C++ Classes

The big picture before the details

- Provide a constructor (or the compiler will, and it will be wrong!)
- When your object holds dynamic data (pointers and new memory) you want to ensure deep copies so….
- YOU MUST IMPLEMENT
  - Destructor
  - Copy constructor
  - Assignment operator (one of many operators)

- If you are lazy make the copy and assignment operator private, then they cannot be copied or assigned
- RAII - Use the destructor to deallocate your dynamic memory

Outline

- Friend functions verses getters and setters
- Objects in Libraries
- Constructors
- Destructors
- What compiler creates for you
- Dynamic memory and objects
- RAII

Friend Functions

- Sometimes a function may require access to private parts of an object (like non member operator overloads do).
- Declaring the function to be a friend of the class, gives it access to protected and private members of the class without opening up access to anyone else.
- See project 9_getter_setter_friends_are_all_bad

Use case - using getters

```
#ifndef GETTERSETTERFRIEND_H_
#define GETTERSETTERFRIEND_H_

class GetterSetterFriend {
private:
    int i;
public:
    GetterSetterFriend(int i);
    virtual ~GetterSetterFriend();

    //a horrid little getter
    int geti();
};

bool isEqual_using_getter(GetterSetterFriend &gsf1,
        GetterSetterFriend &gsf2);

#endif /* GETTERSETTERFRIEND_H_ */
```

```
#include "GetterSetterFriend.h"

GetterSetterFriend::GetterSetterFriend(int i):i(i) {
}

GetterSetterFriend::~GetterSetterFriend() {
    // TODO Auto-generated destructor stub
}

int GetterSetterFriend::geti(){
    return i;
}


bool isEqual_using_getter(GetterSetterFriend &gsf1,
        GetterSetterFriend &gsf2){
    //note getter access (can read only)
    return (gsf1.geti() < gsf2.geti());

}
```

isEqual - using Friend Functions, sorta fixes getter and setter

```
#ifndef GETTERSETTERFRIEND_H_
#define GETTERSETTERFRIEND_H_

class GetterSetterFriend {
private:
    int i;
public:
    GetterSetterFriend(int i);
    virtual ~GetterSetterFriend();

    //a friend which exposes internal data
    friend bool isEqual_friend(GetterSetterFriend &gsf1,
            GetterSetterFriend &gsf2);
};

#endif /* GETTERSETTERFRIEND_H_ */
```

```
#include "GetterSetterFriend.h"

GetterSetterFriend::GetterSetterFriend(int i):i(i) {
}

GetterSetterFriend::~GetterSetterFriend() {
    // TODO Auto-generated destructor stub
}

//a friend func, note that its not part of the class
//but still has access to class internals
bool isEqual_friend(GetterSetterFriend &gsf1,
        GetterSetterFriend &gsf2){
    //note direct access (can read/write)
    //gsf1.i = 9;   //demo direct private member access
    return (gsf1.i < gsf2.i);
}
```

Friend function of a class

- It is not a member function
- Has direct access to private members
- Just as member functions do

Use keyword *friend* in front of function declaration
Specified IN class definition
But they're NOT member functions!

Friends - summary

- Break encapsulation (but not as bad as getters)
- In the .h file
  - Friend function declared in class with prefix ‘friend’
- In the .cpp file
  - Friend function definition

Objects in Libraries

- See 9_Library_class_demo and 9_Library_class
- *Link* just like a library full of functions
- *Use* just like a library full of functions

Constructors (review)

- Default constructor (no arguments)
  - Classname::classname()
- Overloaded constructors (with arguments)
  - Classname::classname(type varName,…)
- Copy constructor
- Assignment operator

Get to these in a bit

- Constructors set up the object for use

What to do in constructors?
At a minimum initialize member variables!

Constructors- Member Initialization

- If you do NO initialization
  - For Objects – their default constructor is called
  - Primitives (ints, bools, doubles, longs, char etc) – NO INITIALIZATION AT ALL ….Best to initialize…

```
class NoMemberInitilization {
private:
    std::string myString;
    int         myInt;

public:
    NoMemberInitilization();
    virtual ~NoMemberInitilization();
};
```

```
NoMemberInitilization::NoMemberInitilization() {
    //myStrings no argument (default) constructor called set to 0
    //myInt has garbage in it
}
```

You don't make the 'forget to initialize' mistake when you first write object.

Its when you add data members later that you forget to initialize

Constructors – member initialization

- Could initialize in constructor body

```
NoMemberInitilization::NoMemberInitilization() {
    myString = "";
    myInt=0;
}
```

- But, construction of objects proceeds in 2 phases
  - 1. Initialization of Data members
  - 2. Execution of the body of the constructor that was called
- So if you initialize in constructor body, you initialize an object with default constructor then assign in constructor body.
- 2 calls!
- Also what if any members vars are const?

Constructors – member initialization THE CORRECT WAY

- Use initializer list
- Uses copy constructor
- 1 call

```
NoMemberInitilization::NoMemberInitilization():myString(""),myInt(0) {
```

- And what if member vars are const?

```
class NoMemberInitilization {
private:
    std::string myString;
    const int         myInt;

public:
    NoMemberInitilization();
    virtual ~NoMemberInitilization();
};
```

Then you MUST use initializer lists

See 9_Classes InitializerList.cpp and .h

Destructor (review)

- A function that gets called when an object is destroyed.
- Called when object goes out of scope (whether statically or dynamically allocated)
- Its purpose is to clean up after object
  - dynamically allocated memory that the object has pointers to
  - Close open filestreams
  - Close database connections
  - Close network connections
- Syntax:

```
classname::~classname();
```

See 9_library_class – class_destructor

Destructor
Objects with dynamic memory (review)

- If dynamic memory allocated deallocate it.

```
HoldsDynamicData::~HoldsDynamicData() {
    if (ps)
        delete[] ps;
}
```

- **Must** write yourself if object has dynamically allocated members, or object members that cannot make a copy of themselves.
- Otherwise let compiler handle it

Shallow and Deep Copies

- Shallow copy
  - Assignment copies only member variable contents over (so only pointer addresses copied, NOT the data pointed to)
  - This is how default (compiler generated) assignment and copy constructors work
  - Fine if No dynamic memory involved
- Deep copy
  - Pointers, dynamic memory involved
  - Must dereference pointer variables to "get to" data for copying.
  - And then copy that data

When your object holds dynamic data

- YOU MUST IMPLEMENT
  - **Class destructor**
    - Special member function
    - Automatically destroys objects
  - **Copy constructor**
    - Single argument member function
    - Called automatically when temp copy needed
    - Like when you pass an object to a function by value
    - MUST DO DEEP COPY
  - **Assignment operator**
    - Must be overloaded as member function
    - MUST DO DEEP COPY

Compiler created Functions

- Given this class

```
 * defaultClass.cpp

#include "defaultClass.h"
defaultClass::defaultClass(int i) {
    this->i=i;
}

defaultClass::~defaultClass() {
}
```

```
 * defaultClass.h

#ifndef DEFAULTCLASS_H_
#define DEFAULTCLASS_H_
class defaultClass {
private:
    int i;
public:
    defaultClass(int i);
    virtual ~defaultClass();
};
#endif /* DEFAULTCLASS_H_ */
```

- Why does this work? (see 9 classes)

```
//why does this work
defaultClass d11(1);     //1 arg constructor
defaultClass d22(d11);   //copy constructor
defaultClass d33(2);
d33 = d11;               //assignment operator
```

Where is the copy constructor and assignment operator? I did not write it.

- Compiler did.
- It will invisibly write copy constructor, assignment operator destructor for you and others if needed
- Does ‘shallow’ copy (variable to variable)
  1. Fine if class has no dynamically allocated memory
  2. or all member variables know how to make copy of themselves
- What if you have dynamic data, or ignorant variables?
  - Show demo (9_classes project – **HoldsDynamicData.cpp**)
- Need a ‘deep’ copy (dynamic mem to new dynamic mem)
- Must write these 3 functions if have dynamic member vars
  - Copy constructor
  - Assignment operator
  - Destructor

Copy Constructor
Objects with dynamic memory

- A special constructor that is used to make a copy of an existing instance

```
//copy constructor
HoldsDynamicData(const HoldsDynamicData& other);
```

- Where is it used?
  - Initializer lists (Constructing a new instance from another)
  - Pass by value to a function
  - Show demo (9_classes holdsdynamicdata.cpp)

- **Must** write yourself if object has dynamically allocated members, or object members that cannot make a copy of themselves.
- Otherwise let compiler handle it

Assignment Operator
Objects with dynamic memory

- One of many operators

```
//assignment operator
HoldsDynamicData & operator= (const HoldsDynamicData & other);
```

- Used with =
- Show demo (9_classes holdsdynamicdata.cpp)

- **Must** write yourself if object has dynamically allocated members, or object members that cannot make a copy of themselves.
- Otherwise let compiler handle it

Yikes! How can I remember all this

- Follow the template in
- 9_copy_and_assign_template

Don’t want to implement?

- If you don’t want to implement copy constructor or assignment operator (or others as well)
- Why not just not write them?
  - Because compiler will if you dont

- Solution: Mark them as private
- But friend functions, and class methods can still access
- Solution: Do not define them, then they are declared so compiler will not generate them, but if you try to call them anywhere, You get a linker error.

Automatic memory Management

- RAII (Resource Acquisition Is Initialization)
- Object manages dynamic memory allocation
- As well as deletion, (you cant lose)

```
RAII::RAII(const char *ps) {
    if (ps) {
        int len = strlen(ps) + SPACE_FOR_SLASH0;     //how much?
        pmyString = new char[len];                   //allocate it
        strncpy(pmyString, ps, len);                 //copy, include '/0'
    }
    else
    {
        pmyString  = new char[1];
        *pmyString = '\0';
    }
}

RAII::~RAII() {
        delete[] pmyString;
}
```

Summary

- When your object holds dynamic data – YOU MUST IMPLEMENT
  - Destructor
  - Copy constructor
  - Assignment operator (one of many operators)