

CSCI 104 Classes

Mark Redekopp

Revised: 05/05/2021

Review from CS 103 [1]

Const function arguments

- Will this code compile?
- Indicate what will be printed (assuming it compiles)

```
void f1(const vector<int>& x){
    x.push_back(103);
    x.push_back(104);
}

void f2(string& y){
    y = "Bye";
}

int main()
{
    vector<int> a; string b = "Hi";
    f1(a);
    f2(b);
    cout << b.size() << endl;
    return 0;
}
```

Const member functions

- What does the highlighted const keyword imply in the code below?

```
class Item
{ int val;
public:
    void foo();
    int bar() const;
};

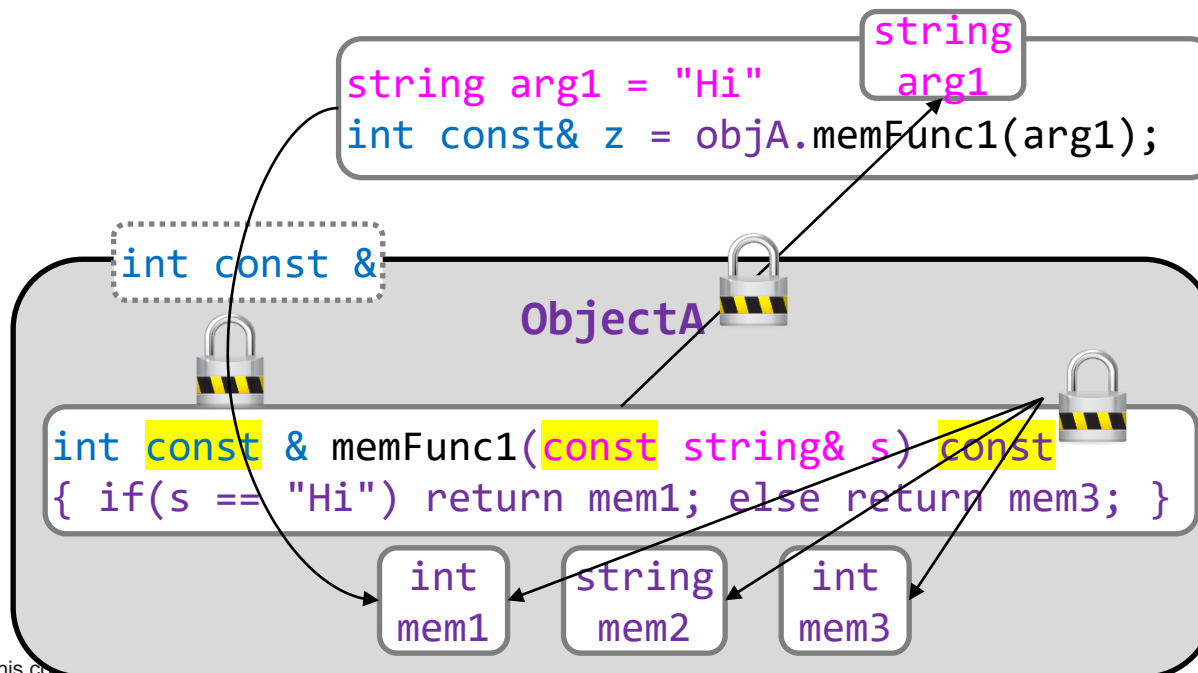
void Item::foo()
{ val = 5; }

int Item::bar() const
{ return val+1; }

void f1(const Item& arg) {
    int x = arg.bar(); // fine
    arg.foo(); // Compiler Error!
}
```

'const' Keyword

- const keyword can be used with
 1. Input arguments to ensure they aren't modified
 2. After a member function to ensure data members aren't modified by the function
 3. Return values to ensure they aren't modified



Exercises

- `cpp/cs104/classes/const_members`
- `cpp/cs104/classes/const_members2`
- `cpp/cs104/classes/const_return`

Review from CS 103 [2]

Constructor Initialization Lists

- What is the most efficient means to initialize the vals member to an initial array size of 20 and s to a user-defined argument?

```
class Thing {
public:
    Thing(const std::string& s_init);
private:
    vector<int> vals;
    string s;
};

Thing::Thing(const std::string& s_init)
{
    // is this the most efficient way?
    vals.resize(20);
    s = s_init;
}
```

Construction Order

- What is printed by the code below?

```
class ABC {
public:
    ABC() { cout << "ABC" << endl; }
};
class DEF {
public:
    DEF() { cout << "DEF" << endl; }
};
class XYZ {
    ABC m1; DEF m2;
public:
    XYZ()
        { cout << "XYZ" << endl; }
};
int main() {
    XYZ x1;
    return 0;
}
```

Review from CS 103 [3]

Friend Functions

- What does the highlighted friend keyword imply in the code below?
- What would break if we remove it?

```
class Complex
{
public:
    Complex();
    Complex(double r, double i);
    friend Complex operator+(const int&, const Complex&);
private:
    double real, imag;
};

Complex operator+(const int& lhs, const Complex &rhs)
{
    Complex temp;
    temp.real = lhs + rhs.real;    temp.imag = rhs.imag;
    return temp;
}
```

Friend Classes

- Can DEF::clear() access obj.x?
- If not, how can class ABC grant access to DEF?

```
class ABC {
    int x;    // data member
public:

    ...
};

class DEF {
public:
    void clear(ABC& obj) { obj.x = 0; }
};
```

NESTED TYPES

Duplicate Types

- Recall linked lists use a helper struct to model each item in the list
 - Stores a value (of a certain type) and a pointer to the next
- If we want to use a different type list, we would need a different Item struct, but would need to name it differently
- Solution:
 - Different names: IntItem vs. DoubleItem
 - Templates (more later)

```
// integer linked list
struct Item {
    int val;
    Item* next;
};

class ListInt
{
public:
    ListInt();
    ~ListInt();
    void append(int v); ...
private:
    Item* head_;
};
```

```
// double linked list
struct Item {
    double val;
    Item* next;
};
// ERROR - Duplicate type name
```

– Nested Types!!

Nested Types

- A struct or class can be defined inside another and is known as a nested type
- Good practice to nest 'helper' types (i.e. structs/classes that exist SOLELY in support of the outer class)
- Nested types can share the **same name** but have **different implementations** when defined inside of **different** objects
- Examples:
 - Linked list Item struct
 - Iterators (later in the class)

```
// integer linked list
class ListInt {
public:
    // Define a nested type
    struct Item {
        int val;
        Item* next;
    };
    Item* find(int x) const;
```

```
private:
    Item* head_;
};

// double linked list
class ListDbl {
public:
    // Nested type
    struct Item {
        double val;
        Item* next;
    };
    Item* find(double x) const;
```

```
int main()
{
    ListInt::Item x;
    x.val = 3;

    ListDbl ld;
    // ...
    ListDbl::Item* p;
    p=ld.find(2.5);
}
```

Declaring and Using Nested Types

- Non-members must scope the type name:
 - `classname::typename`
- Member function `code` do not have to scope the type once inside the member function scope
 - Notice the `return type` of a function is not inside the member function scope

```
class ListInt {
public:
    // Define a nested type
    struct Item {
        int val; Item* next;
    };
    ListInt();
    void append(int v);
    Item* find(int v) const;
private:
    Item* head_;
};

void ListInt::append(int v){
    Item* x = new Item; // no scoping
}
// requires scoping the type
ListInt::Item* ListInt::find(int v) const
{ ... }

int main()
{
    ListInt mylist;
    // requires scoping the type
    ListInt::Item* p = mylist.find(2);
    // ...
}
```

STATIC MEMBERS

One For All

- As USCStudent objects are created we want them to have unique IDs
- How can we accomplish this?

```
class USCStudent {  
    public:  
        USCStudent(string n) : name(n)  
        {   id = _____ ; // ????  
        }  
  
    private:  
        string name;  
        int id;  
}  
  
int main()  
{  
    // should each have unique IDs  
    USCStudent s1("Tommy");  
    USCStudent s2("Jill");  
  
}
```

One For All

- Can we just make a counter data member of the USCStudent class?
- What's wrong with this?

```
class USCStudent {  
    public:  
        USCStudent(string n) : name(n)  
        { id = id_cntr++; }  
  
    private:  
        int id_cntr;  
        string name;  
        int id;  
}  
  
int main()  
{  
    USCStudent s1("Tommy"); // id = 1  
    USCStudent s2("Jill");  // id = 2  
  
}
```

One For All

- It's not something that we can do from within an instance
 - A student doesn't assign themselves an ID, they are told their ID
- Sometimes there are functions or data members that make sense to be part of a class but are shared (only 1 exists) amongst all instances
 - The variable or function doesn't depend on the instance of the object, but just the general class (family of objects)
 - We can make these 'static' members which means one definition shared by all instances

```
class USCStudent {
public:
    USCStudent(string n) : name(n)
    { id = id_cntr++; }

private:
    static int id_cntr;
    string name;
    int id;
}

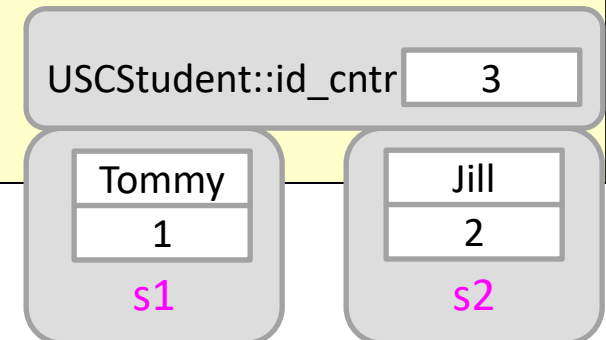
// initialization of static member
int USCStudent::id_cntr = 1;

int main()
{
    USCStudent s1("Tommy"); // id = 1
    USCStudent s2("Jill");  // id = 2
    ...
}
```

Static Data Members

- A **static** data member is a single variable that all instances of the class share
- Can think of it as belonging to the class and not each instance
- Declare with keyword **static**
- Initialize outside the class in a .cpp (can't be in a header)
 - Must be scoped with class name

```
class USCStudent {  
public:  
    USCStudent(string n) : name(n)  
    { id = id_cntr++; }  
  
private:  
    static int id_cntr;  
    string name;  
    int id;  
}  
  
// initialization of static member  
int USCStudent::id_cntr = 1;  
  
int main()  
{  
    USCStudent s1("Tommy"); // id = 1  
    USCStudent s2("Jill");  // id = 2  
    ...  
}
```



Example: Class Constants (string::npos)

- Sometimes there are constants that are useful to define for a class but the same value for all instances
- `std::string::npos` is such a constant
 - Used as an input value for a length parameter that means *"until the end of the string"*
 - Returned by a call to `string::find()` or `string::rfind()` to indicate *"no match"*

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s1 = "cs104";
    if(s1.find("103") == string::npos)
    {
        cout << "We're not in 103 "
              "anymore" << endl;
    }
    return 0;
}

// Note: in the above example,
// C++ automatically concatenates
// multiple string constants on
// different lines if not
// separated by any operator
```


Example: Class Constants (string::npos)

- `std::string::npos` is set to the largest unsigned value supported by the system (all 1s in binary) which can be achieved by casting -1 (which is all 1s in signed binary) to an unsigned value

```
// simplified string class
class string {
public:
    static const size_t npos;
    ...
};

const size_t string::npos = (size_t)-1;
```

Another Example: Singleton

- In addition, to **static data members**, **static member functions** are also allowed
- Does NOT take a `this` pointer (not executing on an instance)
 - Called by scoping with the class name
- Can access private members of the class

```
class President {
public:
    static President* makePresident(string name);
    void printName() const { cout << name_ << endl; }
private:
    string name_; // representative of data member
    // private to disallow other instances
    President(string name) : name_(name) {}
    static President* thePres; // THE president
};
// init static member
President* President::thePres = nullptr;

President* President::makePresident(string name) {
    if(nullptr == thePres){
        // calls private constructor
        thePres = new President(name);
    }
    return thePres;
}

int main() {
    President* p = President::makePresident("Carol");
    President* p2 = President::makePresident("Mark");
    p->printName(); // prints "Carol"
    p2->printName(); // still "Carol"
    return 0;
}
```

A Related Example

- All US Citizens share the same president, though it changes over time
- Rather than wasting memory for each citizen to store a pointer to the president, we can make it static
- However, private static members can't be accessed from outside functions
- For this we can use a static member functions

```
class USCitizen{
public:
    USCitizen();

private:
    static President* pres;
    string name;
    int ssn;
}

int main()
{
    USCitizen c1;
    USCitizen c2;
    President* curr = new President;

    // won't compile..pres is private
    USCitizen::pres = curr;
}
```

Static Member Functions

- Static member functions do tasks at a class level and can't access data members (since they don't belong to an instance)
- Call them by preceding with 'className::'
- Use them to do common tasks for the class that don't require access to an instance's data members
 - Static functions could really just be globally scoped functions but if they are really serving a class' needs it makes sense to group them with the class

```
class USCitizen{
public:
    USCitizen();
    static void setPresident(President* p)
    {   pres = p;   }

private:
    static President* pres;
    string name;
    int ssn;
}

int main()
{
    USCitizen c1;
    USCitizen c2;
    President* curr = new President;
    USCitizen::setPresident(curr);
    ...
    President* next = new President;
    USCitizen::setPresident(next);
}
```

DEFAULT ARGUMENTS

Default Arguments

- Default arguments can be provided
 - User can provide a different value or not provide any, in which case the default is taken
- Only list the default argument in the prototype **but not both** the prototype and definition

```
class IntVector {
public:
    // usually put default arg in prototype
    IntVector(size_t n = 10);
    ...
private:
    size_t n_; int* array;
};

// Should not repeat the default arg
IntVector::IntVector(size_t n) : n_(n)
{
    array = new int[n_];
}

int main()
{
    // both call the same constructor above
    IntVector vec1(50); // size 50
    IntVector vec2;    // will use default 10
    ...
}
```

Other Limitations

- You can have many default arguments, but they must terminate the argument list; a non-default argument CANNOT come AFTER a default argument
- Ensure that two functions signature is not ambiguous

```
// good
void good1(int a, int b = 10, string s = "hi");

// bad (non-default arg, s, after default arg, b)
void bad1(int a, int b = 10, string s);

// bad - ambiguous with other func due to default args()
void good1(int b);
```

```
17:12: error: call of overloaded 'good1(int)' is ambiguous
17:12: note: candidates are:
4:6: note: void good1(int, int, std::string)
12:6: note: void good1(int)
```

CONDITIONAL COMPILATION

Multiple Inclusion

- Often separate files may #include's of the same header file
- This may cause compiling errors when a duplicate declaration is encountered
 - See example
- Would like a way to include only once and if another attempt to include is encountered, ignore it

```
class string{  
... };
```

string.h

```
#include "string.h"  
class Widget{  
public:  
    string s;  
};
```

widget.h

```
#include "string.h"  
#include "widget.h"  
int main()  
{ }
```

main.cpp

```
class string { // inc. from string.h  
};  
  
class string{ // inc. from widget.h  
};  
class Widget{  
... }  
int main()  
{ }
```

main.cpp after preprocessing

Conditional Compiler Directives

- Compiler directives start with '#'
 - #define XXX
 - Sets a flag named XXX in the compiler
 - #ifdef, #ifndef XXX ... #endif
 - Continue compiling code below until #endif, if XXX is (is not) defined
- Encapsulate header declarations inside a
 - #ifndef XX
 - #define XX
 - ...
 - #endif

```
#ifndef STRING_H
#define STRING_H
class string{ ... };
#endif
```

string.h

```
#include "string.h"
class Widget{
public:
    string s;
};
```

widget.h

```
#include "string.h"
#include "string.h"
```

main.cpp

```
class string{ // inc. from string.h
};
class Widget{ // inc. from widget.h
...
}
```

main.cpp after preprocessing

Conditional Compilation

- Often used to compile additional DEBUG code
 - Place code that is only needed for debugging and that you would not want to execute in a release version
- Place code in a `#ifdef NAME...#endif` bracket
- Compiler will only compile if a `#define NAME` is found
- Can specify `#define` in:
 - source code
 - At compiler command line with `(-DNAME)` flag
 - `g++ -o stuff -DDEBUG stuff.cpp`

```
int main()
{
    int x, sum=0, data[10];
    ...
    for(int i=0; i < 10; i++){
        sum += data[i];
#ifdef DEBUG
        cout << "Current sum is ";
        cout << sum << endl;
#endif
    }

    cout << "Total sum is ";
    cout << sum << endl;
}
```

stuff.cpp

```
$ g++ -o stuff -DDEBUG stuff.cpp
```

PRE-SUMMER 2021 SLIDES

OVERVIEW AND CONCEPTS

C Structs vs. Classes

- Needed a way to group values that are related, but have different data types
- NOTE: struct has changed in C++!
 - C
 - Only data members
 - Some declaration nuances
 - C++
 - Like a class (data + member functions)
 - Default access is **public** where as class' default to **private**

```
struct Person{
    char name[20];
    int age;
};

int main()
{
    // Anyone can modify
    // b/c members are public
    Person p1;
    p1.age = -34;
    // probably not correct

    return 0;
}
```

Classes & OO Ideas

- Classes are used as the primary way to organize code
- Encapsulation
 - Place data and operations on data into one code unit
 - Protect who can access data via private members
- Abstraction
 - Depend only on an interface regardless of implementation to create low degree of **coupling** between different components
 - Ex. USB interface (any USB device can plug into many different kinds of computer systems)
- Unit of composition
 - Create really large and powerful software systems from tiny components
 - Define small pieces that can be used to compose larger pieces
 - Delegation/separation of responsibility
- Polymorphism & Inheritance
 - More on this later...

Protect yourself from users & protect your users from themselves

```
class Deck {  
public:  
    Deck();    // Constructor  
    ~Deck();   // Destructor  
    void shuffle();  
    void cut();  
    int get_top_card();  
private:  
    int cards[52];  
    int top_index;  
};
```

deck.h

```
#include<iostream>  
#include "deck.h"  
  
int main(int argc, char *argv[]) {  
    Deck d;  
    int hand[5];  
  
    d.shuffle();  
    d.cut();  
  
    d.cards[0] = ACE; //won't compile  
    d.top_index = 5;  //won't compile  
}
```

cardgame.cpp

Coupling

- Coupling refers to how much components depend on each other's implementation details (i.e. how much work it is to remove one component and drop in a new implementation of it)
 - Placing a new battery in your car vs. a new engine
 - Adding a USB device vs. a new video adapter to your laptop
- OO Design seeks to reduce coupling as much as possible by
 - Creating well-defined interfaces to update (write) or access (read) the state of an object
 - Allow alternate implementations that do NOT require interface changes

PARTS OF A CLASS

Parts of a C++ Class

- What are the main parts of a class?
 - Data members
 - What data is needed to represent the object?
 - Constructor(s)
 - How do you build an instance?
 - Member functions
 - How does the user need to interact with the stored data?
 - Destructor
 - How do you clean up an after an instance?

```
class IntLinkedList {  
    public:  
        IntLinkedList( );  
        IntLinkedList( int n ) ;  
        ~IntLinkedList( );  
        void prepend(int n);  
        void remove(int toRemove);  
        void printList();  
        void printReverse();  
    private :  
        void printHelper(Item *p);  
        Item *head;  
};
```

Notes About Classes

- Member data can be **public** or **private** (and later **protected**)
 - Defaults is private (only class functions can access)
 - Must explicitly declare something public
- Most common C++ operators will not work by default (e.g. `==`, `+`, `<<`, `>>`, etc.)
 - You can't cout an object (`cout << myobject;` won't work)
 - The only one you get for free is `'=`' and even that may not work the way you want (more on this soon)
- Classes may be used just like any other data type (e.g. `int`)
 - Get pointers/references to them (`Obj*`, `Obj&`)
 - Pass them to functions (by copy, reference or pointer)
 - Dynamically allocate them (`new Obj`, `new Obj[100]`)
 - Return them from functions (`Obj f1(int x);`)

C++ Classes: Constructors

- Called when a class is instantiated allowing you to initialize data members to desired values
- No return value
- **Default (no argument) Constructor**
 - Can have one or none in a class
 - Signature: `ClassName()`;
 - If class has no constructors, C++ will make a default
 - But it is just an empty constructor (e.g. `Student::Student() {}`)
 - When arrays of an Object are declared, C++ automatically calls default constructor on each array element
- **Overloaded/Initializing Constructors**
 - Can have zero or more
 - These constructors take in arguments
 - **Appropriate version is called based on how many and what type of arguments are passed when a particular object is created**
 - If you define a constructor with arguments you **should also** define a default constructor (otherwise no default constructor will be available)

```
class Student {  
public:  
    // Default constructor  
    Student( );  
  
    // Initializing constructor  
    Student(const string& name);  
  
    // Destructor  
    ~Student( );  
    ...  
private:  
    string name_;  
    int id_;  
    vector<int> grades_;  
};
```

Examples of Constructors

1

```
class Obj {
public:
    // no user-defined constructor
    void setNum(int n);
    string getStr();
    int num; string s1;
};
```

1

```
class Obj {
public:
    // Initializing constructor
    Obj(int n, string s)
    { num = n; s1 = s; }
    void setNum(int n);
    string getStr();
    int num; string s1;
};
```

2

```
class Obj {
public:
    Obj() { }
    // compiler generated
    // default constructor

    void setNum(int n);
    string getStr();
    ...
};
```

2

```
class Obj {
public:
    Obj() { }
    // compiler does not generate
    // constructor

    void setNum(int n);
    string getStr();
    ...
};
```

3

```
int main() {
    Obj x; // calls
           // default constructor
}
```

3

```
int main() {
    Obj x, y[100]; // no arrays
    // if no def. constructor
    Obj y(5, "hi");
}
```

Identify that Constructor

- Prototype what constructors are being called here
- s1
 - Student::_____
- s2
 - Student::_____
- dat
 - vector<int>::_____

```
class Student {
public:
    // Default constructor
    Student( );

    // Initializing constructor
    Student(const string& name);
    ...
private:
    string name_;
    int id_;
    vector<int> grades_;
};

int main()
{
    Student s1;
    Student s2("Tommy");
    // note: anything in "" is
    // type const char*
    vector<int> vals(10);
    ...
}
```

Identify that Constructor

- Prototype what constructors are being called here
- `s1`
 - `Student::Student()`
 // default constructor
- `s2`
 - `Student::Student(const char*)`
- `dat`
 - `vector<int>::vector<int>(int);`

```
class Student {
public:
    // Default constructor
    Student( );

    // Initializing constructor
    Student(const string& name);

    ...
private:
    string name_;
    int id_;
    vector<int> grades_;
};

int main()
{
    Student s1;
    Student s2("Tommy");
    // note: anything in "" is
    // type const char*
    vector<int> vals(10);

    ...
}
```

Initializing data members of a class

CONSTRUCTOR INITIALIZATION LISTS

Consider this Struct/Class

- Examine this struct/class definition...
 - How can I initialize the members?

```
#include <string>
#include <vector>

struct Student
{ std::string name;
  int id;
  std::vector<double> scores;
  // say I want 10 test scores per student

  Student(); // default constructor

  Student(std::string n, int ident);
  // initializing constructor
};

int main()
{
  Student s1;
  Student s2("Tommy", 12345);
}
```

string name
int id
scores

Composite Objects

- **Fun Fact 1:** Before the constructor of an object executes, all of its data members must be constructed
 - Before a baby is born all its organs must develop and start working
- **Fun Fact 2:** Constructors for objects get called (and can ONLY EVER get called) at the time of creation (when memory is allocated)
 - Once the object's constructor starts executing, it is too late to call data members' constructors. The data members have already been constructed.

```
#include <string>
#include <vector>

struct Student
{
    std::string name;
    int id;
    std::vector<double> scores;
    // say I want 10 test scores per student

    Student()
    // constructors for members called here
    {
        // TOO LATE TO CALL DATA MEMBER
        // CONSTRUCTORS
        name("Tommy Trojan");
        id = 12313;
        scores(10);
    }
};

int main()
{ Student s1; // memory for Student allocated
  //...
}
```

string name

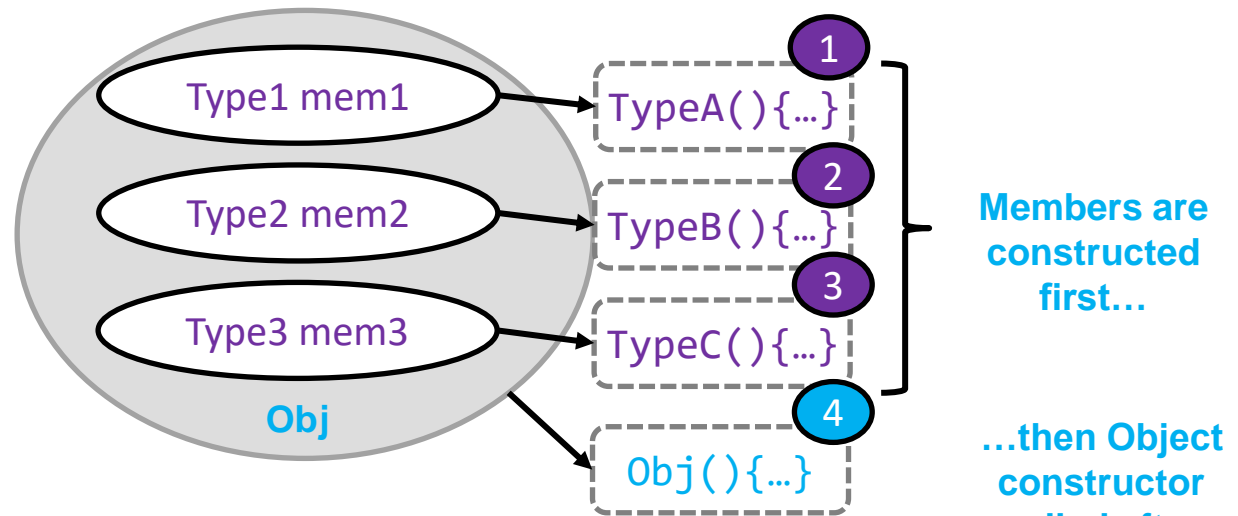
int id

scores

Initializing Members

- When an object is constructed the individual members are constructed first
 - Member constructors are called **BEFORE** object's constructor

```
Class Obj
{ public:
  Obj();
  // public members
private:
  Type1 mem1;
  Type2 mem2;
  Type3 mem3;
};
```



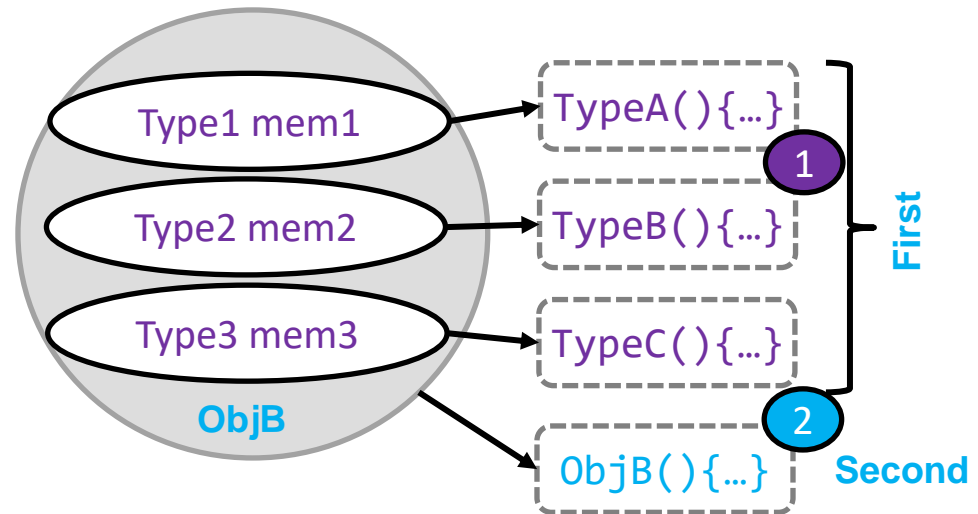
Allocating and Deallocating Members

- Members of an object have their constructor called automatically **BEFORE** the object's constructor executes

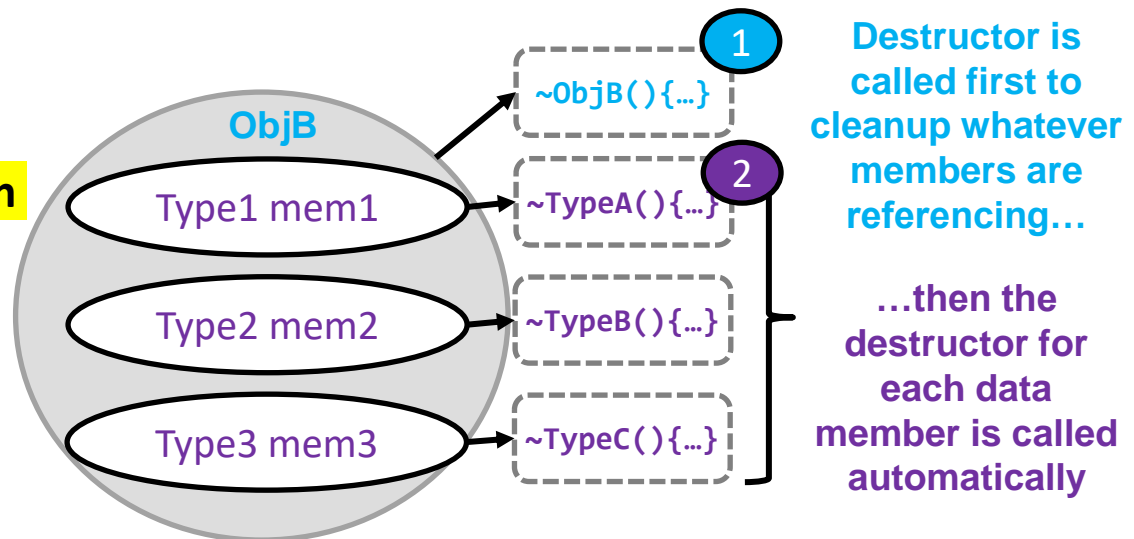
- Construction works **inside-out (from smaller to larger)**

- When an object is destructed the members are destructed automatically **AFTER** the object's destructor runs

- Destruction works **outside-in (from larger to smaller)**



Construction



Destruction

Old Initialization Approach

```
Student::Student()  
{  
    name = "Tommy Trojan";  
    id = 12313  
    scores.resize(10);  
}
```

If you write this...


```
Student::Student() :  
    name(), id(), scores()  
    // calls to default constructors  
{  
    name = "Tommy Trojan"; // now modify  
    id = 12313  
    scores.resize(10);  
}
```

The compiler will still generate this.

- Though you do not see it, realize that the **default constructors** are implicitly called for each data member before entering the {...}
- You can then assign values (left side code)
 - But this is a **2-step** process: default construct, then replace with desired value

New Initialization Approach

```
Student::Student() :  
    name(), id(), scores() /* compiler generated */  
{  
    name = "Tommy Trojan";  
    id = 12313  
    scores.resize(10);  
}
```



Default constructors implicitly called and then values reassigned in constructor

```
Student::Student() :  
    name("Tommy"), id(12313), scores(10)  
{  
}  
}
```

You would have to call the member constructors in the initialization list context

- We can initialize with a **1-step** process using a C++ **constructor initialization list**
 - Constructor(param_list) : member1(param/val), ..., memberN(param/val)
{ ... }
- We are really calling the respective constructors for each data member at the time memory is allocated

Summary

```
Student::Student()  
{  
    name = "Tommy Trojan";  
    id = 12313  
    scores.resize(10);  
}
```

You can still assign data members in the {...}

```
Student::Student() :  
    name(), id(), scores()  
    // calls to default constructors  
{  
    name = "Tommy Trojan";  
    id = 12313  
    scores.resize(10);  
}
```

But any member not in the initialization list will have its default constructor invoked before the {...}

- You can still assign values in the constructor but realize that the default constructors will have been called already
- So generally if you know what value you want to assign a data member it's good practice to do it in the initialization list

```
Student::Student() :  
    name("Tommy"), id(12313), scores(10)  
{ }
```

This would be the preferred approach especially for any non-scalar members (i.e. an object)

Exercise: [cpp/cs104/classes/constructor_init2](#)

What NOT to do!

- So we CANNOT call constructors on data members INSIDE the constructor)
 - So what can we do??? Use initialization lists!

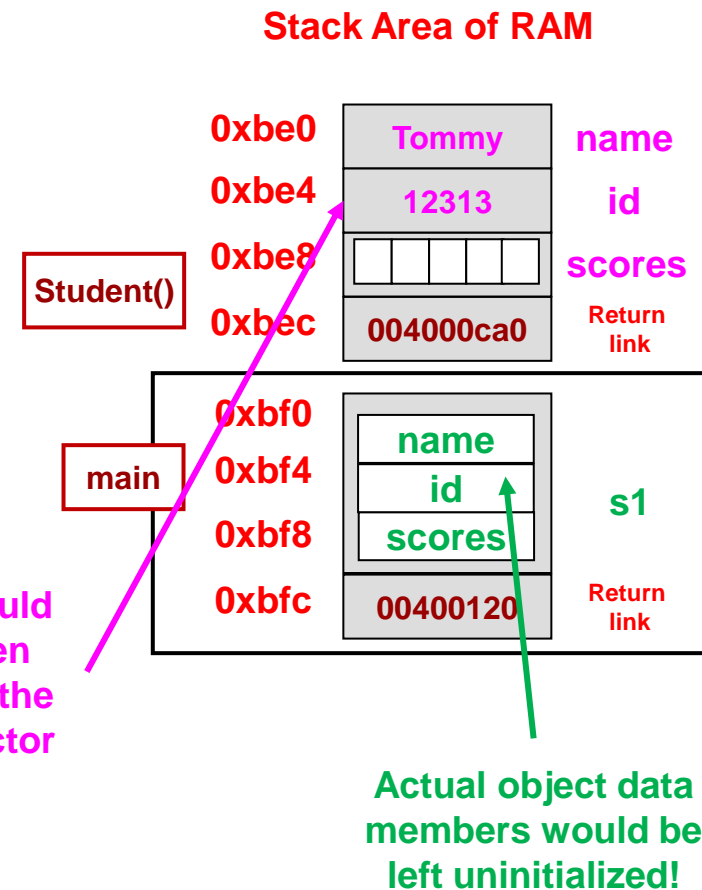
```
#include <string>
#include <vector>

struct Student
{ std::string name;
  int id;
  std::vector<double> scores;
  // say I want 10 test scores per student

  Student() /* mem allocated here */
  { // Can I do this to init. members?
    string name("Tommy"); // or
    // name("Tommy")
    int id = 12313;
    vector <double> scores(10);
  }
};

int main()
{ Student s1;
  //...
}
```

Local variables would be created but then immediately die at the end of the constructor



Calling Constructors

- You CANNOT use one constructor as a helper function to help initialize members
 - DON'T call one constructor from another constructor for your class

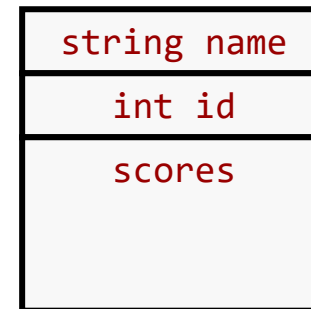
```
struct Student
{ std::string name;
  int id;
  std::vector<double> scores;

  Student() : name("Tommy"), id(-1), scores(10)
  {

  }

  Student(string n)
  { Student();
    name = n;
  }
};

int main()
{
  Student s1("Jane Doe");
  // more code...
}
```



Can we use `Student()` inside `Student(string name)` to initialize the data members to defaults and then just replace the name?

No!! Calling a constructor always allocates memory for another object. So rather than initializing the members of `s1`, we have created some new, anonymous `Student` object which will die at the end of the constructor

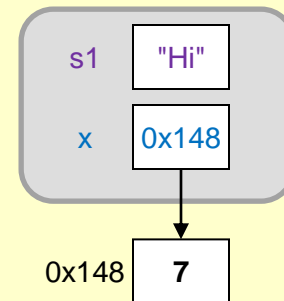
C++ Classes: Destructors

- Destructors are called when an object goes out of scope or is freed from the heap (by “delete”)
- Destructors
 - Can have **one** or **none** (if no destructor defined by the programmer, compiler will generate an empty destructor)
 - Have no return value
 - Have the name ~ClassName()
 - Data members of an object have their destructor's called automatically upon completion of the destructor.
- Why use a destructor?
 - Not necessary in simple cases
 - Clean up resources that won't go away automatically (e.g. when data members are pointing to dynamically allocated memory that should be deallocated when the object goes out of scope)
 - Destructors are only needed only if you need to do more than that (i.e. if you need to release resources, close files, deallocate what pointers are point to, etc.)
 - The destructor need only clean up resources that are referenced by data members.

```
class Item
{ string s1;
  int* x;
public:
  Item();
  ~Item();
};

Item::Item()
{ s1 = "Hi";
  x = new int;
  *x = 7;
}

Item::~~Item()
{
  delete x;
} // data members
// destructed here
```



OTHER IMPORTANT CLASS DETAILS

Member Functions

- Object member access uses dot (.) operator
- Pointer-to-object member access uses arrow (->) operator
- Member functions have access to all data members of a class
- **Use “const” keyword if it won't change member data**
 - This is good practice and you should starting doing it

```
class Item
{ int val;
  public:
    void foo();
    int bar() const;
};

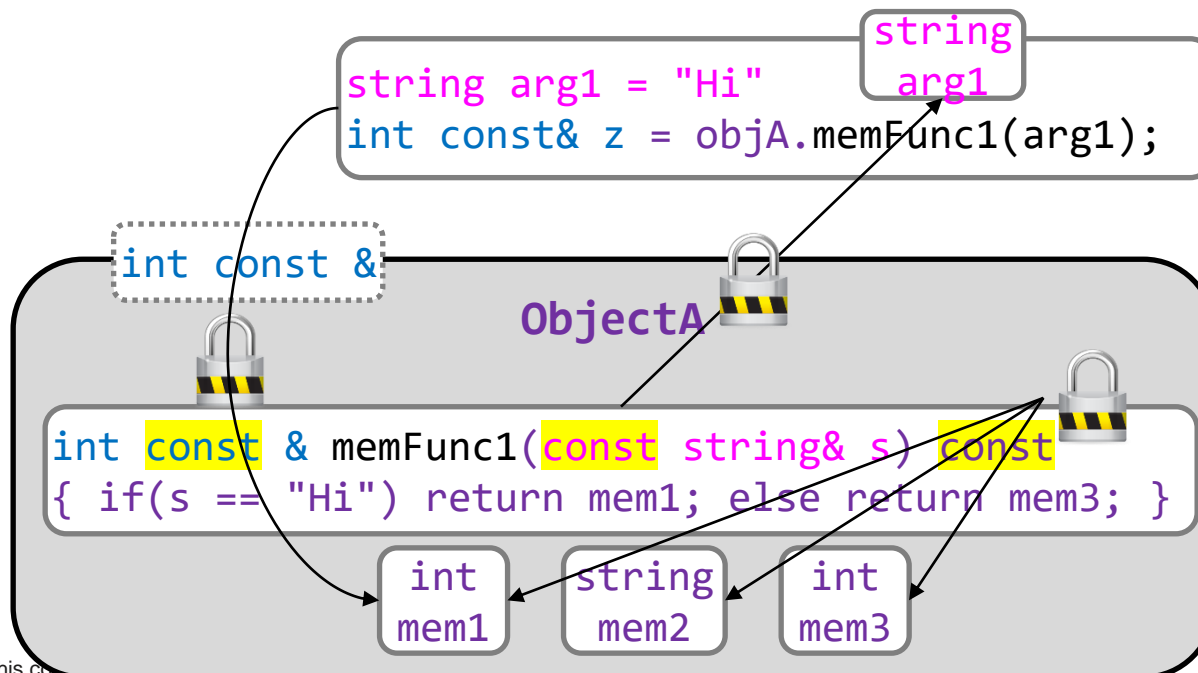
void Item::foo()
{ val = 5; }

int Item::bar() const
{ return val+1; }

int main()
{
    Item x;
    x.foo();
    Item *y = &x;
    (*y).bar();
    y->bar(); // equivalent
    return 0;
}
```

'const' Keyword

- const keyword can be used with
 1. Input arguments to ensure they aren't modified
 2. After a member function to ensure data members aren't modified by the function
 3. Return values to ensure they aren't modified



Exercises

- `cpp/cs104/classes/const_members`
- `cpp/cs104/classes/const_members2`
- `cpp/cs104/classes/const_return`

C++ Classes: Other Notes

- Classes are generally split across two files
 - ClassName.h – Contains interface description
 - ClassName.cpp – Contains implementation details
- Make sure you remember to **prevent multiple inclusion errors** with your header file by using **#ifndef**, **#define**, and **#endif**

```
#ifndef CLASSNAME_H
#define CLASSNAME_H
class ClassName { ... };
```

```
#endif
```

```
#ifndef STRING_H
#define STRING_H
class string{
    string();
    size_t length() const;
    /* ... */
};
#endif
```

string.h

```
#include "string.h"

string::string()
{ /* ... */ }

size_t string::length() const
{ /* ... */ }
```

string.cpp

SOLUTIONS

Review from CS 103 [1]

Const function arguments

- Will this code compile? **No, modification of x in f1()**
- Indicate what will be printed (assuming it compiles) – **b.size() will be 3**

```
void f1(const vector<int>& x){
    x.push_back(103);
    x.push_back(104);
}

void f2(string& y){
    y = "Bye";
}

int main()
{
    vector<int> a; string b = "Hi";
    f1(a);
    f2(b);
    cout << b.size() << endl;
    return 0;
}
```

Const member functions

- What does the highlighted const keyword imply in the code below?
 - **No data members can be modified nor non-const member functions called**

```
class Item
{ int val;
public:
    void foo();
    int bar() const;
};

void Item::foo()
{ val = 5; }

int Item::bar() const
{ return val+1; }
```

Review from CS 103 [2]

Constructor Initialization Lists

- What is the most efficient means to initialize the vals member to an initial array size of 20 and s to a user-defined argument?

```
class Thing {  
public:  
    Thing(const std::string& s_init);  
private:  
    vector<int> vals;  
    string s;  
};  
  
Thing::Thing(const std::string& s_init)  
    : vals(20)  
{  
  
}
```

Construction Order

- What is printed by the code below?
 - ABC
 - DEF
 - XYZ

```
class ABC {  
public:  
    ABC() { cout << "ABC" << endl; }  
};  
class DEF {  
public:  
    DEF() { cout << "DEF" << endl; }  
};  
class XYZ {  
    ABC m1; DEF m2;  
public:  
    XYZ() { cout << "XYZ" << endl; }  
};  
int main() {  
    XYZ x1;  
    return 0;  
}
```

Review from CS 103 [3]

Friend Functions

- What does the highlighted friend keyword imply in the code below?
 - That function can access Complex private members
- What would break if we remove it?
 - Could not access rhs.real / rhs.imag

```
class Complex
{
public:
    Complex();
    Complex(double r, double i);
    friend Complex operator+(const int&, const Complex&);
private:
    double real, imag;
};

Complex operator+(const int& lhs, const Complex &rhs)
{
    Complex temp;
    temp.real = lhs + rhs.real;    temp.imag = rhs.imag;
    return temp;
}
```

Friend Classes

- Can DEF::clear() access obj.x? Yes
- If not, how can class ABC grant access to DEF?
 - Add friend definition

```
class ABC {
    int x; // data member
public:
    friend class DEF;
    ...
};

class DEF {
public:
    void clear(DEF& obj) { obj.x = 0; }
};
```