CSCI 104
Classes

Mark Redekopp
David Kempe
Sandra Batista

OVERVIEW AND CONCEPTS

C Structs

- 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

```
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
  - Keep state hidden/separate from other programmers (or yourself) via private members
- Abstraction
  - Depend only on an interface!
    - Ex. a microwave…Do you know how it works? But can you use it?
  - Hide implementation details to create low degree of ***coupling*** between different components
- 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

```
struct Machine{
  Piece* pieces;
  Engine* engine;
};
int main()
{
  Machine m;

  init_subsystemA(&m);

  change_subsystemB(&m);

  replace_subsystemC(&m);

  m.start();
  // Seg. Fault!! Why?
}
```

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 card to your laptop
- OO Design seeks to reduce coupling as much as possible by
  - Creating well-defined interfaces to change (write) or access (read) the state of an object
  - Allow alternate implementations that may be more appropriate for different cases

PARTS OF A CLASS

Parts of a C++ Class

- What are the main parts of a class?
  - Member variables
    - What data must be stored?
  - 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 (for now)
  - 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
  - C++ won't automatically initialize member variables
  - No return value
- Default Constructor
  - Can have one or none in a class
  - Basic no-argument constructor
  - Has the name ClassName()
  - If class has no constructors, C++ will make a default
    - But it is just an empty constructor (e.g. Item::Item() { })
- Overloaded 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 IntLinkedList {
 public:
  IntLinkedList( );
  IntLinkedList(int n);
  ~IntLinkedList( );
  ...
};
```

Identify that Constructor

- Prototype what constructors are being called here

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

int main()
{
  string s1;
  string s2("abc");

  vector<int> dat(30);

  return 0;
}
```

Identify that Constructor

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

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

int main()
{
  string s1;
  string s2("abc");

  vector<int> dat(30);

  return 0;
}
```

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

};

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

Composite Objects

- **Fun Fact 1**: Memory for an object comes alive before '{' of the constructor code
- **Fun Fact 2**: Constructors for objects get called (and can *ONLY EVER* get called) at the time memory is allocated

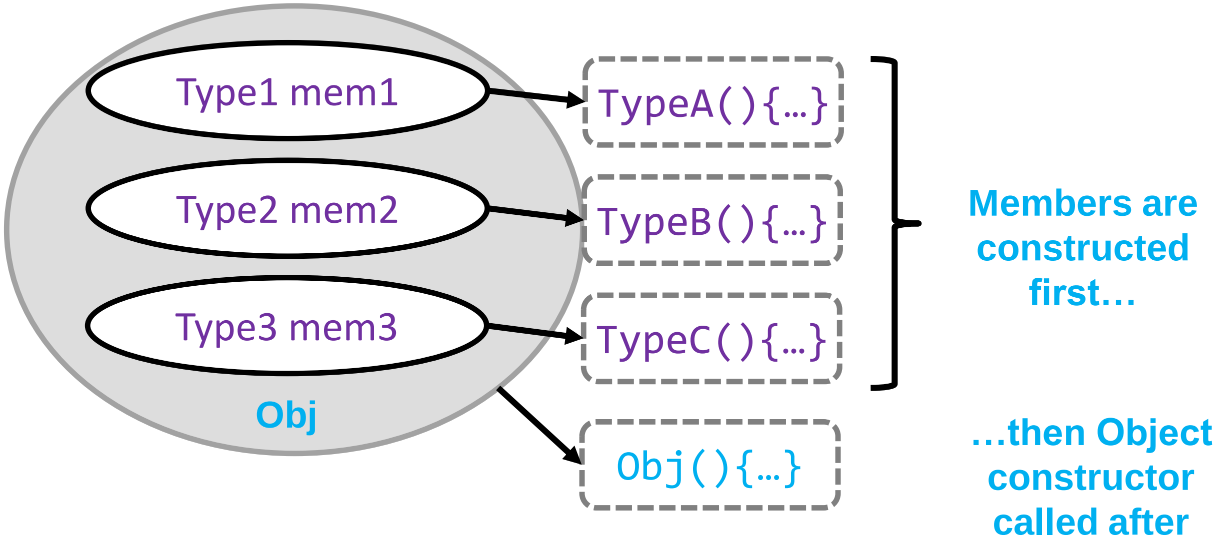
string name
int id
scores

```
#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 call string & vector
    // constructors to init. members?
    name("Tommy Trojan");
    id = 12313;
    scores(10);
  }
};

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

Initializing Members

- **To recap**: When an object is constructed the individual members are constructed first
  – Members constructors are called before object's constructor

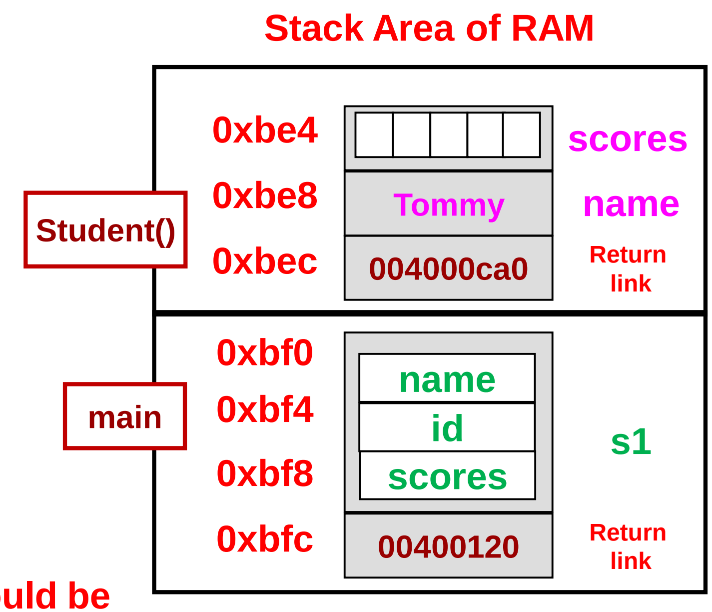
```
Class Obj
{ public:
    Obj();
    // public members
  private:
    Type1 mem1;
    Type2 mem2;
    Type3 mem3;
};
```



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")
    id = 12313;
    vector <double> scores(10);
  }
};
int main()
{ Student s1;
  //...
}
```



This would be "constructing" name twice. It's too late to do it in the {…}

Old Initialization Approach

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

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

If you write this…

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

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

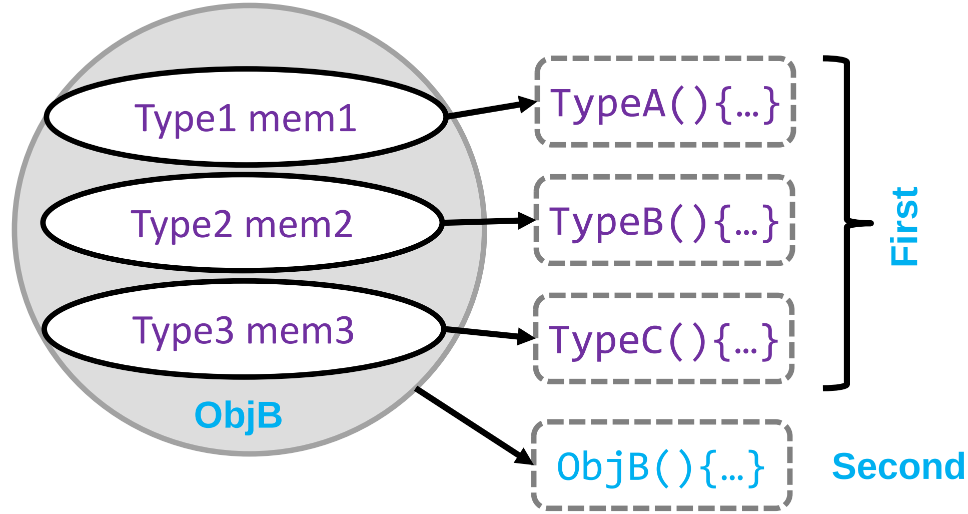
```
struct Student
{ std::string name;
  int id;
  std::vector<double> scores;

  Student()
  { name = "Tommy Trojan";  // default
    id = -1;     // default
    scores(10); // default 10 assignments
  }
  Student(string n)
  { Student();
    name = n;
  }
};

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

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

No!! Calling a constructor always allocates 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

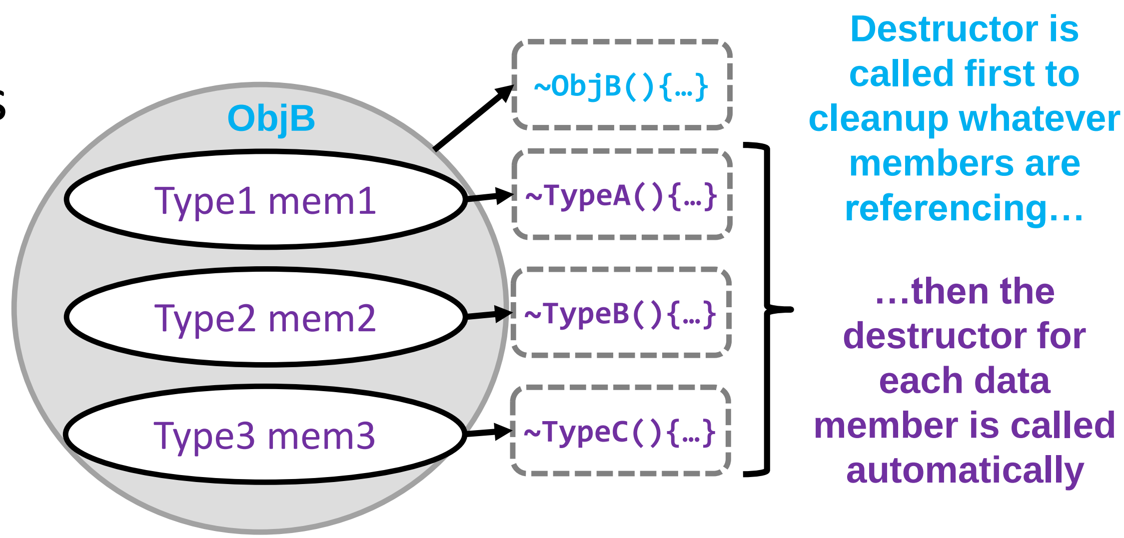
Allocating and Deallocating Members

- Members of an object have their constructor called automatically before the Object's constructor executes



Construction

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

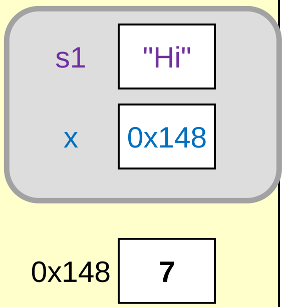


Destruction

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

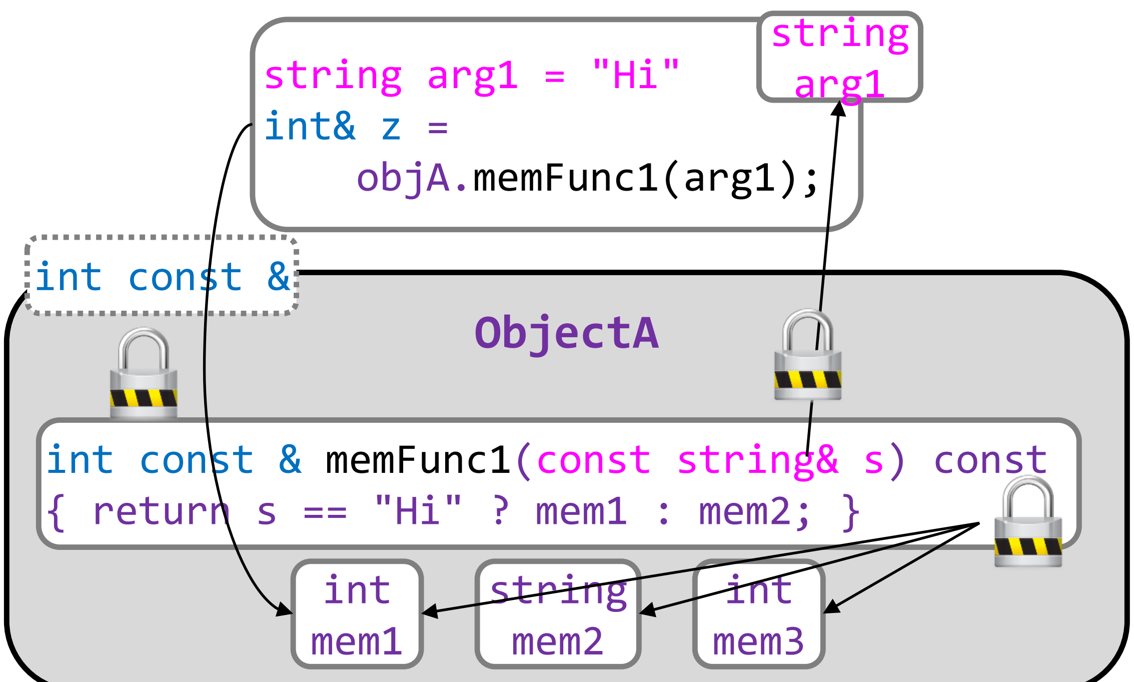
Member Functions

- 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**

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

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

void Item::bar() const
{ }

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

'const' Keyword

- `const` keyword can be used with
  - Input arguments to ensure they aren't modified
  - After a member function to ensure data members aren't modified by the function
  - 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

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

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 -DDEGUG 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
```