CSI33 Data Structures

Department of Mathematics and Computer Science
Bronx Community College

November 6, 2017

Outline

Outline

1. Chapter 10: C++ Dynamic Memory
   - Dynamic Memory Classes
   - Dynamic Memory Errors

Proper Memory Management

Classes which allocate memory must manage it properly. Default behavior of C++ is insufficient:

- The assignment operation will not perform a "deep copy" of objects using references (pointers).
- Rather, the pointers themselves will be copied, leading to shared memory.
- Shared memory requires reference counting to be properly deallocated. This is not built into C++, and is hard to program.
- C++ objects will not even deallocate memory they have allocated themselves unless made to do so by the programmer.

Proper Memory Management

- A class must deallocate memory it allocates. Each object, when its lifetime is over, should free any memory it has used.
- A class must copy, not share, **referenced** object data when an existing object is being assigned a value from another object.
- A class must copy, not share, **referenced** object data when a new object is being created using the value of another.

Destructor

- The destructor for a class is a special member function written to perform any cleanup work at the end of an object's lifetime. If no such special work is necessary, a destructor need not be provided.
- A destructor is required for dynamic memory classes to prevent a memory leak (an object must deallocate any memory it has allocated when it has done its work.
- The destructor's name is tilde (˜) followed by the class name.
- The destructor takes no parameters.
- The destructor is called automatically when an object goes out of scope.
- The destructor is called automatically when the `delete` operator is called for a pointer to an object in that class.

Copy Constructor

- When a constructor for a class is called with an existing object of the class as an actual parameter, the data members are copied into the memory allocated for the new object. In C++, this is the behavior of the default copy constructor. What will not happen automatically is a deep copy of the memory pointed to by data members which are pointers. To force a deep copy, the copy constructor must be explicitly defined for the class.

Assignment Operator

- When an existing object of a class is assigned a value which is an existing object of the class, the data members are copied into the memory of the assigned-to object. In C++, this is the behavior of the default assignment operator. What will not happen automatically is a deep copy of the memory pointed to by data members which are pointers. To force a deep copy, the assignment operator, `operator=`, must be explicitly defined for the class.

Example Dynamic Array Class: List

```
class List {
public:
   List(size_t capacity=10); // constructor
   List(const List &a); // copy constructor
   ~List(); // destructor
   int& operator[](size_t pos); // bracket operator
   List& operator=(const List &a); // assignment
   List& operator+=(const List &a); // += operator
   void append(int item);
   size_t size() const { return size_; }
private:
   void copy(const List &a);
   void resize(size_t new_size); //larger array
   int *data_; // dynamic array
   size_t size_; // size of dynamic array
   size_t capacity_; }; // capacity of dynamic array
```

Example Dynamic Array Class: List

```
List::List(size_t capacity)
{
  data_ = new int[capacity];
  capacity_ = capacity;
  size_ = 0;
}
```

Example Dynamic Array Class: List

```
List::~List() {
  delete [] data_;
}
```

Example Dynamic Array Class: List

```
List::List(const List &list)
{
   copy(list);
}
void List::copy(const List &list) {
   size_t i;
   size_ = list.size_;
   capacity_ = list.capacity_;
   data_ = new int[list.capacity_];
   for (i=0; i<list.capacity_; ++i) {
      data_[i] = list.data_[i];
   }
}
```

Example Dynamic Array Class: List

```
List& List::operator=(const List &list)
{
  if (&list != this) {
    // deallocate existing dynamic array
    delete [] data_;
    // copy the data
    copy(list);
  }
  return *this;
}
```

Reference Return Types

Reference return types are allowed only when a function returns the address of data that will still be there after the function call is finished. (This disallows returning a reference to a local variable.) You can return a reference to a data member (attribute) of an object in the class to which the member function belongs. In the `List` class, this is done in the implementation of the `[]` operator to allow indexed data to be on the left side of an assignment statement, as in

```
List l(3);
l[0] = 0;
l[1] = 1;
l[2] = 2;
```

Reference Return Types

```
inline int& List::operator[](size_t pos)
{
  return data_[pos];
}
```

Memory Leaks

When memory that has been allocated is not deallocated in some scenario, then each time the scenario occurs another chunk of memory becomes unavailable to the system which is trying to reuse memory. This can happen thousands or millions of times. As it gets harder to find available memory when it is needed, the operating system tries to use the disk drive to keep memory it is using (this is called paging or swapping). This slows the system down, since disk access is much slower than semiconductor memory. Eventually the system crashes.

Memory leaks are hard to find and fix, which is why they exist in commercially sold software. Rebooting your PC every once in a while to refresh memory can work around this problem.

Memory Leaks

```
// This code is incorrect
{
  int *x;
  x = new int;
  *x = 3;
  x = new int;
  *x = 4;
  delete x;
}
```

Accessing Invalid Memory

Accessing invalid memory means reading or writing on memory which has been deallocated (and reallocated for a different task). This leads to unpredictable and incorrect behavior in a program. Reading such data will produce garbage since the memory has been reallocated and reused by a different task running at the same time. Writing on such memory will cause the task which now uses it to crash, since its contents have been changed.

Accessing Invalid Memory

```
int main() // This program is incorrect
{
   int *y = new int;
   delete y;
   *y = 3;
   return 0;
}
```