Dynamic memory management: Extended Example

CS 115

Dr. Joseph Eremondi, adapted from Dr. Shakil Khan, Dr. Philip Fong, and Dr. Howard Hamilton

Last updated: March 21, 2025

Strings with Dynamic Memory

Dynamic memory management and ADTs

- Why handle manually when we can automate allocation using constructors and deallocation using destructors?

Dynamic memory management and ADTs

- Why handle manually when we can automate allocation using constructors and deallocation using destructors?
  - e.g. Library does manual memory management so client doesn't have to

Dynamic memory management and ADTs

- Why handle manually when we can automate allocation using constructors and deallocation using destructors?
  - e.g. Library does manual memory management so client doesn't have to
- design a wrapper class for c-strings for

Dynamic memory management and ADTs

- Why handle manually when we can automate allocation using constructors and deallocation using destructors?
  - e.g. Library does manual memory management so client doesn't have to
- design a wrapper class for c-strings for
  - allocating and deallocating memory for c-strings

Dynamic memory management and ADTs

- Why handle manually when we can automate allocation using constructors and deallocation using destructors?
  - e.g. Library does manual memory management so client doesn't have to
- design a wrapper class for c-strings for
  - allocating and deallocating memory for c-strings
  - streamline the copying and concatenation of c-strings

Dynamic memory management and ADTs

- Why handle manually when we can automate allocation using constructors and deallocation using destructors?
  - e.g. Library does manual memory management so client doesn't have to
- design a wrapper class for c-strings for
  - allocating and deallocating memory for c-strings
  - streamline the copying and concatenation of c-strings
- In-depth example: a library wrapping C-strings

Dynamic memory management and ADTs

- Why handle manually when we can automate allocation using constructors and deallocation using destructors?
  - e.g. Library does manual memory management so client doesn't have to
- design a wrapper class for c-strings for
  - allocating and deallocating memory for c-strings
  - streamline the copying and concatenation of c-strings
- In-depth example: a library wrapping C-strings
  - We'll get something similar to the stdlib `string`

Dynamic memory management and ADTs

- Why handle manually when we can automate allocation using constructors and deallocation using destructors?
  - e.g. Library does manual memory management so client doesn't have to
- design a wrapper class for c-strings for
  - allocating and deallocating memory for c-strings
  - streamline the copying and concatenation of c-strings
- In-depth example: a library wrapping C-strings
  - We'll get something similar to the stdlib `string`
  - Show how to do the main operations

Dynamic memory management and ADTs

- Why handle manually when we can automate allocation using constructors and deallocation using destructors?
  - e.g. Library does manual memory management so client doesn't have to
- design a wrapper class for c-strings for
  - allocating and deallocating memory for c-strings
  - streamline the copying and concatenation of c-strings
- In-depth example: a library wrapping C-strings
  - We'll get something similar to the stdlib `string`
  - Show how to do the main operations
    - Copy/default/assignment constructors, destructors, operators

Interface and Implementation

Interface and Implementation

```
class String {
public:
  ...
unsigned int length() const;
  char member(unsigned int i) const;
  ...
  private:
  const char *buf;
};

unsigned int String::length() const {
  return strlen(buf);
}

char String::member(unsigned int i) const {
  assert(i < length());

  return buf[i];
}
```

Allocating in the Constructor

Allocating in the Constructor

```
class String {
public:
  String(const char *s);
  unsigned int length() const;
  char member(unsigned int i) const;
private:
  const char *buf;
};

// Usage: String s("Hello World");
// Alternate syntax: String s = "Hello World";
// implementation

String::String(const char *s) {
  char *newbuf = new char[strlen(s) + 1];
  strcpy(newbuf, s);
  buf = newbuf;
}
```

Constructors are not enough

- Constructor allocated memory

Constructors are not enough

- Constructor allocated memory
  - When object goes out of scope, this memory will *leak*

Constructors are not enough

- Constructor allocated memory
  - When object goes out of scope, this memory will *leak*
- We need a *destructor*

Constructors are not enough

- Constructor allocated memory
  - When object goes out of scope, this memory will *leak*
- We need a *destructor*
- Want a way to “cleanup” when we’re done with an object

Constructors are not enough

- Constructor allocated memory
  - When object goes out of scope, this memory will *leak*
- We need a *destructor*
- Want a way to “cleanup” when we’re done with an object
  - Free memory or other resources

Constructors are not enough

- Constructor allocated memory
  - When object goes out of scope, this memory will *leak*
- We need a *destructor*
- Want a way to “cleanup” when we’re done with an object
  - Free memory or other resources
  - Called whenever we do `free` on an object

Constructors are not enough

- Constructor allocated memory
  - When object goes out of scope, this memory will *leak*
- We need a *destructor*
- Want a way to “cleanup” when we’re done with an object
  - Free memory or other resources
  - Called whenever we do `free` on an object
  - Automatically called on locally-declared variables that go out of scope

Constructors are not enough

- Constructor allocated memory
  - When object goes out of scope, this memory will *leak*
- We need a *destructor*
- Want a way to “cleanup” when we’re done with an object
  - Free memory or other resources
  - Called whenever we do `free` on an object
  - Automatically called on locally-declared variables that go out of scope
    - Since their memory only exists for the duration of the function call

A String Destructor

A String Destructor

```
class String {
public:
  String(const char *s);
  ~String();     // Destructor prototype
  unsigned int length() const;
  char member(unsigned int i) const;

private:
  const char *buf;
};
// Usage
    { // begin of scope
      String s = "Hello World";
      ...
        } // end of scope: memory should be
// reclaimed here

// Implementation
String::~String() {
  delete [] buf;
}
```

Default Constructor for String

Default Constructor for String

```
class String {
public:
  String();         // Default constructor
  String(const char *s);
  ~String();      // Destructor prototype

  unsigned int length() const;
  char member(unsigned int i) const;

private:
  const char *buf;
};
// Implementation
String::String(){
  // Create a c-string of length 0
  char *newbuf = new char[1];
  newbuf[0] = '\0';
  buf = newbuf;
}
```

Copy Constructor: Shallow vs. Deep Copies

Copy Constructor: Shallow vs. Deep Copies

```
class String {
public:
  String();
  String(const char *s);
  String(const String &original);
  ~String();
  unsigned int length() const;
  char member(unsigned int i) const;
private:
  const char *buf;
};
// deep copying intended

String::String(const String &original) {
  unsigned int len = original.length();
char* nonConstBuf
= new char[len + 1];
strcpy(nonConstBuf, original.buf);
buf = nonConstBuf;
}
// is the & before original really required?
```

Shallow vs. Deep Copies

- Shallow copy only copies pointers

Shallow vs. Deep Copies

- Shallow copy only copies pointers
  - Changes to the copy will affect the original

Shallow vs. Deep Copies

- Shallow copy only copies pointers
  - Changes to the copy will affect the original
  - Mulitiple pointers to the same memory

Shallow vs. Deep Copies

- Shallow copy only copies pointers
  - Changes to the copy will affect the original
  - Mulitiple pointers to the same memory
- Deep Copies allocate new memory and copy the data over

Shallow vs. Deep Copies

- Shallow copy only copies pointers
  - Changes to the copy will affect the original
  - Mulitiple pointers to the same memory
- Deep Copies allocate new memory and copy the data over
  - Change to one won’t affect the other

Shallow vs. Deep Copies

- Shallow copy only copies pointers
  - Changes to the copy will affect the original
  - Mulitiple pointers to the same memory
- Deep Copies allocate new memory and copy the data over
  - Change to one won’t affect the other
- Each useful in different circumstances

Shallow vs. Deep Copies

- Shallow copy only copies pointers
  - Changes to the copy will affect the original
  - Mulitiple pointers to the same memory
- Deep Copies allocate new memory and copy the data over
  - Change to one won’t affect the other
- Each useful in different circumstances
  - Important to document which you’re defining

Copy Constructor Uses

- Three different uses:

Copy Constructor Uses

- Three different uses:

Copy Constructor Uses

- Three different uses:

```
// for initializing a string object by another
String s("Hello"); // Const. invoked
String t(s);           // Copy const. invoked
// Alternative syntax
String s = "Hello"; // Const. invoked
String t = s;           // Copy const. invoked
// for passing String args. by value
void f(String s){...}
...
String t = "Hello";
f(t);    // Copy const. invoked

// for returning string instances as value
String f(...){
  String s;
  ...
    // Copy const. invoked to create
    // return value
    return s;
}
```

One Final Point

- Can the & before &original be left out?

One Final Point

- Can the & before &original be left out?

One Final Point

- Can the & before &original be left out?

```
String::String(const String &original) {
  unsigned int len = original.length();
  char *nonConstBuf = new char[len + 1];
  strcpy(nonConstBuf, original.buf);
  buf = nonConstBuf;
}
```

- No: The copy constructor is always invoked whenever an argument of the type is passed by value to any function

One Final Point

- Can the & before &original be left out?

```
String::String(const String &original) {
  unsigned int len = original.length();
  char *nonConstBuf = new char[len + 1];
  strcpy(nonConstBuf, original.buf);
  buf = nonConstBuf;
}
```

- No: The copy constructor is always invoked whenever an argument of the type is passed by value to any function
- so if & is left out, it will repeatedly call the copy constructor till the stack overflows!

Assignment Operator: Prototype

Assignment Operator: Prototype

```
class String {
public:
  // good idea to follow standard prototype for = operator
  String &operator=(const String &original);
  ...
  private:
  const char *buf;
};

String s;                    // default constructor
String t = "Hello";  // auxiliary constructor
String u(t);                 // copy constructor
s = u;                       // assignment operator
```

Assignment Operator: Implementation Attempt

Assignment Operator: Implementation Attempt

```
String &String::operator=(const String &original) {
  // len is length of string to be copied
  unsigned int len = original.length();
  // allocate new space of size len
  char *nonConstBuf = new char[len + 1];
  // copy original string to new space (Line 4)
  strcpy(nonConstBuf, original.buf);
  // deallocate old string (Line 5)
  delete [] buf;
  // make old string pointer point to newly allocated space
  buf = nonConstBuf;
  return *this;
}
```

Problem with that implementation

Problem with that implementation

```
// a potential issue
String s = "Hello, World";
s = s;   // Self assignment! Q: Is there anything wrong with this?
// Ans. might do extra work; even more problematic if Line 5 is moved
```

Assignment (fixed version)

Assignment (fixed version)

```
String &String::operator=(const String &original) {
  if (&original != this){ // Don't duplicate if they're the same
    // len is length of string to be copied
    unsigned int len = original.length();
    // allocate new space of size len
    char *nonConstBuf = new char[len + 1];
    // copy original string to new space
    strcpy(nonConstBuf, original.buf);
    delete [] buf; // deallocate old string
    // make old string pointer point to newly allocated space
    buf = nonConstBuf;
  }

  return *this;
}
```

Concatenation

- Merging Two Strings into one

Concatenation

- Merging Two Strings into one

Concatenation

- Merging Two Strings into one

```
class String {
public:
  ...
String &append(const String &s);
  ...
  private:
  const char *buf;
};

// note: none of the following work
// void append(const String &s)
//  String append(const String &s)
String s = "Hello";
String t = " World";
s.append(t);              // "Hello World"
String s = "Hello";
String t = " ";
String u = "World";
s.append(t).append(u);
```

Pseudocode for Concatenation

- Allocate a buffer that is big enough to hold both the content of the current object and that of the argument

Pseudocode for Concatenation

- Allocate a buffer that is big enough to hold both the content of the current object and that of the argument
- Copy the content of the current object to the beginning of the buffer

Pseudocode for Concatenation

- Allocate a buffer that is big enough to hold both the content of the current object and that of the argument
- Copy the content of the current object to the beginning of the buffer
- Append the content of the argument to the end of the buffer

Pseudocode for Concatenation

- Allocate a buffer that is big enough to hold both the content of the current object and that of the argument
- Copy the content of the current object to the beginning of the buffer
- Append the content of the argument to the end of the buffer
- Delete the original content of the current object

Pseudocode for Concatenation

- Allocate a buffer that is big enough to hold both the content of the current object and that of the argument
- Copy the content of the current object to the beginning of the buffer
- Append the content of the argument to the end of the buffer
- Delete the original content of the current object
- Install the buffer into the current object

Concatenation Implementation

Concatenation Implementation

```
String &String::append(const String &s){
  unsigned int len = strlen(buf) + strlen(s.buf);
  char *newbuf = new char[len + 1];
  strcpy(newbuf, buf);
  strcat(newbuf, s.buf);
  delete [] buf;
  buf = newbuf;

  return *this;
}
```

Creating a new string: Inefficient Version

- append is a mutation operation, let’s formulate a creation version

Creating a new string: Inefficient Version

- append is a mutation operation, let’s formulate a creation version
  - Leave original strings unchanged, make a new one that merges both

Creating a new string: Inefficient Version

- append is a mutation operation, let’s formulate a creation version
  - Leave original strings unchanged, make a new one that merges both

Creating a new string: Inefficient Version

- append is a mutation operation, let's formulate a creation version
  - Leave original strings unchanged, make a new one that merges both

```
String concatenate(const String &s) const; // prototype

// usage
String s = "Hello";
String t = " World";
String u = s.concatenate(t);

// implementation
String String::concatenate(const String &s) const {
  return String(*this).append(s);
}
```

A (maybe) More Efficient Version

- Efficient only if supported by the compiler

A (maybe) More Efficient Version

- Efficient only if supported by the compiler
- First, let’s define a private helper constructor

A (maybe) More Efficient Version

- Efficient only if supported by the compiler
- First, let’s define a private helper constructor

A (maybe) More Efficient Version

- Efficient only if supported by the compiler
- First, let's define a private helper constructor

```
String::String(const char *s, const char *t) {
  unsigned int len = strlen(s) + strlen(t);
  char *newbuf = new char[len + 1];
  strcpy(newbuf, s);
  strcat(newbuf, t);
  buf = newbuf;
}

String String::concatenate(const String &s) const {
  // still calls the copy constructor, but some smart compilers
  // will be able to recognize and avoid this unnecessary task
  return String(buf, s.buf);
}
```