Computer Science 3307a/b

Managing Object Resources

Jeff Shantz
Department of Computer Science

Based on notes created by our very own PhD student, Beth Locke

Manual Memory Management

- Like C, C++ has no automatic memory management (e.g. garbage collection)

- As programmers, we are responsible for ensuring:
  - No uninitialized values are used
  - Addresses used are correct (e.g. pointers point to the right places)
  - Dynamically allocated memory is reclaimed
  - Integrity of objects is maintained

C++ Memory Management

- Entering a scope (e.g. calling a function):
  - Space allocated on stack for local variables, arguments
- When `new` is called:
  - Space allocated on heap for the object
- Leaving a scope (e.g. returning from a function):
  - Stack space used by local scope reclaimed
- When `delete` is called:
  - Heap space used by object reclaimed

Uninitialized Data

Listing 1: `uninit.cpp`

```
main()
{
  int i;
  float f;
  double d;
  bool b;
  Person* p;

  cout << "int: " << i << endl;
  cout << "float: " << f << endl;
  cout << "double: " << d << endl;
  cout << "bool: " << b << endl;
  cout << "ptr: " << p->_name << endl;
}
```

Uninitialized Data

Output on the Research Network

```
int: -4195156
float: 5.60519e-45
double: 8.48591e-314
bool: 48
Bus Error (core dumped)
```

Output on my Mac

```
int: 32767
float: 1.03424e+21
double: 0
bool: 0
Segmentation fault: 11
```

- Can't rely on compiler to initialize values
- Common error in C programming

Constructor

- Called when an object of a class is created
- Allows one to guarantee that member variables of a new data type will always be properly initialized
- Can be overloaded:

Listing 2: `Person.h`

```
class Person {
  public:
    Person(std::string name);
    Person(std::string name, int age);
    Person(std::string name, int age, bool female);
```

Constructor

- Should initialize all members:
    - Primitive types (`int _x;`):
        - Give meaningful/default value
    - Objects (`std::string _name;`):
        - Call the appropriate constructor
    - Pointers (`Pet* _pet`):
        - Assign meaningful address, or set to 0 (NULL)
        - Dynamically allocate space, if required
        - Ensure the entity pointed at is also initialized

Default Constructor

- Constructor with no arguments:

```
Person::Person() { ... }
...
Person baby;
```

- Generated by compiler if no constructors are defined:
  - Default constructs the base class (if any)
  - Default constructs any object members
  - No other initialization performed
  - Can't rely on compiler to zero out values as seen earlier

Default Constructor

Listing 3: `Person.h`

```
class Person {

  public:
    std::string name() const;
    int age() const;
    bool female() const;
    Person* best_friend() const;

  private:
    std::string _name;
    int _age;
    bool _female;
    Person* _bff;
};
```

Default Constructor

Listing 4: default_ctor.cpp

```
  Person p;  // default constructor called

  cout << "Name:        " << p.name() << endl;
  cout << "Age:         " << p.age() << endl;
  cout << "Female:      " << p.female() << endl;
  cout << "Best Friend: " << p.best_friend() <<
      endl;
```

Default Constructor

Output on the Research Network

```
Name:
Age:         -4195212
Female:      0
Best Friend: 0xffbffc7c
```

Output on my Mac

```
Name:
Age:         368376262
Female:      87
Best Friend: 0
```

Default Constructor

- If any constructors are defined, the default constructor will not be generated by the compiler

Listing 5: `no_default_ctor.cpp`

```cpp
class Employee
{
  Employee(int id);
};

Employee::Employee(int id) { }

main()
{
  Employee e; // won't compile
}
```

```
no_default_ctor.cpp: In function  int main()  :
no_default_ctor.cpp:10: error: no matching function for call to
      Employee::Employee()
no_default_ctor.cpp:6: note: candidates are: Employee::Employee(int)
no_default_ctor.cpp:2: note:                 Employee::Employee(const
      Employee&)
make: *** [no_default_ctor] Error 1
```

Initialization Lists

- Before constructor code is entered, the object is already allocated. This will:
  - create space for members
  - call constructors for member objects

Initialization Lists

- What is the problem with the Person constructor?

Listing 6: `Person.h`

```
class Person {
  public:
    Person(std::string name, std::string pet_name);

  private:
    std::string _name;
    Pet _pet;
};
```

Listing 7: `Person.cpp (without an initialization list)`

```
Person::Person(string name, string pet_name)
{
  cout << "In constructor Person(name, pet_name)"
       << endl;
  this->_name = name;
  this->_pet = Pet(pet_name);
}
```

Initialization Lists

Listing 8: `no_init_list.cpp`

```
#include "Person.h"

main()
{
  Person p("Jeff", "Maggie");
}
```

```
In default Pet constructor
In constructor Person(name, pet_name)
In constructor Pet(name)
In Pet operator=()
```

- Default constructors called for `_name` and `_pet`
- `_name` and `_pet` are then re-initialized in the `Person` constructor

Initialization Lists

Listing 9: `Person.cpp (with an initialization list)`

```
Person::Person(string name, string pet_name) :
    _name(name), _pet(pet_name)
{
  cout << "In constructor Person(name, pet_name)"
       << endl;
}
```

Listing 10: `init_list.cpp`

```
#include "Person.h"

main()
{
  Person p("Jeff", "Maggie");
}
```

```
In constructor Pet(name)
In constructor Person(name, pet_name)
```

Initialization Lists

- Allow us to specify which constructors to use when creating the object
- Provided in implementation file (`.cpp` file)
- Makes initialization more efficient

Destructor

- Constructor: called when an object is created
- Destructor: called when an object is destroyed
  - Variable holding object goes out of scope
  - `delete` called on a pointer to an object
- Cannot be overloaded
- Should reclaim dynamically allocated memory
- Should release system resources
  - Close files, database connections, network connections, etc.

Destructor

Listing 11: `Person.h`

```
class Person {
  public:
    Person(std::string name);
    ~Person();

  private:
    std::string _name;
};
```

Listing 12: `Person.cpp`

```
Person::Person(string name) : _name(name)
{
  cout << "In Person constructor" << endl;
}

Person::~Person()
{
  cout << "In Person " << this->_name << " destructor" << endl;
}
```

Destructor

Listing 13: `dtor.cpp`

```
void f(Person p)
{
  cout << "In f();" << endl;
}

main()
{
  Person p1("Jeff");
  Person* p2 = new Person("Bob");
  Person* p3 = new Person("Joe");

  cout << "Calling f()" << endl;
  f(p1);

  delete p2;
}
```

```
In Person constructor
In Person constructor
In Person constructor
Calling f()
In f();
In Person Jeff destructor
In Person Bob destructor
In Person Jeff destructor
```

Destructor

- When members of a class are destroyed:
  - Primitive types (`int _x;`):
    - Space reclaimed implicitly, following destructor
  - Objects (`std::string _name;`):
    - Following destructor, each object's destructor is called implicitly and its space is reclaimed
  - Pointers (`Pet* _pet`):
    - Space reclaimed for pointer only (remember: a pointer is merely a 32-bit integer)
    - Programmer must `delete` dynamically allocated memory explicitly in constructor

Default Destructor

- If a destructor is not defined by a class:
    - The compiler implicitly defines it as empty `{}`
    - This will still correctly reclaim memory allocated for primitive object members
        - Destructors will be implicitly called on object members
    - Will not delete pointer-referenced data
        - Memory leak!

`delete`

- `new`:
  - Space allocated on the heap for data
  - Constructor called
- `delete`
  - Destructor called
  - Space reclaimed

delete

Listing 14: Person.h

```
class Person
{
  public:
    Person(std::string name);
    Person(std::string name, Person* bff);
    ~Person();

    std::string name() const;

  private:
    std::string _name;
    Person* _bff;
};
```

delete

Listing 15: new_delete.cpp

```
main()
{
  Person* p1 = new Person("Jeff");
  Person* p2 = new Person("Joe", p1);
  Person* p3 = new Person("John", p2);

  cout << "Deleting p1" << endl;

  delete p1;  // p2 now has a dangling pointer
  p1 = NULL;

  Person** people = new Person*[2];

  people[0] = p2;
  people[1] = p3;

  cout << "Deleting array" << endl;

  // Reclaim memory allocated for array;
  // does not delete array elements
  delete [] people;

  cout << "I am P2 and I am still valid: " << p2->name() << endl;

  //memory leak
}
```

`delete`

Output

```
In constructor(name) for Person Jeff
In constructor(name, bff) for Person Joe
In constructor(name, bff) for Person John
Deleting p1
In destructor for Person Jeff
Deleting array
I am P2 and I am still valid: Joe
```

`delete`

- Match `new` and `delete`
- Any data allocated with `new` should be reclaimed with `delete`
- Any data allocated with `new[]` should be reclaimed with `delete[]`
- Be cognizant that `delete[]` only deletes the space allocated for the array – it does not delete the array elements themselves
- If `new` called in constructor, should typically call `delete` in destructor

Copy Constructor

- Constructor that is called when:
  - creating an object by copying another:

Listing 16: copy_ctor.cpp

```cpp
    Person p1("Jeff");

    Person p2 = p1;    // copy constructor
    Person p3(p1);     // copy constructor
```

Output

```
In constructor(name) for Person Jeff
In copy constructor for Person
In copy constructor for Person
.
.
In destructor for Person Jeff
In destructor for Person Jeff
In destructor for Person Jeff
```

Copy Constructor

- Passing objects by value:

Listing 17: `copy_ctor.cpp`

```
void f(Person p) // copy constructor
{
  cout << "In f()" << endl;
}
```

```
  cout << "Calling f()" << endl;
  f(p1);
  cout << "After f()" << endl;
```

Copy Constructor

Output

```
.
.
Calling f()
In copy constructor for Person
In f()
In destructor for Person Jeff
After f()
.
.
```

Copy Constructor

- When returning objects by value:

Listing 18: copy_ctor.cpp

```
Person g(Person p) // copy constructor
{
  cout << "In g()" << endl;
  return p; // copy constructor
}
```

```
  cout << "Calling g()" << endl;
  Person p4 = g(p1);
  cout << "After g()" << endl;
```

Copy Constructor

Output

```
.
.
Calling g()
In copy constructor for Person
In g()
In copy constructor for Person
In destructor for Person Jeff
After g()
.
.
```

Copy Constructor

Listing 19: `Person.cpp`

```
Person::Person(const Person& other) : _name(other._name), _bff(other.
      _bff)
{
  cout << "In copy constructor for Person" << endl;
}
```

- Parameter **must** be a reference
  - Why? What if we passed by value?
- Parameter **should** be `const`
  - Guarantees that we cannot change the object being copied
  - Allows `const` objects to be copied
  - What would happen if we didn't use `const`?

Default Copy Constructor

- Generated by compiler if no copy constructor is defined
- Creates a shallow copy:
  - Makes copies of data members in class
  - primitives, objects and pointer values
    - May lead to unintentionally sharing resources

Default Copy Constructor

Listing 20: `Array.h`

```
class IntArray
{
  public:
    IntArray(int size);
    ~IntArray();

    int& operator[](const int idx);
    int operator[](const int idx) const;
    int size() const;

  private:
    int* _values;
    int _size;
};
```

Listing 21: `Array.cpp`

```
IntArray::IntArray(int size) : _size(size)
{
  this->_values = new int[this->_size];
}

IntArray::~IntArray()
{
  delete [] this->_values;
}
```

Default Copy Constructor

Listing 22: `shallow_copy.cpp`

```
main()
{
  IntArray a(4);

  a[0] = 4;
  a[1] = 3;
  a[2] = 2;
  a[3] = 1;

  IntArray b = a;

  a[0] = 99;

  cout << "a[0] = " << a[0] << endl;
  cout << "b[0] = " << b[0] << endl;
}
```

Output

```
a[0] = 99
b[0] = 99
Abort trap: 6
```

- Two problems here: what and why?

Copy Constructor - Deep Copy Semantics

- If an object has pointers, we will likely wish to implement *deep copy semantics*
  - i.e. make copies of what pointers (to dynamically allocated entities) reference

Copy Constructor - Deep Copy Semantics

Listing 23: `Array.h`

```
class IntArray
{
  public:
    IntArray(int size);
    IntArray(const IntArray& other);
    ~IntArray();

    int& operator[](const int idx);
    int operator[](const int idx) const;
    int size() const;

  private:
    int* _values;
    int _size;
};
```

Copy Constructor - Deep Copy Semantics

Listing 24: `Array.cpp`

```
IntArray::IntArray(int size) : _size(size)
{
  this->_values = new int[this->_size];
}

IntArray::IntArray(const IntArray& other) : _size(other.size())
{
  this->_values = new int[other.size()];

  for (int i = 0; i < this->_size; ++i)
  {
    this->_values[i] = other[i];
  }

}

IntArray::~IntArray()
{
  delete [] this->_values;
}
```

Copy Constructor - Deep Copy Semantics

Listing 25: deep_copy.cpp

```
main()
{
  IntArray a(4);

  a[0] = 4;
  a[1] = 3;
  a[2] = 2;
  a[3] = 1;

  IntArray b = a;

  a[0] = 99;

  cout << "a[0] = " << a[0] << endl;
  cout << "b[0] = " << b[0] << endl;
}
```

Output

```
a[0] = 99
b[0] = 4
```

Assignment Operator

- Similar to the copy constructor, but called when assigning to an object that has already been initialized
- More responsibilities than the copy constructor:
  - test against self assignment
  - clean up existing object
  - return a reference to `*this`

Listing 26: `asn_operator.cpp`

```
main()
{
  IntArray a(4);    // constructor
  IntArray b(4);    // constructor

  a[0] = 4;
  a[1] = 3;
  a[2] = 2;
  a[3] = 1;

  IntArray c = b;   // new object -> copy constructor

  c = a;            // existing object -> asn operator
```

Default Assignment Operator

- As with the copy constructor,
  - Compiler generates default assignment operator if none is provided
  - Default assignment operator implements shallow copy semantics

Assignment Operator - Deep Copy

Listing 27: `Array.h`

```
class IntArray
{
  public:
    IntArray(int size);
    IntArray(const IntArray& other);
    ~IntArray();

    IntArray& operator=(const IntArray& other);

    int& operator[](const int idx);
    int operator[](const int idx) const;
    int size() const;

  private:
    int* _values;
    int _size;
};
```

Assignment Operator - Deep Copy

Listing 28: `Array.cpp`

```
IntArray::IntArray(const IntArray& other) : _size(other.size())
{
  this->_values = new int[other.size()];

  for (int i = 0; i < this->_size; ++i)
  {
    this->_values[i] = other[i];
  }

}

IntArray& IntArray::operator=(const IntArray& other)
{
  if (this != &other)
  {
    int* temp = new int[other.size()];

    for (int i = 0; i < this->_size; ++i)
    {
      temp[i] = other[i];
    }

    delete [] this->_values;
    this->_values = temp;
  }
```

Assignment Operator - Deep Copy

Listing 29: asn_operator.cpp

```cpp
main()
{
  IntArray a(4);    // constructor
  IntArray b(4);    // constructor

  a[0] = 4;
  a[1] = 3;
  a[2] = 2;
  a[3] = 1;

  IntArray c = b;   // new object -> copy constructor

  c = a;            // existing object -> asn operator

  a[0] = 99;

  cout << "a[0] = " << a[0] << endl;
  cout << "c[0] = " << c[0] << endl;
}
```

Output

```
a[0] = 99
c[0] = 4
```

Assignment Operator

- Points of discussion:
  - What was the purpose of the following line in the assignment operator?

    ```
    if (this != &other)
    ```

  - Hint: what happens if we execute `a = a;`?
  - Why did we return a reference? Why not `void`?

    ```
    IntArray& IntArray::operator=(const IntArray& other)
    ```

  - Hint: what happens if we execute `a = b = c;`?

Best Practices

- Define a default constructor
- Explicitly define a destructor, even if empty
- **Rule of Three**:
  - If you need one of the following, you probably need all three:
    - Copy constructor
    - Assignment operator
    - Non-empty destructor
- If a class manages pointers to dynamically allocated entities, define the big three