



CS216: Program and Data Representation

- These slides adopted from slides from the "old" version of CS201.
- Topics include:
 - Pointers and Dynamic Memory
 - Dynamic classes
 - The Gang of Three
 - Class and SW Design

Overview

- Pointers in general
- SW design issues
 - Application design with pointers.
Functions: responsibility for handling problems
- Dynamic memory
 - new, delete, delete []
- Classes with dynamic memory members
 - Value semantics matter!
Copy constructor, assignment operator, destructor
- Example: Bag class, dynamic arrays

Review Pointers

- Problems to avoid!
 - Can you name them?

Review Pointers

- Problems to avoid!
 - A pointer that is uninitialized
 - Don't dereference it!
 - Make sure it points to allocated memory
 - An existing named variable
 - Something allocated with new
 - A pointer that points to some memory that is uninitialized
 - Pointers that point to memory that has been deallocated
 - Called a memory leak

Fun with claymation

- Binky's fun with pointers video:
<http://cslibrary.stanford.edu/104/>
 - Choose C++ version

Using new

- new is an operator
 - it's a defined C++ identifier, carries out an operation
 - allocates memory, returns an address
 - store address in a pointer
 - Whatever was pointed to before is possibly unreferenced now
- Memory allocated on the "heap"
 - "normal" variable allocated on the "stack"
- Can new fail?

new, failure

- new **can** fail – no more memory
- How does C++ tell us?
 - Throws an exception
 - We'll talk about exceptions later
- For now, this causes the program to halt
- What else could new do to inform us?
 - Could return a special value
 - Null pointer: address of zero
 - The C language works this way

Functions and Errors

- What new does on failure is an example of a general problem:
- *If a function (or component) detects a problem, what should its responsibility be in handling this?*
- There are many possible design decisions a development can make for this!

Choices in Handling Errors

- Abort the program
 - assert() or exit()
 - Print a new message (but where, if GUI?)
- Pass responsibility "up" to the caller
 - Special return value (e.g. a NULL pointer etc)
 - A error code parameter
 - Throw a C++ exception (later)

Error Handling is a Design Issue

- This is a design question!
 - Some functions are low-level, highly-focused components that do one specific task
 - Other functions are larger, with more management and coordination responsibility
- Should the "worker-bee" function make the decision to halt the program?
 - Perhaps. (When?) But usually not.
 - Perhaps the caller can recover.
- A function's documentation should say what errors might be detected **and** what it does

Our example here: a bag class

- What's a bag?
 - Like a set but can include duplicate items
- This is an ADT (abstract data type)
 - A model of information with rules
 - But many ways of implementing it with data structures
- We'll assume
 - Bag has a **capacity**: max it can hold
 - Bag has a current **size**: how many are in it

new and Pointers to Objects

- Consider an example bag class
 - Two constructors: default and with an initial capacity
- ```
bag *bagPtr1 = new bag; // default constructor
bag *bagPtr2 = new bag(10); // call other constructor
bag *bagPtr3 = new bag[10]; // array, default constr
```
- Note the first two create one object. The 3<sup>rd</sup> creates 10 objects

## Dereferencing Pointers to Objects

- Dereferencing pointers to objects  
`bag *bagPtr1 = new bag;`  
then `*bagPtr1` is a bag object.
- Accessing its member functions? Why not:  
`*bagPtr1.size()`
- Nope! Two operators here: `*` and `.`  
Which has higher precedence? The `.` operator  
But we want to dereference first, then access  
member in side the object.
- So, must use either: `(*bagPtr1).size()`  
`bagPtr1->size()`  
This latter syntax is used more often. (Follow the  
pointer.)

## Arrays of Objects

- Let's do:  
`bag *bagPtr3 = new bag[10]; // array, default constr`
- Each element in the array is one bag  
object. Example:  
`bagPtr3[2] = "hello"; // if this is legal`  
`int i = bagPtr3[2].size() /* 3rd bag's size */`

## Classes Defined Using Dynamic Memory

- Often we want a class to have a pointer as  
a data member
  - Uses dynamic memory
  - Flexible "parts" to each object
- When does the dynamic object get  
created?
  - Constructor, perhaps
  - Dynamically comes and goes as member  
functions are called

## Classes Defined Using Dynamic Memory

- But this complicates things for us:
  - When the object is destroyed, we must delete or  
`delete[]` the allocated memory
  - How? Define a *destructor* member function
  - Otherwise, a *memory leak* will occur as your program  
runs
- Also, value semantics now matter
  - We must define our own versions of the copy  
constructor and `operator=` to do a "deep copy"  
instead of a "shallow copy"
  - i.e. make a copy of the data pointed to, not the  
pointer

## "Gang of Three"

- In C++ there's a standard "prescription"  
for creating a dynamic class
- If you use dynamic memory, implement  
your own
  - copy constructor
  - assignment operator (`operator=`) as a friend  
function
  - destructor
- Otherwise.... (doom!)

## bag class example

- Private data members:  
`value_type *data;`  
`size_type used; // how many now`  
`size_type capacity; // we could have this many`
- Also in header file, in class statement:  
`typedef int value_type;`  
`typedef std::size_t size_type;`  
`static const size_type DEFAULT_CAPACITY = 30;`  
`bag(size_type initial_capacity = DEFAULT_CAPACITY);`  
`bag(const bag& source);`  
`~bag();`

### dynamic bag class: constructors

```
bag::bag(size_type initial_capacity) { // main constructor
 data = new value_type[initial_capacity];
 capacity = initial_capacity;
 used = 0;
}
bag::bag(const bag& source) { // copy constructor
 data = new value_type[source.capacity];
 capacity = source.capacity;
 used = source.used;
 copy(source.data, source.data + used, data); // see p.111
}
```

### An aside: constructors and member initialization

- C++ has an additional feature for constructors
    - Very often a constructor initializes a data member using something pass as a parameter
- ```
PlayList::PlayList(const string& n, int num) {
    name = n; size = num; }
```
- Alternative syntax:

```
PlayList::PlayList(const string& n, int num)
    : name(n), size(num) { /* anything else to do? */ }
```
 - Syntax is colon, then list of member(initial-value) pairs

Why use member initialization?

- Sometimes you have to! (Later...)
 - This is more efficient if members are objects
 - What's happening here?
- ```
PlayList::PlayList(const string& n, int num) {
 name = n; size = num; }
```
- The data member name is created first, with the default constructor
  - Then, operator= changes name
  - With member initialization lists, the copy constructor is called (so just one function call)

### dynamic bag class: destructor

```
bag::~bag() {
 delete [] data;
}
```

- When is this called?

### Class operators: members or not?

- They can be member functions or not?
  - Which? Why?
  - Note that non-members can be friends if needed
- Some general rules for operators and other functions:
  - operator>> and operator<< are never members
    - Must return the stream to support "chaining"
    - Probably friend functions
  - If a binary operator might support type conversion, make it a non-member
    - This idea is perhaps more advanced than where we are!
    - Example: playList1+playList2 and playList1+medRec1
    - How would we support both of these?
  - Everything else should be a member function
    - Especially operator= and other compound operators

### Output Operator

- We can overload the insertion operator:

```
ostream& operator<< (ostream &outs, const Point &p)
{
 // right here, print however you want to stream outs
 return outs;
}
```
- Usage with a hypothetical Point class:

```
Point x, y; ... cout << x; cout << x << y << endl;
```
- Note that cout << x << y is really:

```
((cout << x) << y);
```

just like x+y+z is ((x+y)+z)
- Reference return value: allows "chaining" here

## operator= for Dynamic Classes

- operator= is defined according to some standard rules. Always follow these!
  - **Step 1:** Check for self-assignment: `x=x;` is legal (but meaningless). Do nothing!
  - C++ keyword `this` is the address of the current object
    - Only meaningful inside a member function! Why?
    - BTW: `*this` is the current object itself
  - Remember that `x=y;` “clobbers” old data in `x`
  - **Step 2:** If you need more storage than currently in the current object, allocate it
    - May need to delete the old data in current object

## operator= for Dynamic Classes

- (Continued) operator= is defined according to some standard rules:
  - **Step 3:** Make current object a copy of the source object (passed as parameter)
    - Copy **all** data members (deep copy if needed)
  - **Step 4:**
    - return `*this` and make return value a reference to an object  
`bag& bag::operator= (const bag& source);`
    - Why? Again, “chaining”: `b1 = b2 = b3;` which associates like this: `b1 = (b2 = b3);` which really is this:  
`b1.operator=(b2.operator=(b3));`

## dynamic bag class: operator=

```
bag& bag::operator=(const bag& source) {
 value_type *new_data;
 if (this == &source) // self-assignment?
 return *this;
 if (capacity != source.capacity) { // Need more room?
 new_data = new value_type[source.capacity];
 delete [] data;
 data = new_data;
 capacity = source.capacity;
 }
 // Copy the data from the source array:
 used = source.used;
 copy(source.data, source.data + used, data);
 return *this;
}
```

## Program defensively: class invariant

- Order you do things is important here!
- We could get more room by:
  - Delete old data by calling `delete[]` on pointer
  - Then, calling `new` and assigning new address to pointer
- What if new fails?
  - Old data gone, exception thrown
  - Your class is left in an in-between state
  - Doesn't meet *class invariant* (see next slide)
  - Exception causes a return to the caller immediately
    - Your object is left in a state where the pointer does not point to an array that holds the items (it's a dangling pointer)
    - Make sure the class invariant is true at the time when you allocate memory

## Invariants

- **An invariant** – a property of something that we know will be true
  - Defining invariants is an important strategy for creating and knowing part of a program is correct
- Loop invariant
  - One or more conditions that we know are true at (say) when the loop-condition is tested
- Class invariant
  - Rules about the state (or values) of the member functions that are true at all times for a given object
  - Example from page 105 for the bag class
    - The variable `used` stores the number of items in the bag
    - Items are stored in the array from index 0 through `used-1`

## Summary

- Dynamic memory, dynamic arrays
  - Constructors and destructors must use `new` and `delete` properly
  - Functions that might require more capacity must check. A `reserve()` function helps us.
  - Value semantics. Shallow copying won't work.
    - Define a copy constructor and `operator=`
- Remember: each call to `new` must eventually be followed by a call to `delete`

### Summary (cont'd)

- New C++ features
  - Constructor's member initialization lists
  - Copy constructors
  - Standard steps for operator=
  - The keyword *this* inside a member function
  - Returning a reference to an object

### Summary (cont'd)

- Invariants and creating quality code
  - Define class invariants and make sure they're true even if a function fails or exits early
  - Value semantics for your class
  - Document both of these!
- Pointer issues in C++
  - Don't dereference a pointer that's not pointing to something valid
  - Make sure the memory referenced has valid data in it
  - Memory leaks: losing all references to memory allocated with `new`
  - Dangling pointers: deleting memory that some other pointer still considers a valid reference