CS3157: Advanced Programming

Lecture #11
Apr 10
Shlomo Hershkop
shlomo@cs.columbia.edu

Outline

• CPP continued
• Language basics: identifiers, data types, operators, type conversions, branching and looping, program structure
• data structures: arrays, structures
• pointers and references
• I/O: writing to the screen, reading from the keyboard, iostream library
• functions: defining, overloading, inlining, overriding
• classes: defining, scope, ctors and dtors
• listing of keywords

• Reading
  – c++core ch 3-6
  – c++nutshell 5-6,9
Announcements

• This weekend (Thur/Friday) Passover begins
• I wont be available, please contact the Ta’s for any help
• Don’t forget the hw’s are due next week

Next 2 weeks

• We will be covering practical CPP
• For those taking data structures in cpp, this will be very very very very useful
• For those of you not taking this will be very very very very useful
• Still also fun!
Before we get started

• So we’ve tasted
  – Perl
  – C
  – CPP
  – Java (hopefully in the past)

Programming languages

• When you taking a formal course in programming languages
  – Programming Languages and Translators (PLT)
• Covers the limitations of a language through mathematical models

• But a practical question:
• You want to program something…..

• How do you choose a language??

How to choose

• Depends on project
• Depending on requirements
• Depending on available libraries
• Depending on skill availability
• Hardware constraints
• Operating constraints
CPP classes

- So we covered basic CPP with basic classes

- I really hope you did the lab already

Random important stuff

- I’m going to step through some random cpp stuff incase you’ve missed it
Pass by reference

• In C we noticed default function argument was pass by value

• How does C pass by reference?

CPP pass by reference

• Another way of passing by reference
  ```c
  int count = 10;
  int &rcount = count;
  ```
references

void foo2(int &);

void foo(int &refint){
    refint *= refint;
}

Variable scope

• CPP allows you to specify scope through unary scope operator (::)
• So can differentiate between local and global variables
code

int count = 10;

int main(){
    int count = 5;

    // count is local
    // ::count is global
    // std::count is the same as 2

Inline functions

• We covered these….any ideas?

• Where do you code them?
Functions organization

- You’ve programmed classes in Java
- What kind of functions exist with well designed classes

Functions

- Accessor
- Mutator
- Helper
- Predicate
CPP classes

- A class is a collection of functions and variables
- In CPP we have constructors and destructors

Order of running program

- In c we saw that the program always starts from main

- This is different in cpp
What can go wrong

• The good thing about cpp is that your program can now crash many times even before reaching main 😊

Ordering and where to look for problems

• Global variables
  – Assignments and constructors
  – What else ??
• Main
• Local variables
• End local variables
• End main
• Global destructors
code

- I’d like to cover a bunch of code examples now illustrating the power of classes

- Will start from simple array and work out a complex class

- You will do the same with the string class in this week’s lab

Abstraction with member functions

- example #1: array1.cpp
- example #2: array2.cpp
  - array1.cpp with interface functions
- example #3: array3.cpp
  - array2.cpp with member functions

- class definition
- public vs private
- declaring member functions inside/outside class definition
- scope operator (::)
- this pointer
array1.cpp

```c
struct IntArray {
    int *elems;
    size_t numElems;
};

main() {
    IntArray powersOf2 = { 0, 0 }; 
    powersOf2.numElems = 8;
    powersOf2.elems = (int *)malloc( powersOf2.numElems * 
    sizeof( int ));
    powersOf2.elems[0] = 1;
    for ( int i=1; i<powersOf2.numElems; i++) {
        powersOf2.elems[i] = 2 * powersOf2.elems[i-1];
    }
    cout << "here are the elements:\n";
    for ( int i=0; i<powersOf2.numElems; i++) {
        cout << "i= " << i << " powerOf2=" << 
        powersOf2.elems[i] << "\n";
    }
    free( powersOf2.elems );
}
```

array2

```c
void IA_init( IntArray *object ) { 
    object->numElems = 0;
    object->elems = 0;
} // end of IA_init()

void IA_cleanup( IntArray *object ) { 
    free( object->elems );
    object->numElems = 0;
} // end of IA_cleanup()

void IA_setSize( IntArray *object, size_t value ) { 
    if ( object->elems != 0 ) { 
        free( object->elems );
    } 
    object->numElems = value;
    object->elems = (int *)malloc( value * sizeof( int ));
} // end of IA_setSize()

size_t IA_getSize( IntArray *object ) { 
    return( object->numElems );
} // end of IA_getSize()
```
Class friends

- allows two or more classes to share private members
- e.g., container and iterator classes
- friendship is not transitive

hierarchy

- composition:
  - creating objects with other objects as members
  - example: array4.cpp
- derivation:
  - defining classes by expanding other classes
  - like “extends” in java
  - example:
    ```
    class SortIntArray : public IntArray {
    public:
      void sort();
    private:
      int *sortBuf;
    }; // end of class SortIntArray
    ```
- “base class” (IntArray) and “derived class” (SortIntArray)
- derived class can only access public members of base class
• complete example: array5.cpp
  – public vs private derivation:

• public derivation means that users of the derived class can access the public portions of the base class

• private derivation means that all of the base class is inaccessible to anything outside the derived class

• private is the default

Class derivation

• encapsulation
  – derivation maintains encapsulation
  – i.e., it is better to expand IntArray and add sort() than to modify your own version of IntArray

• friendship
  – not the same as derivation!!
  – example:

• is a friend of
• B2 is a friend of B1
• D1 is derived from B1
• D2 is derived from B2
• B2 has special access to private members of B1 as a friend
• But D2 does not inherit this special access
• nor does B2 get special access to D1 (derived from friend B1)
Derivation and pointer conversion

- derived-class instance is treated like a base-class instance
- but you can’t go the other way
- example:

```c
main() {
    IntArray ia, *pia;
    // base-class object and pointer
    StatsIntArray sia, *psia;
    // derived-class object and pointer
    pia = &sia; // okay: base pointer -> derived object
    psia = pia; // no: derived pointer -> base pointer
    psia = (StatsIntArray *)pia; // sort of okay now since:
    // 1. there’s a cast
    // 2. pia is really pointing to sia,
    //    but if it were pointing to ia, then
    //    this wouldn’t work (as below)
    psia = (StatsIntArray *)&ia; // no: because ia isn’t a StatsIntArray
}
```

- danger:
  - don’t point a base class pointer to an array of derived objects!
  - they aren’t the same size!
Const variables

• Can have const variables in a class

• Any ideas for this?

Operator overloading

• Most operators can be overloaded in cpp
• Treated as functions
• But it's important to understand how they really work
Operators which can’t be overloaded

- .
- .*
- ::
- ?:  
- sizeof
- \( X = X + Y \)
- Need to overload
  
  +

  =

- But this doesn’t overload \(+=\)

- Functions can be member or non-member
- Non-member as friends
- If its member, can use this
- ( ), [ ], -> or any assignments must be class members

- When overloading need to follow set function signature
• Code from fig18_03 (c book)

• Will cover next class in depth

unary

• Y += Z
• Y.operator+=( Z )

• ++D
• member
  – D.operator++()
• Non member
  – operator++(D)
For lab

• Read up on classes, and class overloading

• Will be easier lab since homework will be due

• Next week lab, you will be presenting your Othello program to the class
  – You need to show up to lab (if possible)
  – Else someone needs to present it for you
  – Will vote for best homework
  – Some kind of prize