Outline

• More CPP
  – class abstractions
  – class inheritance
  – examples
  – templates

• Reading:
  – chapters 18, 19, 20
Announcements

• wrapping up cpp today

• hope you started homework 2

• last lab released today

So Far

• we’ve covered basic classes

• creating simple classes

• working with some simple members

• lets take it to the next step
Task

- would like to create a String class

- would like to say
  ```cpp
  int main (void) {
    String s1 = String("first");
    String s2 = String("second");
    String s3;
    s3 = s1 + s2;
    cout << s3;
    return 0;
  }
  ```

Operator overloading

- Most operators can be overloaded in cpp
- Treated as functions
  - important to understand how arguments are organized
  - need to know how to code them
  - how to code them efficiently
    - example define them in terms of each other
      - += could call +
reminder

- s3 = s1 + s2;
- Need to overload
  +
  =
- But this doesn’t overload +=
• Functions can be member or non-member, your choice!
  – Non-member as friends if need private data
• If its member, can use the this pointer

• Exception: operators (), [], -> or any assignments must be class members

• When overloading need to follow set function signature

\[
\text{cout}
\]

• \text{cout} \ll \text{yourclass}

• left operand is ostream &
• so non member functions (belongs to ostream)
• friend if you would like

• lets code something
String class

- let's define a simple string class
- put output in its const and dest so we can follow
- constructor should take `const char *`
- would like to have following defined:
  - `int length();`
  - `int hash();`
- any ideas on how to do it?

overload printing

```cpp
friend ostream & operator <<(ostream &, const String &); 

ostream &operator<<(ostream &output, String &str) {
    output << "'" << ??? << "'";
    return output;
}
```
note

• when you call:
cout << s1 << s2;
• it is first:
operator<<(cout,s1)
• and then
operator<<(cout,s2)

Next

• want to overload the unary operator !
• test if a string is blank
  • int operator!() const;
  • or
  • friend int operator(const String &);

• !s1
  • s.operator!() or operator!(s)
same idea

• const String operator+=(const String &)

• vs

• friend const String &operator+=(String &, const String &)

• what will s1 += s2 produce?

Array Class

• Arrays are hard to work with directly since there is no support for out of bounds

• lets look at 18.4 from the book
extending

• any ideas on how to extend the base class ??

• so how can we tell the difference between ++s1 and s1++
signatures

- s1++
- s1.operator++(0)
- operator++(s1,0)

- ++s1;
- s1.operator++()
- operator++(s1)
reuse

• one of the powers to OOP is the idea of reuseability

• if I spend 5 billion hours working on my code, I probably want to get some use out of it outside of the specific task
  – design issues
  – extension issues

inheritance

• idea: allow a new class to inherit data members and functions from a base class

• can add members and functions

• represents a more specific idea

• vehicle -> minivan
• you can access protected members of parent

• can not access private members of parent  
  – can still use public accessors and modifiers

code

class IntArray: public Array {

  • simplest type of inheritance
  • private members not inherited
  • public/protected inherited accordingly
code
• create a point class
  – setPoint
  – <<
• derive Square
  – getArea()
  – <<

overriding
• we can redefine a base class function in the derived class and have c++ call the correct one
Question

- can
- Point *pp1;
- Square *sp1;

- given
- Point p = Point(3,4);
- Square s = Square(..

- can we say:
  - pp1 = s  ??????
  - sp1 = p  ??????

private inheritance

- we have used public inheritance

- private inheritance makes everyone from the base class come in as private members of the derived class
base class constructors

• need to launch base class constructor in derived class if you don’t want the default to be called

• destructors are reversed

• lets see this in action

is a vs has a

• one important design decision is to know when to derive and when to use member variable
issue

• one issue with overriding, is that if the derived class doesn’t provide a function, we will use the base class definition

• this doesn’t always make sense

• Example I want a function MPG for any type of vehicle, but doesn’t make sense of base class

virtual functions

• solution :

• declare the function to be virtual

• virtual double MPG();

• allow you to use a base class pointer to call at runtime the correct function (polymorphism)
abstract class

• sometimes its even useful to have a base class which can’t be instantiated
• if any virtual function is declared pure virtual:
  • virtual int MPG() = 0;

note

• constructors can not be virtual

• need virtual destructors to make everything work if you are going to have destructors in any of your classes (do it anyway)
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
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:
";
for ( int i=0; i<powersOf2.numElems; i++ ) {
    cout << "i=" << i << " powerOf2=" << powersOf2.elems[i] << "\n";
}
free( powersOf2.elems );
}

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()
hierarchy

- composition:
  - creating objects with other objects as members
  - example: array4.cpp

- derivation:
  - defining classes by expanding other classes
  - like "extends" in java
  - example:
    ```cpp
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
  ```
switching gears

- recursive programming

- any one know how to recursively solve a problem?

- different types of recursions

- left tail recursion

- non tail recursion
Templates

template<typename X>
void foo(X &first, X second){
    first += second;
}

STL

- standard template library
- tons of useful stuff here

- if you do any serious programming you should consider STL
  - they’ve worked out all the bugs 😊
  - very efficient
  - make sure you understand what you are doing
Lab

• let's get started on the next lab now