Classes in C++98

January 21, 2019

Brian A. Malloy

1. Overview

- When we refer to C++98, we are referring to C++98 and C++03, since they differ only slightly.
- C++98 contained 3 types of constructors, but C++11 added a move constructor.
- The C++ class is one of the most difficult constructs to write correctly
- Some methods are written silently by the compiler
- Some methods are required w/ pointers
- These slides describe classes, including 3 of the 4 constructors.
- We describe move semantics in separate slides

2. What is a *class*?

- Unit of encapsulation:
  - Public operations
  - Private implementation
- Abstraction:
  - string: abstracts char* of C
  - student
  - sprite
- C++ Classes: easy to write, hard to get right!
- Need lots of examples

2.1. The actions of a *class*

- Constructors: initialize data attributes
- Constructors: allocate memory when needed
- Destructor: De-allocate memory when necessary

2.2. C++ *class* vs C++ *struct*

- Default access is only difference
- Generally, structs used for data
- Classes used for data and methods

Bad class	Good Class

`class Student {`
`public:`
`   string name;`
`   float gpa;`
`};`
	`class Student {`
	`   string name;`
	`   float gpa;`
	`};`

2.3. Object: an instantiated class

- C++ objects can be stored on the stack:

```
class A{};
int main() {
  A a, b;
};
```

- Or on the heap:

```
int main() {
  A *a = new A;
  A *b = new B;
};
```

- Compiler does stack; programmer does heap!

3. Constructors & Destructors

- No name and cannot be called directly
- Init data through initialization lists
- Constructor types are distinguished by their parameters.
- The four types of constructors are:

  1. Default
  2. Conversion
  3. Copy
  4. Move (which we describe in later slides)

Constructor examples:

```
class Student {
public:
  Student();                   // default: no params
  Student(char * n);           // convert
  Student(const Student&);     // copy: param is Student
  Student(Student&&);          // move
  ~Student();                  // destructor (no params)
private:
  char* name;
};
```

3.1. Default Constructor

```
class string {
public:
  string() : buf(new char[1]) { buf[0] = '\0'; }
private:
  char* buf;
};
```

- No parameters to default constructor
- Uses an initialization list to create a "buffer" of length 1 characters: buf(new char[1])
- Places the null termination character into the newly created buffer.
- cppreference: Constructs an empty string, with a length of zero characters.

Overview
What is a class?
Constructors & ...
What if I don't write ...
Why Prefer Init?
Static Class Variabless
Principle of Least ...
Interface vs ...
Makefiles
Overload Operators
Slide 9 of 38
Go Back
Full Screen
Quit

3.2. Prefer initialization to assignment

- Initialization is more efficient for data members that are objects (demo later)
- Only way to pass parameters to base class:

```
class Person {
public:
   Person(int a) : age(a) {}
private:
   int age;
};
class Student : public Person {
public:
  Student(int age, float g) : Person(age), gpa(g) {}
private:
  float gpa;
};
```

3.3. Init performed in order of declare

- In Student, the constructor will initialize iq first, then age, because iq appears first in declaration (line 5).
- Initialization list not needed for built-in types.

```
1 class Student {
2 public:
3   Student(int a) : age(a), iq(age+100) {}
4 private:
5   int iq;
6   int age;
7 };
```

3.4. Conversion Constructor

```
1 class string {
2 public:
3   string(const char* b) :
4     buf(new char[strlen(b)+1]) {
5     strcpy(buf, b);
6   }
7 private:
8   char* buf;
9 };
```

- Converts b, on line 3, into a string
- strlen returns the size of the c-string, not including the null termination
- On line 4 we allocate strlen(b)+1 bytes, where +1 allows for the null termination

3.5. Copy Constructor

```
1 class string {
2 public:
3   string(const string& s) :
4     buf(new char[strlen(s.buf)+1]) {
5     strcpy(buf, s.buf);
6   }
7 private:
8   char* buf;
9 };
```

- Copy constructor uses the parameter s, line 3, to make a **deep** copy.
- Notice the parameter transmission mode: const&

3.6. Destructor

```
1 class string {
2 public:
3   ~string() { delete [] buf; }
4 private:
5   char* buf;
6 };
```

- We used new char[] in the constructors to allocate an array
- We use delete [] on line 3 to indicate that we are deallocating an array.

Slide 14 of 38
Go Back
Full Screen
Quit

4. What if I don't write one

I write this:

```
class Empty{};
```

Compiler writes this:

```
class Empty {
public:
  Empty();
  Empty(const Empty &);
  ~Empty();
  Empty& operator=(const Empty &);
};
```

4.1. Here's what they look like:

```
inline Empty::Empty()  {}
inline Empty::~Empty() {}

inline Empty * Empty::operator&() {return this;}

inline const Empty * Empty::operator&() const {
   return this;
}
```

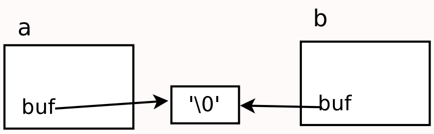
The copy constructor & assignment operator simply do a member wise copy, i.e., shallow. Note that the default copy/assign may induce leak/dble free

4.2. What can go wrong? Consider:

```cpp
#include <iostream>
#include <cstring>
class string {
public:
  string() : buf(new char[1]) { buf[0] = '\0'; }
  string(const char * s) :
    buf(new char[strlen(s)+1]) {
    strcpy(buf, s);
  }
  ~string() { delete [] buf; }
private:
  char * buf;
};
int main() {
  string a, b(a);
}
```

4.3. Shallow Copy

- The previous example gives undefined behavior, usually double free.
- Default constructor creates string a, line 15
- However, the compiler generated copy constructor simply copies the address in a.buf into b.buf, which makes a shallow copy
- In memory it looks like:



Deletion of a is okay; deletion of b is a problem!

4.4. Prevent Compiler Generated Ctors

- To address the problem of shallow copies, C++98 developers suggested placing signatures in private (line 10).
- Use of copy constructor won't compile
- This is Item #6 in Meyers Effective C++.

```
1   #include <iostream>
2   #include <cstring>
3   class string {
4   public:
5     string();
6     string(const char * s);
7     ~string() { delete [] buf; }
8   private:
9     char * buf;
10    string(const string&);
11  };
```

4.5. C++11 Solution

- If the special syntax = delete is used, the function is defined as deleted (line 8)
- Any use of a deleted function is ill-formed and the program will not compile.

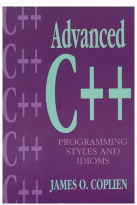
```
1  #include <iostream>
2  #include <cstring>
3  class string {
4  public:
5    string();
6    string(const char * s);
7    ~string() { delete [] buf; }
8    string(const string&) = delete;
9  private:
10   char * buf;
11 };
```

4.6. Canonical Form

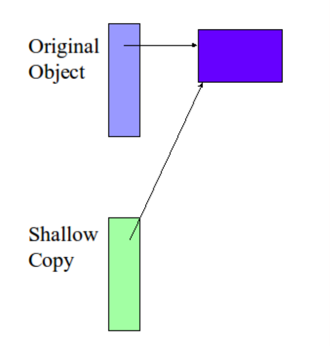
- James Coplien: a class with pointer data should be in *Canonical Form*, aka The Rule of Three, which means programmer writes:

  1. Copy constructor
  2. Copy assignment
  3. Destructor

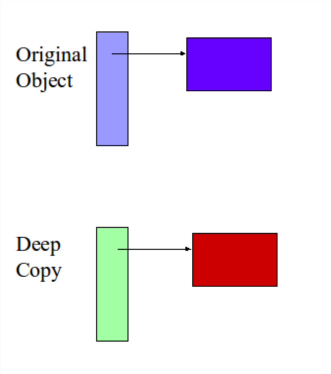
- Canonical form prevents shallow copy



4.7. Compiler generated ⇒ Shallow Copy



4.8. Canonical Form ⇒ Deep Copy



5. Why Prefer Init?

- Meyers, in Item #4 of Effective C++, says "prefer initialization to assignment" in ctors.
- The two examples in Sections 5.1 and 5.2 illustrate a considerable efficiency boost when using initialization rather than assignment.
- The two examples are exactly the same except for line 18:
  - Section 5.1, line 18, assignment::
    `TestAssign(char* n) { name = n; }`
  - Section 5.2, line 18, initialization list:
    `TestAssign(char* n) : name(n) { }`

5.1. assign Example

```
#include <iostream>
#include <cstring>
class string {
public:
  string() { std::cout << "default" << std::endl; }
  string(const char* b) { std::cout << "convert" << std::endl; }
  string(const string& s) { std::cout << "copy" << std::endl; }
  ~string() { std::cout << "destructor" << std::endl; }
  string& operator=(const string&) {
    std::cout << "assign" << std::endl;
    return *this;
  }
private:
  char* buf;
};
class TestAssign {
public:
  TestAssign(char* n) { name = n; }
private:
  string name;
};
int main() { TestAssign test("dog"); }
```

- The output for the previous program in Section 5.1 is:

```
default
convert
assign
destructor
destructor
```

- The first line of output, default, results when the compiler tries to initialize name in an initialization list. Since there isn't one, it uses the default constructor.

- The next two lines of output, convert and assign result from name = n, which doesn't match any function call as written. However, if n is **converted** to a string then it will match: string.operator=(string).
- The first destructor call results when the compiler reallocates the temporary string that was created with the convert.
- The final destructor call results when the compiler deallocates name in Student.

5.2. Init Example

```cpp
#include <iostream>
#include <cstring>
class string {
public:
  string() { std::cout << "default" << std::endl; }
  string(const char* b) { std::cout << "convert" << std::endl; }
  string(const string& s) { std::cout << "copy" << std::endl; }
  ~string() { std::cout << "destructor" << std::endl; }
  string& operator=(const string&) {
    std::cout << "assign" << std::endl;
    return *this;
  }
private:
  char* buf;
};
class TestInit {
public:
  TestInit(char* n) : name(n) { }
private:
  string name;
};
int main() { TestInit test("dog"); }
```


Overview
What is a class?
Constructors & . . .
What if I don't write . . .
Why Prefer Init?
Static Class Variabless
Principle of Least . . .
Interface vs . . .
Makefiles
Overload Operators
Slide 28 of 38
Go Back
Full Screen
Quit

- The output for the previous program in Section 5.2 is:

```
convert
destructor
```

- Clearly, the initialization list, `name(n)`, is a use of the conversion constructor in `string` to convert `n` to a string.

6. Static Class Variabless

•

7. Principle of Least Privilege

- A const class method cannot change any of the class data attributes.
- Use **const** as much as possible!
- Can reduce debugging
- Provides documentation
- Allow a function enough data access to accomplish its task and no more!
- Most beginners take them all out . . . probably need more!

7.1. Example of Least Privilege

```
class string {
public:
  string(const char* n) : buf(new char[strlen(n)+1]) {
    strcpy(buf, n);
  }
  const char* get() const { return buf; }
private:
  char *buf;
};
std::ostream&
operator<<(std::ostream& out, const string& s) {
  return out << s.get();
}
int main() {
  string x("Hello");
  std::cout << x.get() << std::endl;
}
```

7.2. What's wrong with this class?

```
class Student {
public:
  Student(const char * n) : name(n) { }
  const getName() const { return name; }
  void setName(char *n) { name = n; }
private:
  char *name;
};
```

8. Interface vs Implementation

Interface goes in .h file:

```
class Student {
public:
getName() const { return name; }
getGpa() const { return gpa; }
private:
  char * name;
  float gpa;
};
ostream& operator <<(ostream &, const Student &);
```

Implementation goes in .cpp file:

```
ostream & operator<<(ostream& out, const Student& s) {
  out << s.getName() << s.getGpa();
  return out;
}
```

9. Makefiles

- Useful as projects grow larger with multiple files.
- Consist of definitions,
- Followed by sequences of 2 line commands.
  - First line begins with $< id >$**:**, followed by dependencies of $< id >$.
  - Second line is the rule to make $< id >$; this line MUST be preceded by a tab
- To use the make file type: make $\{< id >\}$, or simply: **make**

9.1. Simple makefile

```
CCC=g++
FLAGS=-Wall

main:  main.o Binary.o
      $(CCC)  $(FLAGS) -o main main.o Binary.o

main.o: main.cpp Binary.h
      $(CCC)  $(FLAGS) -c main.cpp

Binary.o: Binary.cpp Binary.h
      $(CCC) $(FLAGS) -c Binary.cpp

clean:
      rm -f main *.o core
```

9.2. Discussion of Makefile

- $(CCC) permits us to easily switch to another compiler; e.g. clang++
- *make clean* will clean the directory of large files
- -o option creates an executable
- -c option creates .o file

10. Overload Operators

```
class string {
public:
  string();
  string(const char*);
  string(const string&);
  ~string();
  string operator+(const string&);
  string& operator=(const string&);
  char& operator[](int index);
  const char& operator[] const (int index);
private:
  char *buf;
};
ostream& operator<<(ostream&, const string&);
string operator+(const char*, const string&);
```

Overloaded operators will be described separately.