Life Cycle Methods

CIS-3012, C++ Programming

Vermont State University

Peter Chapin

External Resources

- Many classes manages resources that exist outside the class objects themselves…
  - … **memory** is the most common such resource, but…
  - … open file handles…
  - … open network connections…
  - … handles to graphical contexts…
  - … open hardware devices (serial ports, printers, etc.)…
  - There are other examples.

Garbage Collection

- Many languages automatically reclaim dynamically allocated memory that can no longer be reached or used.
  - The runtime system (i.e., “garbage collector”) automatically locates and recycles that unreachable memory (i.e., “garbage”).
- C++ implementations typically do not include garbage collectors.
  - (In theory they could, and some do, but it is rare).
- Garbage collection is great, but there are two problems with it…

Problems with Garbage Collection

- It greatly complicates the runtime system.
  - For example, in Java garbage collection is done by the Java Virtual Machine, which is a huge body of software.
  - For some small-scale, highly constrained embedded devices, there just aren't the resources (memory, processor power) to run a garbage collector.
- Classic garbage collection is great for memory, but it does nothing about all the other resources the program is using!
  - You might not be leaking memory, but are you leaking open file handles??
  - Some programming languages have glued-on features to deal with this.
  - C++ has a comprehensive solution.

The Destructor

- A class's *destructor* is a method that releases any external resources held by the object.
- It is automatically called by the compiler when appropriate.
  - … when a local variable disappears at the end of its scope.
  - … when a function parameter disappears when a function returns.
  - … when a global variable disappears when the program ends.
- It is possible, but exceedingly rare to call the destructor manually.

Class BigInteger

```
class BigInteger {

    // The default constructor.
    BigInteger( );

    // The destructor.
   ~BigInteger( );

};
```

In general, there can be many constructors with various parameters

There is only one destructor, and it is always parameter-less

Destructor Definition

No return type, like constructors

No parameters

```
BigInteger::~BigInteger( )
{
    delete [ ] digits;
}
```

Release the dynamically allocated array

Invariants?

- Since the destructor is only called (by the compiler) when the object is disappearing, it need not leave the object in a sensible state.
  - Destructors only must worry about releasing external resources.
  - It is okay for invariants to be violated.
  - It is ~~impossible~~ difficult to even access an object after destruction.

Constant-ness?

- During construction, the members of a const object are not considered to be const.
  - Objects are changing during their initialization even if they are ultimately constants after that point.
- Also… during destruction, the members of a const object are again not considered to be const.
  - Even constant objects need to have their resources released!

Resource Management

- Objects acquire resources during construction
  - … or during their lifetimes.
- Objects release resource during destruction.
- Examples:
  - Allocate memory in constructor / free memory in destructor.
  - Open file in constructor / close file in destructor.
  - Connect to server in constructor / disconnect from server in destructor.
  - Configure serial port in constructor / restore port to original configuration in destructor.
  - Open window in constructor / close window in destructor.

Exceptions and Destructors

```
void f( )
{
    string some_string{ etc };
    …
    bad_thing( );
    …
}
```

The function `bad_thing( )` throws an exception…
The dynamic memory held by `some_string` is **still** reclaimed!

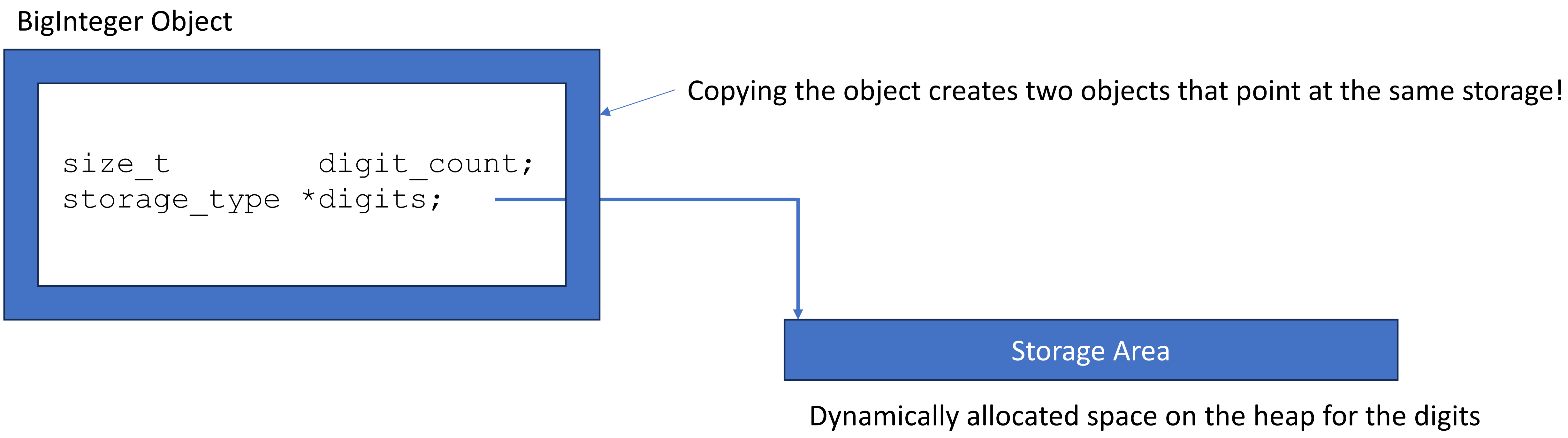
Exceptions and Destructors

- When an exception propagates to the callers…
  - The *destructors of all* *fully constructed* *local objects in each abandoned context are automatically called*.
- Thus, as an exception *unwinds* the call stack looking for a handler…
  - Memory is automatically reclaimed…
  - Files are automatically closed…
  - Network connections are automatically disconnected…
  - Hardware configurations are automatically restored…
- All provided *you* write destructors appropriately!

RAI

- **<u>R</u>**esource **<u>A</u>**cquisition is **<u>I</u>**nitialization
  - An idiom whereby resources are acquired in constructors (during initialization)
  - … and then released in destructors.
- RAI is extremely common in C++ class design. You will see it everywhere.
  - This is why you don't have to worry about deallocating the memory held by `std::string` objects or `std::vector` objects. Their destructors do it.
  - It is also used for locking in multi-threaded applications (constructor acquires lock, destructor releases lock… even when an exception is thrown).
  - It is used by iostreams to ensure files always get closed.

What About Copying?



The Problem With Copying

```
size_t        digit_count;
storage_type *digits;
```

```
size_t        digit_count;
storage_type *digits;
```

Storage Area

- Changes to one object are seen by the other. This violates expected value semantics.
- If one object reallocates, the other object can't find the new allocation.
- If one object deletes the allocation (i.e., is destroyed), the other object is left pointing at unallocated memory.

Copy Management

```
// Copy constructor
BigInteger( const BigInteger &other );

// Copy-assignment operator
BigInteger &operator=( const BigInteger &other );
```

- Normally these are automatically generated by the compiler.
- You can define them yourself to "do the right thing."
- Why two?
  - The copy constructor is used to *initialize* an object with a copy of `other`.
  - The copy-assignment operator is used to *assign* a copy of `other` to an already existing (and already initialized) object.

Copy Constructor

- The copy constructor is an ordinary constructor that can be called using a single argument of the class.
  - So, additional parameters with default values would be okay.
- It is used automatically by the compiler whenever you try to initialize an object by copying some other object of the same type.
  - In declarations: `BigInteger x; BigInteger y{ x };`
  - When passing a function argument by value: `void f( BigInteger value); f( x )`. Notice here that `f` is declared as taking a `BigInteger` by value, not by reference (as might often be the case).
  - When returning from a function: `BigInteger g( ); x = y + g( )`. The return value is copy-constructed to a temporary that is added to `y`.

BigInteger Copy Constructor

```
BigInteger::BigInteger( const BigInteger &other )
{
    // Allocate a new storage area to hold the copy of the digits.
    digits = new storage_type[ other.digit_count ];

    // Copy the digits from the other object using C’s memcpy for speed.
    memcpy( digits, other.digits, other.digit_count * sizeof( storage_type ) );

    // Don’t forget to make a copy of the other object’s digit count!
    digit_count = other.digit_count;
}
```

This implementation isn’t quite right because it doesn’t deal with the case when `other` is zero

I’m ignoring that for now to avoid distraction

Copy-Assignment: Why?

- If copy constructors can copy, why do we need a separate copy assignment operator?
- *Initialization and assignment are not the same*!
  - Initialization *gives an object its first value*.
  - Assignment *overwrites an existing value* with a new value.
- Thus…
  - When doing copy construction, there is no need to clean up the target object.
  - When doing assignment there is such a need.
  - Assignment is somewhat like destruction + copy construction…
  - … the compiler does *not* automatically generate that, however!

Initialization vs Assignment

- People are sometimes confused about the distinction because for simple types there is no effective difference.
  - Also, in languages that only handle complicated types by reference (e.g., Java), the matter doesn't come up because the references themselves are simple.

```
int x;
…
x = 42;
```

"default construct" x
(which does nothing for type int)

Assign to x. The old value is removed by simply overwriting it with the new value.

```
int x = 42;
```

"copy construct" x
(there is no "old value" to remove.

```
int x{ 42 };
```

Same as above using uniform initialization syntax

Initialization vs Assignment

- For complicated types, there is a big difference!

Default construct x
(which is non-trivial)

```
BigInteger x;
…
x = 42;
```

Assign to x. This first requires that the storage previously allocated for x be removed. Then, new storage is allocated for the copy.

```
BigInteger x{ 42 };
```

Copy construct x. Storage is allocated
To hold a copy of the initializer.
(there is no ”old value” to remove.

Initialization is potentially <u>faster</u>!
This is because there is no need to clean up the target object first
… and no need to execute a pointless default constructor.

C++ Allows Declarations Anywhere

- This is not just a convenience feature.
  - BP: Always initialize (i.e., call an appropriate constructor on) an object when it is declared. *Instead of declaring it first and assigning to it later*.
  - If you don't know the initializer (i.e., constructor arguments) yet, move the declaration to a place where you do.
  - Consider declaring the object `const` if possible.
- This is normal for functional languages where objects are all immutable and can't be assigned a value after initialization (i.e., construction).
- *There are places where exceptions to this idea are appropriate*.

BigInteger Copy-Assignment Operator (v1)

```
BigInteger &BigInteger::operator=( const BigInteger &other )
{
    // Clean up target object (*this).
    delete [ ] digits;

    // Copy `other` value.
    digits = new storage_type[ other.digit_count ];
    memcpy( digits, other.digits, other.digit_count * sizeof( storage_type ) );
    digit_count = other.digit_count;

    // Return a reference to the target object.
    return *this;
}
```

This implementation has some problems

(other than the fact that it also doesn't handle zero properly)

Problem #1: Exception Safety

- If an exception is thrown during the execution of a method, in what state will that leave the object?
  - **Strong Safety**: The object retains its original value and continues to work properly. *Any effect the method had before the exception is thrown is undone*.
  - **Basic Safety**: The object's value may have been changed, but *the object continues to work properly* (all invariants remain satisfied).
  - **No Safety**: The object is corrupted and unusable. However, *the object remains destructible* (meaning, the destructor will execute without crashing and recover all resources as usual)
  - **There Be Dragons**: The object is no longer destructible. Do not go there!!

Evaluating Exception Safety

- First… which operations in the method might throw?
  - For BigInteger's copy-assignment operator (v1)…
  - … the only operation that might throw is the dynamic memory allocation.
  - It might throw `std::bad_alloc` if there is insufficient memory.
- Now, suppose it does throw. Where does that leave the object?
  - The `digits` array has just been deleted (deallocated)
  - The `digit_count` member continues to have its original value.
  - The invariant is violated!
- *It's worse*
  - The object is not destructible! The `digits` array will be double-deleted.

BigInteger Copy-Assignment Operator (v2)

```
BigInteger &BigInteger::operator=( const BigInteger &other )
{
    // Try the allocation first. If this throws there is no other effect.
    storage_type *temp = new storage_type[ other.digit_count ];

    // Nothing below this point can throw.

    // Clean up target object (*this).
    delete [ ] digits;

    // Copy `other` value.
    digits = temp;
    memcpy( digits, other.digits, other.digit_count * sizeof( storage_type ) );
    digit_count = other.digit_count;

    // Return a reference to the target object.
    return *this;
}
```

Exception Safety?

- In version 2, the allocation (that might throw) is done first.
  - If an exception is thrown, the object is unchanged: **we have strong exception safety**!
- The downside:
  - For a short time, we need enough memory to make a copy of the other object's digits while at the same time hold on to the memory for the target object's digits.
  - Thus, the exception safety has memory costs
  - No big deal if the numbers have only a few digits. What if they have billions?
- Conclusion: *You can't have it all!*

Problem #2: Self-Assignment

- Normally the copy-assignment operator needs to protect itself from the possibility that an object is being assigned to itself.
- The BigInteger copy-assignment operator v1 fails spectacularly in that case.
  - It deletes `digits` before it copies `other.digits`. If `other` is the same object, it will be trying to copy a deleted array.
- What about v2?
  - Hint: It has the same problem.
- We could rearrange the code to deal with this too, but first… why should we even care about this?

Self-Assignment

- Self-assignment looks like this (for integers):

```
int x;
int *p = &x; // p points at x
…
x = *p;      // Assigns x to itself.
```

- Here is a more compelling example:

```
int array[128];
…
// Copy element at position k to every array location.
for( int i = 0; i < 128; ++i ) {
    array[i] = array[k];   // When i == k this assigns array[k] to itself.
}
```

BigInteger Copy-Assignment Operator (v3)

```
BigInteger &BigInteger::operator=( const BigInteger &other )
{
    // Boiler plate for avoiding self-assignment.
    if( this != &other ) {

        storage_type *temp = new storage_type[ other.digit_count ];

        // Clean up target object (*this).
        delete [ ] digits;

        // Copy `other` value.
        digits = temp;
        memcpy( digits, other.digits, other.digit_count * sizeof(storage_type) );
        digit_count = other.digit_count;
    }
    return *this;
}
```

return *this??

- In C and C++, assignment is an operator, and we have *assignment expressions*.
  - This is unusual. In many languages assignment is a statement form.
- C has what are called *expression statements* that are made by adding a semicolon to the end of an expression.
  - Most languages don't do this unless they are based on C semantics.

```
int x, y, z;

x + y;      // Legal. An expression statement from an add expr.
z = x + y;  // Legal. An expression statement from an assignment expr.
```

Say What?

- Consider:
  - `x + y;` is a valid statement, but it has no effect since addition changes neither operand and nothing is done with the result.
  - `x = y;` is a valid statement, bit it *does* have an effect since assignment changes its left operand.
  - In any event, `=` is an operator in C/C++ and, in C++, it can be overloaded.
- Assignment normally returns the left operand after the assignment (and any implicit type conversions) has happened.
  - Thus: `x = y + (a = b);` is legal. It puts the value of `b` into `a`, returning the new `a` (i.e., the value of `b` after implicit type conversions), adds that result to `y` and puts the final answer into `x`.

Is It Useful?

- Sometimes
  - One semi-common usage is to *chain assignments*

```
int x, y, z;
x = y = z = 0;
```

  - Because assignment associates from right to left, the above is the same as

```
int x, y, z;
x = (y = (z = 0));
```

This has the effect of assigning zero to `z`. Then since `z = 0` returns 0, that zero gets assigned to `y`, etc. *You might want to do this with your own classes too!*

User-Defined Copy-Assignment

- You could declare your `operator=( )` to return **`void`**.
  - Most of the time nobody would notice and it's less quirky.
  - But it will prevent chaining assignments.
- Thus, it is normal to declare `operator=( )` to return a reference to the class (`BigInteger &`).
  - Then, as the last statement of the implementation:
  - `return *this;`

Copy Construction vs Copy-Assignment

- Copy constructors are much simpler than copy-assignment operators.
  - There is no existing value to clean up.
  - Exception safety is easier
    - No existing value to worry about preserving.
    - No need to maintain invariants or destructibility because the destructor will not run on objects that fail to construct*. Also, such objects are ~~impossible~~ difficult to access so the programmer can’t touch/use them.
  - No need to worry about self-assignment
  - Constructors don’t return anything, so the return type is not relevant
- The copy constructor is likely faster and/or consumes fewer resources
  - <u>*Initialize objects when they are declared. Avoid assigning to objects!*</u>

* If you throw in a constructor, be sure to release resources already acquired before the throw!

The Triad (Life Cycle Methods)

- The following three methods go together:

```
~BigInteger( )
 BigInteger( const BigInteger &other );
 BigInteger &operator=( const BigInteger &other );
```

- If you have one, you probably need all three.
  - Some compilers will warn if you are missing one or two.
- Classes that manage external resources need…
  - … a destructor to release those resources AND
  - ... a copy constructor and copy assignment operator to manage copying those resources.

I Don’t Want To Copy

- Certain classes don’t make sense to copy.
  - The `std::thread` class manages a thread of execution. What would it even mean to copy an executing thread?
- However, if you don’t define your own copy constructor and copy-assignment operator, the compiler will generate one that copy (constructs/assigns) the members.
- You can suppress this:

```
BigInteger( const BigInteger &other ) = delete;
BigInteger &operator=( const BigInteger &other ) = delete;
```

Deleting the methods tells the compiler to not generate them.
Also, you don’t implement them.
Attempts to copy objects become compile-time errors.

But Wait! There's move… er… more!

- *FINISH ME!*