CSE 143
SUMMER 1998

Canonical Form
[Section 7.8]

Introduction

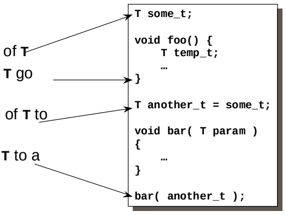
```
/canonical/ <<k@"nQnIk(@)l>> adj. & n.

adj.
2. authoritative, standard, accepted.
```

- C++ is a huge language
- There are many ways to solve a given problem
- Some techniques are better than others
  - Some become the "right" way
  - Some become expected, almost required
- Canonical techniques allow us to think less!

Operations on (Almost) All Types

- For any type **T**, you should be able to:
  - Declare an instance of **T**
  - Have an instance of **T** go out of scope
  - Assign one instance of **T** to another
  - Pass an instance of **T** to a function



```
T some_t;

void foo() {
    T temp_t;
    …
}

T another_t = some_t;

void bar( T param )
{
    …
}

bar( another_t );
```

Default Operations

- C++ generates defaults for all of these operations if not given explicitly
- These defaults are not always what you want!
  - Especially with data structures that use dynamic memory

Declaring an Instance

- Recall
  - The default declaration invokes the default constructor
  - If class contains no constructors, a default constructor is generated automatically
    - Call default constructor on all members (recursion!)
- Not every class needs a default constructor
  - But if one is needed, it should probably be written explicitly
    - E.g, **IntStack** should initialize its pointer member

Going Out Of Scope

- What happens every time you call **blah()**?

```
class IntStack
{
public:
     IntStack();

     void push( int item );

private:
     IntStackNode *ptop;
};

void blah()
{
     IntStack stack;
     stack.push( 14 );
     stack.push( -99 );
}
```

Destructors

- The opposite of a constructor
  - Gets called whenever an instance is destroyed
    - When an automatic variable goes out of scope
    - When a dynamic object is **`delete`**d
  - Used to clean up any resources associated with that object
    - **`delete`** any dynamic memory allocated by the object
- If no explicit destructor supplied, compiler generates a default one
  - Call destructors of all members

Using Destructors

- Destructor is (almost) never called explicitly
- Called whenever automatic instance goes out of scope

```
void blah()
{
    IntStack stack;
    stack.push( 14 );
    // implicit call to IntStack’s destructor
}
```

- Called whenever dynamic instance is **`delete`**d

```
…
IntStack *pstack = new IntStack;
pstack->push( 19 ); pstack->push( 175 );
delete pstack; // Implicit call to IntStack's destructor
…
```

Writing Destructors

- Similar to constructor
  - Name is ~ followed by class name
  - Not allowed to take any arguments
  - No declared return type, not even **`void`**
- Usually a good idea to provide an explicit destructor
  - Extremely important for dynamic data structures

```
class IntStack
{
public:
    IntStack();
    ~IntStack();
    …

private:
    IntStackNode *ptop;
};

…
IntStack::~IntStack()
{
    delAll();
}
…
```

Assigning Instances

- If no explicit assignment operator given, one is generated automatically
  - Assign each data member
- What happens here?

```
int main( void )
{
    IntStack s1;
    IntStack s2;

    s1.push( 10 );
    s1.push( 20 );

    s2 = s1;

    cout << s2.pop() << endl;
    cout << s1.pop() << endl;
}
```

Shallow vs. Deep Copy

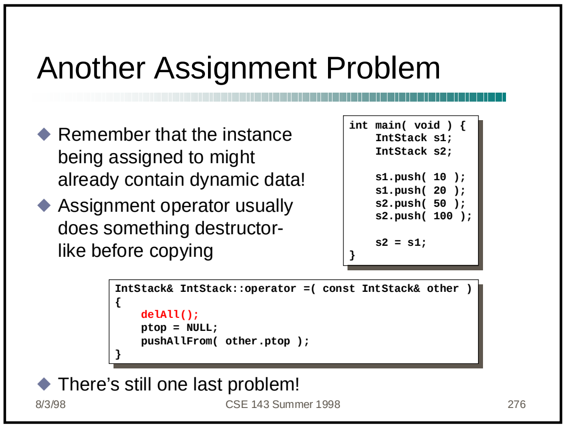
- Default assignment operator is a **shallow** copy
  - Just copy over every data member
  - But pointers are copied across without copying the actual dynamic data
  - This leads to sharing, which is probably not what you want
- For most dynamic data structures, want a **deep** copy
  - Copy complete set of dynamically allocated data
  - So overload the assignment operator!

Writing Assignment Operators

```
IntStack& IntStack::operator =( const IntStack& other )
{
    ptop = NULL;
    pushAllFrom( other.ptop );
}
```

- The **`const`** here means that we promise not to modify **`other`** while making the copy
- Assignment operator returns a **`IntStack&`** so that it can support chained assignment

```
IntStack s1, s2, s3;
s1.push( 10 );
s2 = s3 = s1;          // Equivalent to s3 = s1; s2 = s3;
```



Another Assignment Problem

- Remember that the instance being assigned to might already contain dynamic data!
- Assignment operator usually does something destructor-like before copying

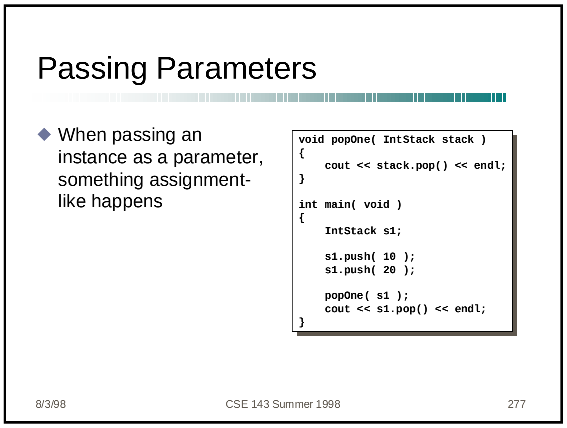
```
int main( void ) {
    IntStack s1;
    IntStack s2;

    s1.push( 10 );
    s1.push( 20 );
    s2.push( 50 );
    s2.push( 100 );

    s2 = s1;
}
```

```
IntStack& IntStack::operator =( const IntStack& other )
{
    delAll();
    ptop = NULL;
    pushAllFrom( other.ptop );
}
```

- There’s still one last problem!



Passing Parameters

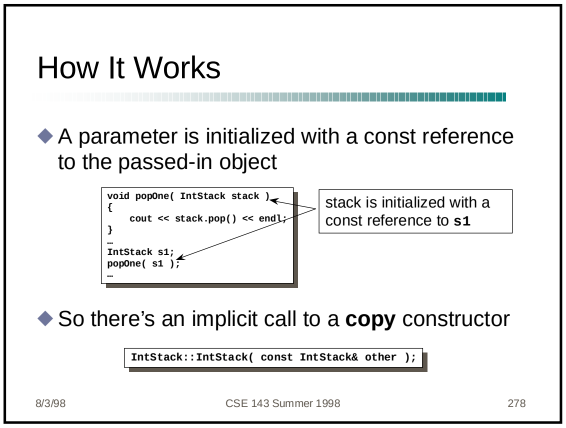
- When passing an instance as a parameter, something assignment-like happens

```
void popOne( IntStack stack )
{
    cout << stack.pop() << endl;
}

int main( void )
{
    IntStack s1;

    s1.push( 10 );
    s1.push( 20 );

    popOne( s1 );
    cout << s1.pop() << endl;
}
```



How It Works

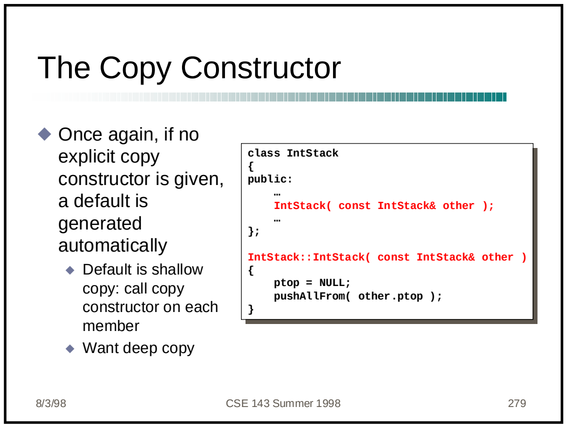
- A parameter is initialized with a const reference to the passed-in object

```
void popOne( IntStack stack )
{
    cout << stack.pop() << endl;
}
…
IntStack s1;
popOne( s1 );
…
```

stack is initialized with a const reference to **s1**

- So there’s an implicit call to a **copy** constructor

```
IntStack::IntStack( const IntStack& other );
```

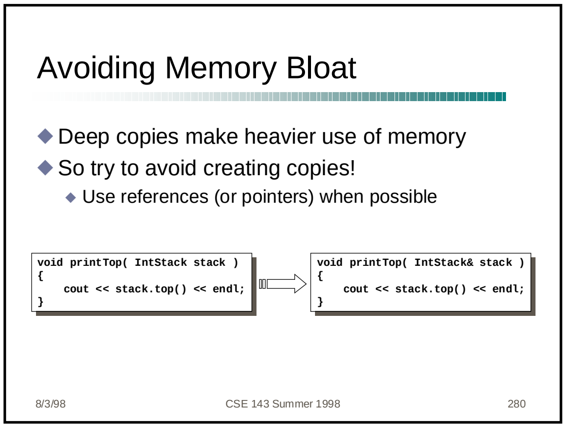


The Copy Constructor

- Once again, if no explicit copy constructor is given, a default is generated automatically
  - Default is shallow copy: call copy constructor on each member
  - Want deep copy

```
class IntStack
{
public:
    …
    IntStack( const IntStack& other );
    …
};

IntStack::IntStack( const IntStack& other )
{
    ptop = NULL;
    pushAllFrom( other.ptop );
}
```

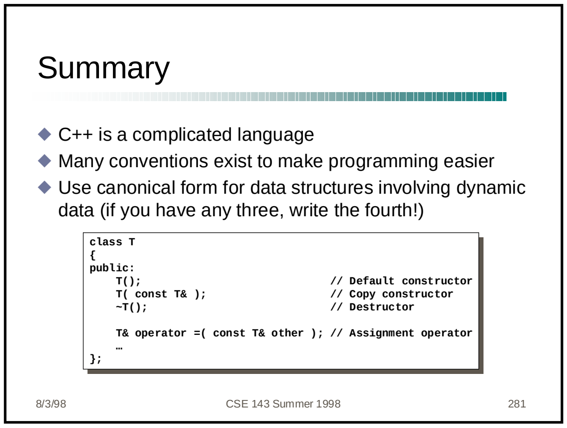


Avoiding Memory Bloat

- Deep copies make heavier use of memory
- So try to avoid creating copies!
  - Use references (or pointers) when possible

```
void printTop( IntStack stack )
{
    cout << stack.top() << endl;
}
```

```
void printTop( IntStack& stack )
{
    cout << stack.top() << endl;
}
```



Summary

- C++ is a complicated language
- Many conventions exist to make programming easier
- Use canonical form for data structures involving dynamic data (if you have any three, write the fourth!)

```
class T
{
public:
    T();                                // Default constructor
    T( const T& );                      // Copy constructor
    ~T();                               // Destructor

    T& operator =( const T& other ); // Assignment operator
    …
};
```