The C++ Class

January 4, 2016

Brian A. Malloy
1. What is a class?

- Unit of **encapsulation**:
  - Public operations
  - Private implementation

- **Abstraction**:
  - string: abstracts char* of C
  - student
  - sprite

- C++ Classes: easy to write, difficult to get right!

- Lots of examples
1.1. The actions of a class

- Initialize it’s data attributes
- Allocate memory when needed
- De-allocate memory when necessary
1.2. C++ class vs C++ struct

- Default access is only difference

<table>
<thead>
<tr>
<th>Bad class</th>
<th>Good Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Student {</td>
<td>class Student {</td>
</tr>
<tr>
<td>public:</td>
<td>string name;</td>
</tr>
<tr>
<td>string name;</td>
<td>float gpa;</td>
</tr>
<tr>
<td>float gpa;</td>
<td>}</td>
</tr>
<tr>
<td>}</td>
<td>}</td>
</tr>
</tbody>
</table>
1.3. Object: an instantiated class

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

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

- Or on the heap:

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

- Compiler does stack; programmer does heap!
2. Constructors & Destructor

- Constructors:
  - init data & allocate memory
  - Init data through initialization lists
- Destructors deallocate memory
- The three types of constructors are:
  1. Default
  2. Conversion
  3. Copy

```cpp
class Student {
public:
    Student();
    Student(char * n);
    Student(const Student&);
    ~Student();
};
```
2.1. Prefer **initialization** to assignment

- Initialization is more efficient for data members that are objects
- Only way to pass parameters to base class

```cpp
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;
};
```
2.2. Init performed in order of declare

class Student {
public:
    Student(int a) : age(a), iq(age+100) {} // Constructor.
private:
    int iq;
    int age;
};
2.3. Principle of Least Privilege

• Make “everything” const!
• Can reduce debugging
• Provides documentation
• Can prevent a member function from modifying data attributes
• Allow a function enough data access to accomplish its task and no more!
• Most beginners take them all out . . . probably need more!
2.4. Least Privilege example

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;
}
2.5. What operations does a class need?

1. All classes should have default constructor
2. Heap based data: *canonical form*:
   (a) Copy constructor
   (b) Destructor
   (c) Overloaded assignment

```cpp
class string {
public:
    string();
    string(const string&);
    ~string();
    string operator=(const string&);
private:
    char *buf;
};
ostream& operator<<(ostream&, const string&);
```
2.6. Why canonical form?
2.7. Why canonical form?
2.8. What can go wrong?

```cpp
1 #include <iostream>
2 #include <cstring>
3 using std::cout; using std::endl;
4
5 class string {
6   public:
7     string() : buf(new char[1]) { buf[0] = NULL; }
8     string(const char * s) : buf(new char[strlen(s)+1]) {
9       strcpy(buf, s);
10     }
11     ~string() { delete [] buf; }
12     const char* getBuf() const { return buf; }
13   private:
14     char * buf;
15   };
```

Looks like a well written class, but it is an accident waiting to happen!
2.9. Unseen Functions

Write this:

```cpp
class Empty{};
```

Get this:

```cpp
class Empty {
public:
    Empty();
    Empty(const Empty &);
    ~Empty();

    Empty& operator=(const Empty &);
    Empty * operator&();
    const Empty * operator&() const;
};
```
2.10. Here’s what they look like:

```cpp
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 assignment may induce a memory leak.
2.11. What’s wrong with this class?

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

private:
    char *name;
};
```
2.12. Practice: What’s the output?

class String {
public:
    String() { cout << "default" << endl; }
    String(char * n) { cout << "convert" << endl; }
    String(const String&) { cout << "copy" << endl; }
    ~String() { cout << "destructor" << endl; }
private:
    char * buf;
};
int main() {
    String a("cat"), b = a;
    String * ptr = new String("dog");
    return 0;
}
2.13. Practice: write class Student

```cpp
void fun(Student stu) {
    std::cout << stu.getName() << std::endl;
}

int main() {
    Student a, b(Darth Maul, 3.5), c = b;
    Student * d = new Student(Anakin, 4.0);
    cout << *d << endl;
    fun(a);
    return 0;
}
```
3. 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[](int index);
private:
    char *buf;
};

ostream& operator<<(ostream&, const string&);
string operator+(const char*, const string&);
3.1. An overloaded binary operator:

- Can be written in math form:
  
  ```cpp
  a = b;
  c = a + b;
  cout << stu;
  ```

- Or can be written in function invocation form:
  
  ```cpp
  a.operator=(b)
  c.operator=(a.operator+(b));
  cout.operator<<(stu)
  ```

- Man prefer the math form
3.2. How to overload assignment

Student & operator=(const Student & stu) {
    if (this == &stu) return * this;
    delete [] name;
    name = new char[strlen(stu.name)+1];
    strcpy(name, stu.name);
    gpa = stu.gpa;
    return *this;
}

(1) Why the comparison on the first line?
(2) Could the first line be: if (*this == stu)?
(3) Why return *this? What does it enable?
(4) Why not return stu, rather than *this?
3.3. Formula for overloaded assignment:

- Check for equality of lhs & rhs
- delete storage for lhs
- Create new storage for lhs, that's size of rhs
- Copy rhs stuff to lhs
- return *this
3.4. Overloading Operators

- Almost all operators can be overloaded
- Operators are binary or unary
- Have the same precedence as their compiler counterpart
- Can be members or friends
- Usually overloaded output operator should not be a member of a user defined class
3.5. Overloading output as \textit{friend}

class Student {
public:
    getName() const { return name; }
    getGpa() { return gpa; }
friend ostream&
    operator<<(ostream &, const Student &);
private:
    char * name;
    float gpa;
};
ostream&
operator<<(ostream& out, const Student& s) {
    out << s.name << \t << s.gpa;
    return out;
}
3.6. Overloading output as stand-alone:

class Student {
public:
    getName() const { return name; }
    getGpa() { return gpa; }
private:
    char * name;
    float gpa;
};
ostream &
operator<<(ostream& out, const Student& s) {
    out << s.getName() << \t << s.getGpa();
    return out;
}
4. Interface vs Implementation

Interface goes in .h file:

```cpp
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:

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

- global constants: ALL CAPS!
- local & global variables: ALL LOWER CASE, USE UNDERSCORE
- Class names: BEGIN EACH WORD WITH UPPER CASE, NO UNDERSCORE
- Class member functions: BEGIN LOWER CASE, then BEGIN EACH WORD WITH UPPER CASE
- Data members: SAME AS MEMBER FUNCTIONS
6. Makefiles

- 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>}`
6.1. 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
6.2. Discussion of Makefile

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

- Design a class for Student
- Write a class string, that encapsulates strings
- Write Binary, an abstraction for binary math.
- Write Stack, an abstraction of a stack.
- Design an experiment to see which is faster: your list or standard C++ library list?
- Faster: your string or standard C++ string?
- Faster: char* or standard C++ string?
7.1. Practice: What’s the output?

class String {
public:
    String() { cout << "default" << endl; }
    String(char * n) { cout << "convert" << endl; }
    String(const String&) { cout << "copy" << endl; }
    ~String() { cout << "destructor" << endl; }
    String& operator=(const String &) {
        cout << "assign" << endl;
    }
private:
    char * buf;
};
void fun(String mule) { cout << mule << endl; }
int main() {
    String a("cat"), b = a;
    String * ptr = new String("dog");
    fun(a);
    mule =(*ptr);
}
8. Template Classes

- Normal functions accept variables as parameters
- Template classes accept types as parameters
8.1. Template class Stack

template <class T>
class Stack {

public:
    Stack() : count(EMPTY) {}  
    void push(const T& n) { items[++count] = n; }
    void pop() { --count; }
    const T top() const { return items[count]; }
    bool isEmpty() const { return count == EMPTY; }
    bool isFull() const { return count == 99; }

private:
    enum {EMPTY = -1};
    T items[100];
    int count;
};