x = A.pop();        // used to be: x = pop(&A);
  if (!A.full()) A.push(2.9);   // used to be: if (!full(&A)) push(&A, 2.9);
  ```
- Also note that the "}" closing off the class declaration must be followed by a "/}".

---

**C++ classes: how they’re used in programs**

```cpp
#include <iostream.h>
#include "dstack.h"

int main()
{
  DStack S[4];
  S.print();
  cout << "\n";
  S.push(2.31);
  S.push(1.19);
  S.push(6.78);
  S.push(5.54);
  S.print();
  cout << "\n";
  if (!S.full()) S.push(8.7); // this should do nothing, as stack is already full.
  S.print();
  cout << "\n";
  cout << "Popped value is: " << S.pop() << "\n";
  S.print();
  cout << "\n";
  S.push(S.pop() + S.pop());
  cout << "Replace top two items with their sum: \n";
  S.print();
  cout << "\n";
  S.pop();
  S.pop();
  S.print();
  cout << "\n";
  if (!S.empty()) S.pop(); // this should also do nothing.
  if (S.num_items() != 0)
  {
    cout << "Error: Stack is corrupt!\n";
  }
  return 0; // destructor for S automatically called
}
```
Program Output

Stack currently holds 0 items:

Stack currently holds 4 items: 2.31 1.19 6.78 0.54
Stack currently holds 4 items: 2.31 1.19 6.78 0.54
Popped value is: 0.54
Stack currently holds 3 items: 2.31 1.19 6.78
Replace top two items with their sum:
Stack currently holds 2 items: 2.31 7.97
Stack currently holds 0 items:

Things to note...

- Notice we’ve separated the *interface* of a DStack from its implementation.
- All DStacks are automatically initialized. There is no way to accidentally access an uninitialized stack.
- Notice also that any dynamic memory used by stacks is automatically freed by calling the destructor ~DStack().
- A DStack can still be treated like any basic data structure, e.g. being passed to and returned from functions.
C++ class implementation

What does the DStack implementation look like?

```cpp
#include<iostream.h>
#include "dstack.h"

DStack::DStack(int N) // how to initialize a stack with
{                      // N items.
    bottom_ = new float[N];
    top_ = bottom_;
    size_ = N;
}

DStack::~DStack()     // how to reclaim memory from local
{                      // stacks when exiting functions
    delete [] bottom_;
}

int DStack::num_items() const // number of items currently in stack
{                           //
    return (top_ - bottom_);  //
}

void DStack::push(float val) // push a new value
{                           //
    *top_ = val;
    top_++;
}

float DStack::pop()     // pop value from top
{                        //
    top_--;
    return *top_;
}

int DStack::full() const // 1 if full, 0 otherwise
{                       //
    return (num_items() >= size_);
}

int DStack::empty() const // 1 if empty, 0 otherwise
{                         //
    return (num_items() < 0);
}

void DStack::print() const
{                               //
    cout << "Stack currently holds " << num_items() << " items: " ;
    for (float *element=bottom_; element!=top_; element++)
    {
        cout << " " << *element;
    }
    cout << "\n";
}                                //
```
How we’d like to use complex numbers

```cpp
#include <iostream.h>
#include "complex.h"

int main()
{
    Complex u(1.1, 3.9);
    Complex v(8.8, 5.4);
    Complex w, z; // default to (0.0, 0.0)
    w = u + v;
    cout << "Default value: " << z << ".\n";
    cout << "Sum of " << u << " and " << v << " is " << w << ".\n";
}
```

Will produce

Sum of (1.1 + 3.9i) and (8.8 + 5.4i) is (9.9 + 9.3i).

Implementation of complex numbers

Declaration in “complex.h”:

```cpp
class Complex
{
    private:
        double real_; 
        double img_; 

    public:
        Complex();
        Complex(double real, double img);
        double real() const { return real_; }
        double img() const { return img_; }
        Complex conjugate() const;
        double norm() const;
        Complex operator+(const Complex& x) const;
        Complex operator-(const Complex& x);
        ~Complex();
};
```

- describing body of function in declaration (e.g. as in real()) means that that function is declared inline.
- functions defined const (e.g. norm(), operator+) mean that calling them does not modify the object.
- the destructor ~Complex() in this case, need not do anything, since the private data members do not utilize dynamic memory.
Complex number class: implementation

#include <math.h>
#include <iostream.h>
#include "complex.h"

Complex::Complex(double real, double img)
{
    real_ = real;
    img_ = img;
}

Complex::Complex()
{
    real_ = 0.0;
    img_ = 0.0;
}

Complex Complex::conjugate() const
{
    Complex t(real_, -img_);
    return t;
}

double Complex::norm() const
{
    return (sqrt(real_*real_ + img_*img_));
}

---

Complex number class (cont’d)

Complex Complex::operator*(const Complex &b) const
{
    return Complex( real_ * b.real_ , img_ * b.img_);
}

Complex & Complex::operator=(const Complex &u)
{
    real_ = u.real_;
    img_ = u.img_;
    return *this;
}

ostream & operator<<(ostream &s, const Complex &u)
{
    s << "(" << u.real_ << " + " << u.img_ << "i\n    return s;
}

Complex::Complex()
{
    // why isn't real_ and img_ declared public and accessed directly. That is, why write c.img() rather than c.img_ ?

    // functions that were declared const in header, must also be declared const where implemented.
Programming Tips

- always define a default constructor and default destructor, even if you're not using dynamic memory! (You may later at some point...)
- use ifndef HEADER_FILENAME macros to avoid including the same include file more than once.

```c
#ifndef STACK_H
#define STACK_H

// declarations of DStack goes here
#endif
```

- use a naming convention to help identify private variables within class codes (e.g. appending an underscore to their name.)

Homework #3

1. Complete the C++ class for the complex numbers.

   Provide the following:
   - constructors (default plus at least one other), and destructor
   - operator definitions as discussed in class (+, -, *, =)
   - some basic functionality (real(), imag(), conj(), norm(), etc.)
   - a print function
   - a short testing program demonstrating class usage

2. Recode your last stack example (Homework #2) as a C++ object. Implement the declaration in dstack.h, the implementation in dstack.cc, and a simple test driver in tester.cc. Use the code examples in this lecture as a guide.