# CPSC 5011: Object-Oriented Concepts

## Lecture 5: Memory

### Memory Management, Shallow & Deep Copying, and Garbage Collection

# Standard Views of Memory

- Memory management is a background process
  - handled by Operating System (OS)
- Abstraction frees the programmer from
  - tedious, low-level platform-dependent details
  - managing specific memory locations
- Software designer's perspective
- Memory is uniform: no distinction is made between
  - the cache and secondary store
- Memory is unlimited resource
  - virtual memory

# Program Memory

- Code segment
- Data segment:  Run-time stack – static allocation
  - Holds activation records associated with function calls
    - Program counter
    - Local data (passed parameters, local declarations)
  - main() is a function (the 'first')
  - Efficient!! (but rigid)
    - Register aids in handling stack frames (activation records)
    - Stack frames fixed size (with possible variable portion)
- Data segment: Heap – dynamic allocation
  - Memory for run-time allocations
  - Less efficient, more flexible

# Table 4.1 Common Difficulties with Program Memory

| *Memory Problem* | *Cause* | *Consequence* | *Effect* |
|---|---|---|---|
| Data corruption | Hidden aliases | Ownership undermined | Data values overwritten |
| Performance Degradation | Fragmented heap | Allocator: more time to find free memory | Software slowed<br>Poor Scalability |
| Memory Leak | Heap memory not collected | Heap memory unusable | Lost resource |
| C++ Memory Leak | Handle to memory lost | Memory inaccessible | Memory allocated but unusable |

# Heap Memory

- Flexible memory allocation
  - Size and frequency of allocation may vary from run to run
  - Supports allocation dependent on environmental factors
- Run-time overhead (=> performance hit)
  - Explicit allocation
    - may slow down if the heap is fragmented
    - May be rejected if free blocks of heap memory too small
  - Explicit deallocation
    - C++ model: programmer responsible for releasing memory
  - Implicit deallocation
    - C#/Java:  execution must be suspended if the garbage collector runs

# C++: Explicit deallocation

- C++ designed to be backward compatible with C
- Emphases on efficiency and control retained
- Explicit deallocation
  - means of retaining control of heap management
  - efficient amortization of the cost of memory deallocation
- Without implicit deallocation, C++ programs
  - not subject to suspension
    - for a run of the garbage collector
  - suitable for real-time systems
- BUT, memory leaks due to software design

# Example 4.1 C++ Allocation of (Heap) Memory at Run-Time

```
// "ptr" is a pointer variable allocated on the stack
// "ptr" holds the address of the heap object return by new
MyType*  ptr = new MyType;    // #A: MyType object allocated

// deallocate heap memory via call to delete operator
delete ptr;                   // #B: MyType object deallocated

// null out pointer to indicate it 'points to nothing'
ptr = 0;              // #C: programmer must reset pointer

// pointers can also hold the address of an array
ptr = new MyType[10]; // #D: 10 MyType objects allocated
…
// must use delete[] when deallocating an array on heap
delete[]       ptr;   // #E: 10 MyType objects deallocated
```

# Example 4.2   C++ primitives: allocation/deallocation clear

```
// function code: primitives used, no class objects

// application programmer ERROR:  NEW not matched with DELETE
void leakMemory() {
        int*    heapData;
        heapData = new int[100];
        …
        return;
}       // memory leak obvious: no explicit deallocation
```

# Example 4.2 continued

```
// function code ok:  NEW matched with DELETE
void noMemoryLeak() {
        int*    heapData;
        heapData = new int[100];       …
        // heap memory explicitly deallocated: delete matches new
        delete[] heapData;
        return;
}

// function code ok:  access to heap memory passed back to caller
// CALLER MUST ASSUME RESPONSIBILITY (ownership) FOR HEAP MEMORY
int* passMemory() {
        int*    heapData;
        heapData = new int[100];       …
        return  heapData;
}
```

# C#/Java: Implicit deallocation

- Java was designed 10 years after C++
  - memory errors in C++ code widely known
    - premature deallocations, data corruption, leaks
  - Java designers decided to
    - remove memory management responsibilities from programmer
    - rely on garbage collection for reclaiming heap memory
  - C# ("Microsoft's Java" )followed suit
- BUT, data corruption and memory leaks still occur
  - garbage collection is not a perfect process.
  - C#/Java programmer should zero out references
- Software designers should track memory

# Tracking ownership

- Who owns this data? Who deallocates it?
- Why is tracking data ownership difficult?
- Multiple handles to the same data
  - Explicit aliases and call by reference
  - difficult to track all handles, especially if scope differs.
- Class construct
  - hides data
  - not obvious to client when heap memory is allocated
  - Constructors can assume memory ownership
  - Other class methods can transfer memory ownership.

# Example 4.3 C++: Why Memory Leaks?

```
// function code looks correct
void whyLeakMemory1() {
        hiddenLeak*    naive;
        naive = new hiddenLeak[100];
        …
        delete[] naive;  // delete[] matches new[]
}

void whyLeakMemory2(hiddenLeak  localVar) {  …  }

void whyLeakMemory3() {
        hiddenLeak     steal;
        hiddenLeak     share;
        …
        steal = share;
        return;
}
```

# Example 4.4 C++ class without proper memory management

```
// IMPROPERLY DESIGNED: heap memory allocated (constructor)
// MISSING: destructor, copy constructor, overloaded =
class hiddenLeak {
   private:
       int*   heapData;
       int    size;
   public:
       hiddenLeak(unsigned s = 100) {
              size = s;
              heapData = new int[size];
       }
       …
};
```

# Example 4.5 C++ class with proper memory management

```
// class definition, .h file, memory managed properly
class noLeak {
   private:
        int*    heapData;
        int     size;
   public:
        noLeak(unsigned s = 100)
        {   size = s;  heapData = new int[size];  }
        // copy constructor supports call by value
        noLeak(const noLeak&);
        // overloaded = supports deep copying
        void operator=(const noLeak&);
        // destructor deallocates heapData
        ~noLeak() { delete[] heapData; }
        …
};
```

# C++ deep copying

```
// copy constructor supports call by value
noLeak::noLeak(const noLeak& src) {
        size = src.size;
        heapData = new int[size];
        for (int k = 0; k < size; k++)
                heapData[k] = src.heapData[k];
}
// overloaded = supports deep copying; check for self-assignment
void noLeak::operator=(const noLeak& src) {
        if (this != &src) {
                delete[] heapData;
                size = src.size;
                heapData = new int[size];
                for (int k = 0; k < size; k++)
                        heapData[k] = src.heapData[k];
        }
}
```

# Example 4.6 C++ class with copying suppressed

```
class noCopy {
   private:
        int*     heapData;
        int      size;
        // copying suppressed!
        noCopy(const noCopy&);
        void operator=(const noCopy&);
   public:
        noCopy(unsigned s = 100)
        {    size = s;  heapData = new int[size];  }

        // destructor deallocates heapData
        ~noCopy () { delete[] heapData; }
        …
};
```

# Shallow vs Deep Copying

- Shallow copying performs first-level bitwise copy
  - shallow copy of an address establishes an alias
- Deep copying performs two-tier copy with objects
  - For an object with heap memory
    - the address is not just copied
    - a new 'copy' of the heap memory is allocated and initialized
    - the address of this replica is then copied
  - Requires C++ copy constructor or C#/Java Cloneable

# Example 4.8 C# Cloning

```
public class uCopy: ICloneable {
        private        anotherClass   refd;
        // 'anotherClass' instance allocated on heap
        //      address of subobject held in reference 'ref'
        public uCopy() {
               refd = new anotherClass();
        }
        …
        // deep copy: more heap memory allocated via clone
        //      subobject copied into new memory
        public object Clone() {
               uCopy   local = this.MemberwiseClone() as uCopy;
               local.refd = this.refd.Clone() as anotherClass;
               return local;
        }
        …
}
```

# Example 4.8    continued

```
// application code
// #A: uCopy object allocated
uCopy        u1 = new uCopy();

// intuitive but yields shallow copy
// #B: shallow copy of uCopy object
uCopy        u2 = u1;

// deep copy: must cast to retrieve type information
// #C: clone of uCopy object
u2 = u1.Clone() as uCopy;
```

# Suppress Copying

- Copying may be suppressed when undesirable
  - Copying large registries (hash tables, etc. )
  - Generation of temporaries
  - C++ supports move semantics
- Deep copying is more expensive than shallow copying.
- Shallow copying may lead to data corruption.

# Circumvent Copying: Move Semantics

- C++11
- Enhance Performance by avoiding copying
  - Acquire heap memory of passed argument (object)
- Applicable for temporaries
  - Save constructor/destructor pairing
- Compiler decides whether to invoke
  - Copy constructor or move constructor
  - Overloaded assignment or move assignment
- Class designer need only define additional methods

# C++ 11 move semantics: denoted by ″&&″

```
// move constructor supports efficient call by value
noLeak::noLeak(const noLeak&& src) {
        size = src.size;
        heapData = src.heapData;
        src.size = 0;
        src.heapData = nullptr;
}

// move assignment exchanges ownership
// return reference to support chained assignment
noLeak& noLeak::operator=(const noLeak&& src) {
        swap(size, src.size);
        swap(heapData, src.heapData);
        return  *this;
}
```

# Garbage Collection

- Implicit deallocation
- Triggered by run-time evaluation of heap
  - Heap too fragmented or insufficient memory available
- Running program must be suspended
  - EXPENSIVE, not feasible for many real-time systems
  - No overhead if garbage collector does NOT run
- All active data is marked
  - remaining 'garbage' is removed
  - allocated but unused blocks of heap memory released.
- Insufficient heap memory
  - remedied by the reclamation of 'garbage'
  - data allocated but no longer used is released

# Example 4.9 Classic Mark and Sweep Algorithm for Identifying Garbage

```
// start with direct references, the root set:
//      all visible variables (active memory) at time of sweep
// trace out to all variable indirectly referenced
void markSweep() {
        for each Object r in rootSet
                mark(r);
}
// if heap object marked: KEEP
//      clear marked status in preparation for subsequent sweeps
// if heap object unmarked: RECLAIM (garbage)
void sweep() {
        for each Object x on heap
                if (x.marked)           x.marked = false;
                else                    release(x);
}
```

# Example 4.9 continued

```
// recursive depth-first marking
// terminates when all reachable objects marked
void mark(Object x) {
      if (!x.marked) {
            x.marked = true;
            for each Object y referenced by x
                  mark(y);
      }
}
```

# Reference Counting

- Implicit deallocation
- Allocation sets reference count to ’1’ for that data item
- Each subsequent alias increments reference count
- Reference count decremented when handle goes out of scope
- (heap) Data deallocated when reference count is ’0’

- Drawbacks
  - Unavoidable overhead for all run-time allocations and deallocations
  - Cycles cannot be detected

# Implicit Deallocation Flaws

- Garbage collection cannot accurately detect all garbage
- Lingering references prevent memory from being collected
  - Live reference to dead object keeps dead object allocated
- Garbage collector cannot accurately discern valid references
- Reference counting cannot detect cycles
  - Objects with references to other objects keep object alive
  - Problem when cycle of objects is not referenced externally

  -> 'garbage', missed, not deallocated because reference count not zero
- Weak references may ameliorate the occurrence of cycles.

# Compaction

- Reassignment of allocated blocks after garbage collector runs
  - Non-trivial, expensive!
- Shift allocated memory blocks to one end of heap
  - free memory is available in large, contiguous blocks
- An excessively fragmented heap
  - may cause severe performance degradation
  - may trigger compaction

# Figure 4.5 C++ Design Guidelines to prevent memory leaks

*Match every new with a delete*

*Class design: DEFINE*

- *constructor, destructor*

*Class design: DEFINE (or suppress)*

- *copy constructor, overloaded assignment operator*

*Use Reference Count to track aliases*

- *Increment with each added reference*
- *Decrement when alias goes out of scope*
- *Deallocate when count is zero*

*Explicitly transfer ownership*

- *Pass pointer by reference*
- *Assume ownership*
- *Reset passed (old owner's) pointer to null*

# Common Best Practices

- C++ classes with internally allocated heap memory
  - MUST make explicit design decisions with respect to copying
  - suppress copying, employ move semantics, support deep copying.
- A destructor must be defined
  - deallocate heap memory when an object goes out of scope.
- A constructor should be defined
  - set the object in an initial state
  - Including initial configuration of heap memory usage.
- C++ objects
  - for efficiency, allocate on the stack.
  - for polymorphic behavior, allocate on the heap.
- For safety, aliases should be minimized and closely tracked.