



Classes in C++98 and C++11

August 30, 2017

Brian A. Malloy



Overview

What is a class?

Constructors & . . .

What if I don't write . . .

Why Prefer Init?

Principle of Least . . .

Interface vs. . . .

Makefiles

Overload Operators



Slide 1 of 37

Go Back

Full Screen

Quit

1. Overview

- 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
- C++98 contained 3 types of constructors, but C++11 added a **move** constructor.
- These slides describe classes, including 3 of the 4 constructors.
- We describe **move** semantics in separate slides



Overview

What is a class?

Constructors & . . .

What if I don't write . . .

Why Prefer Init?

Principle of Least . . .

Interface vs. . . .

Makefiles

Overload Operators



Slide 2 of 37

Go Back

Full Screen

Quit



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

Overview

What is a class?

Constructors & . . .

What if I don't write . . .

Why Prefer Init?

Principle of Least . . .

Interface vs. . . .

Makefiles

Overload Operators



Slide 3 of 37

Go Back

Full Screen

Quit



Overview

What is a class?

Constructors & . . .

What if I don't write . . .

Why Prefer Init?

Principle of Least . . .

Interface vs. . . .

Makefiles

Overload Operators



Slide 4 of 37

Go Back

Full Screen

Quit

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
<pre>class Student { public: string name; float gpa; };</pre>	<pre>class Student { string name; float gpa; };</pre>

Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 5 of 37

Go Back

Full Screen

Quit



Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 6 of 37

Go Back

Full Screen

Quit

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)

Overview

What is a class?

Constructors & . . .

What if I don't write . . .

Why Prefer Init?

Principle of Least . . .

Interface vs. . . .

Makefiles

Overload Operators



Slide 7 of 37

Go Back

Full Screen

Quit

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;  
};
```



Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 8 of 37

Go Back

Full Screen

Quit



3.1. Default Constructor

```
1 class Student {  
2 public:  
3     string() : buf(new char[1]) { buf[0] = '\0'; }  
4 private:  
5     char* name;  
6 };
```

- 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?

Principle of Least ...

Interface vs. ...

Makefiles

Overload Operators



Slide 9 of 37

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;  
};
```

Overview

What is a class?

Constructors & ...

What if I don't write ...

Why Prefer Init?

Principle of Least ...

Interface vs ...

Makefiles

Overload Operators



Slide 10 of 37

Go Back

Full Screen

Quit



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):

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

Overview

What is a class?

Constructors & ...

What if I don't write ...

Why Prefer Init?

Principle of Least ...

Interface vs ...

Makefiles

Overload Operators



Slide 11 of 37

Go Back

Full Screen

Quit



3.4. Conversion Constructor

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

- Converts a `char*`, `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

Overview

What is a class?

Constructors & ...

What if I don't write ...

Why Prefer Init?

Principle of Least ...

Interface vs ...

Makefiles

Overload Operators



Slide 12 of 37

Go Back

Full Screen

Quit



3.5. Copy Constructor

```
1 class Student {
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* name;
9 };
```

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

Overview

What is a class?

Constructors & ...

What if I don't write ...

Why Prefer Init?

Principle of Least ...

Interface vs ...

Makefiles

Overload Operators



Slide 13 of 37

Go Back

Full Screen

Quit



Overview

What is a class?

Constructors & ...

What if I don't write ...

Why Prefer Init?

Principle of Least ...

Interface vs ...

Makefiles

Overload Operators



Slide 14 of 37

Go Back

Full Screen

Quit

3.6. Destructor

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



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 &);  
};
```

Overview

What is a class?

Constructors & ...

What if I don't write...

Why Prefer Init?

Principle of Least ...

Interface vs ...

Makefiles

Overload Operators



Slide 15 of 37

Go Back

Full Screen

Quit



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/dbl free

Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 16 of 37

Go Back

Full Screen

Quit



4.2. What can go wrong? Consider:

```
1 #include <iostream>
2 #include <cstring>
3 class string {
4 public:
5     string() : buf(new char[1]) { buf[0] = '\0'; }
6     string(const char * s) :
7         buf(new char[strlen(s)+1]) {
8         strcpy(buf, s);
9     }
10    ~string() { delete [] buf; }
11 private:
12     char * buf;
13 };
14 int main() {
15     string a, b(a);
16 }
```

Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 17 of 37

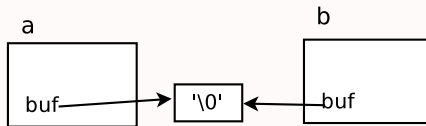
Go Back

Full Screen

Quit

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!

Overview

What is a class?

Constructors & . . .

What if I don't write . . .

Why Prefer Init?

Principle of Least . . .

Interface vs. . . .

Makefiles

Overload Operators



Slide 18 of 37

Go Back

Full Screen

Quit



Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 19 of 37

Go Back

Full Screen

Quit

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 };
```

Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 20 of 37

Go Back

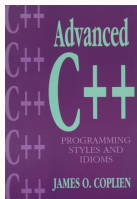
Full Screen

Quit



4.6. Canonical Form

- James Coplien: a class with pointer data should be in *Canonical Form*, which means it should include programmer written:
 - Copy constructor
 - Copy assignment
 - Destructor
- Canonical form prevents shallow copy



Overview

What is a class?

Constructors & . . .

What if I don't write . . .

Why Prefer Init?

Principle of Least . . .

Interface vs. . . .

Makefiles

Overload Operators



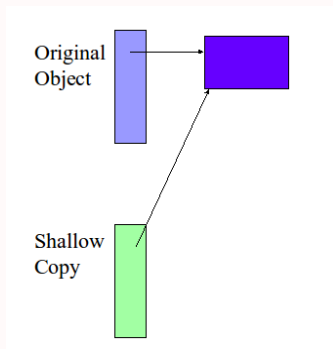
Slide 21 of 37

Go Back

Full Screen

Quit

4.7. Compiler generated \Rightarrow Shallow Copy



Overview

What is a class?

Constructors & . . .

What if I don't write . . .

Why Prefer Init?

Principle of Least . . .

Interface vs. . . .

Makefiles

Overload Operators



Slide 22 of 37

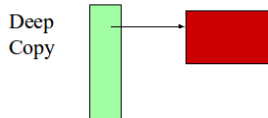
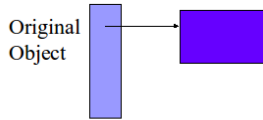
Go Back

Full Screen

Quit



4.8. Canonical Form \Rightarrow 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) { }`

Overview

What is a class?

Constructors & . . .

What if I don't write . . .

Why Prefer Init?

Principle of Least . . .

Interface vs. . . .

Makefiles

Overload Operators



Slide 24 of 37

Go Back

Full Screen

Quit



5.1. assign Example

```
1 #include <iostream>
2 #include <cstring>
3 class string {
4 public:
5     string() { std::cout << "default" << std::endl; }
6     string(const char* b) { std::cout << "convert" << std::endl; }
7     string(const string& s) { std::cout << "copy" << std::endl; }
8     ~string() { std::cout << "destructor" << std::endl; }
9     string& operator=(const string&) {
10         std::cout << "assign" << std::endl;
11         return *this;
12     }
13 private:
14     char* buf;
15 };
16 class TestAssign {
17 public:
18     TestAssign(char* n) { name = n; }
19 private:
20     string name;
21 };
22 int main() { TestAssign test("dog"); }
```

Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 25 of 37

Go Back

Full Screen

Quit

- 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.



Overview

What is a class?

Constructors & ...

What if I don't write ...

Why Prefer Init?

Principle of Least ...

Interface vs ...

Makefiles

Overload Operators



Slide 26 of 37

Go Back

Full Screen

Quit

- 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`.



Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 27 of 37

Go Back

Full Screen

Quit



5.2. Init Example

```
1  #include <iostream>
2  #include <cstring>
3  class string {
4  public:
5      string() { std::cout << "default" << std::endl; }
6      string(const char* b) { std::cout << "convert" << std::endl; }
7      string(const string& s) { std::cout << "copy" << std::endl; }
8      ~string() { std::cout << "destructor" << std::endl; }
9      string& operator=(const string&) {
10         std::cout << "assign" << std::endl;
11         return *this;
12     }
13 private:
14     char* buf;
15 };
16 class TestInit {
17 public:
18     TestInit(char* n) : name(n) { }
19 private:
20     string name;
21 };
22 int main() { TestInit test("dog"); }
```

Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 28 of 37

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.



Overview

What is a class?

Constructors & ...

What if I don't write ...

Why Prefer Init?

Principle of Least ...

Interface vs. ...

Makefiles

Overload Operators



Slide 29 of 37

Go Back

Full Screen

Quit



6. 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!

Overview

What is a class?

Constructors & . . .

What if I don't write . . .

Why Prefer Init?

Principle of Least . . .

Interface vs. . . .

Makefiles

Overload Operators



Slide 30 of 37

Go Back

Full Screen

Quit



6.1. 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;
}
```

Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 31 of 37

Go Back

Full Screen

Quit



6.2. 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;  
};
```

Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 32 of 37

Go Back

Full Screen

Quit



7. 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;  
}
```

Overview

What is a class?

Constructors & ...

What if I don't write ...

Why Prefer Init?

Principle of Least ...

Interface vs ...

Makefiles

Overload Operators



Slide 33 of 37

Go Back

Full Screen

Quit

8. 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**



Overview

What is a class?

Constructors & . . .

What if I don't write . . .

Why Prefer Init?

Principle of Least . . .

Interface vs . . .

Makefiles

Overload Operators



Slide 34 of 37

Go Back

Full Screen

Quit



8.1. 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
```

Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 35 of 37

Go Back

Full Screen

Quit



8.2. 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

Overview

What is a class?

Constructors & ...

What if I don't write ...

Why Prefer Init?

Principle of Least ...

Interface vs ...

Makefiles

Overload Operators



Slide **36** of **37**

Go Back

Full Screen

Quit



9. Overload Operators

```
1  class string {
2  public:
3      string();
4      string(const char*);
5      string(const string&);
6      ~string();
7      string operator+(const string&);
8      string& operator=(const string&);
9      char& operator[](int index);
10     const char& operator[] const (int index);
11 private:
12     char *buf;
13 };
14 ostream& operator<<(ostream&, const string&);
15 string operator+(const char*, const string&);
```

Overloaded operators will be described separately.

Overview

What is a class?

Constructors &...

What if I don't write...

Why Prefer Init?

Principle of Least...

Interface vs...

Makefiles

Overload Operators



Slide 37 of 37

Go Back

Full Screen

Quit