EECS 490 – Lecture 5

Memory Management

Announcements

- Project 1 and Homework 2 will be released later this week

Agenda

- **Storage Duration**
- Value and Reference Semantics
- RAII and Scope-Based Resource Management
- Garbage Collection

Static Storage

- Variables at global, namespace, or static class scope can be accessed at any time, so the associated object's lifetime must span the whole program
- Compiler/linker can determine which objects have static storage duration, so they are often placed in a special *data segment*
- C and C++ allow local variables to have static storage duration with the `static` keyword
- In some languages, initialization of static objects may be deferred until first use

Automatic Storage

- Non-static local variables come into existence upon entry to their function or block

- Stored inside *activation record* or *frame* for the block

- Frame created when block is entered, destroyed upon final exit from block

```
void foo(int x) {
  int y = x * x;
  if (y < 100) {
    int z = 100 - y;
    cout << z << endl;
  }
  cout << y << endl;
}
```

Stack-Based Memory Management

- In many languages, activation records are stored on a stack
  - Upon creation, frame pushed onto stack
  - Upon final exit, frame popped from stack
- Cannot be used in languages with full support for nested function definitions
  - Static (lexical) scope requires access to definition environment even after associated function exits

```
def foo(x):
    def bar(y):
        return x + y
    return bar

fn = foo(3)
fn(4)
```

Dynamic Storage

- Objects that are not tied to a specific scope have *dynamic* storage duration
  - Compiler cannot deduce lifetime from code
- Usually created explicitly by programmer
  - Examples: `malloc(4 * sizeof int)`, `new int[4]`
- Dynamic objects cannot be placed on stack, since their lifetime can exceed that of the block where they are created
- Instead, a special structure called a *heap* is used to store dynamic objects

Managing Dynamic Storage

- Language runtime must provide heap management
  - Find free space when dynamic object is allocated, manage free space when objects are deallocated
- Objects must be reclaimed when they are no longer in use
- User-level management: explicit calls to `free()`, `delete`
- Automatic memory management: garbage collection
  - More on this later

Agenda

- Storage Duration
- **Value and Reference Semantics**
- RAII and Scope-Based Resource Management
- Garbage Collection

Value and Reference Semantics

- Value semantics: a variable is nothing more than a name associated with an object
  - The storage for the variable is the same as that of the object itself
  - The association between a variable and an object cannot be broken as long as the variable is in scope
- Reference semantics: a variable is an indirect reference to an object
  - A variable has its own storage that is distinct from that of the object it refers to
  - A variable can be modified to be associated with a different object

Variables in C++

- Declaring a variable creates an associated object in memory

```
int x = 3;
cout << &x << endl;
x = 4;
cout << &x << endl;
```

Changes the value stored in the object, not which object **x** refers to

C++ References

- A reference in C++ is just an alias for an existing object

```
int x = 3;
int &y = x;
cout << &x << endl;
cout << &y << endl;
y = 4;
cout << x << endl;
```

Pointers

- A pointer is a kind of object whose value is the address of another object

- Refer to objects indirectly, so they provide a form of reference semantics

```
int x = 3;
int y = 4;
int *z = &x;
z = &y;
*z = 5;
cout << x << ", " << y << endl;
```

Changes which object z indirectly "refers to"

Reference Semantics

- In a language with reference semantics, variables behave like C/C++ pointers
  - But can't do arithmetic on them

- Example:

```
>>> x = []
>>> y = x
>>> id(x)
4546751752
>>> id(y)
4546751752
```

Get unique ID of object

C++ equivalent:

```
list *x = new list();
list *y = x;
cout << x << endl;

cout << y << endl;
```

- Python: everything has reference semantics

- Java: primitives have value semantics, everything else has reference semantics

Reference Semantics vs. C++ References

- Java/Python references are **not** the same as C++ references

- Java/Python: a reference **indirectly** refers to an object, can be changed

```
>>> x = []
>>> id(x)
4546749256
>>> id(y)
4546751752
```

- C++: a reference is an **alias** for an object, cannot be changed

Break Time!

- We’ll start again in five minutes.

Agenda

- Storage Duration
- Value and Reference Semantics
- **RAII and Scope-Based Resource Management**
- Garbage Collection

Internal Resources

- A data abstraction may have its own internal resources that it manages
- Example: vector
  - Allocates storage space upon construction
  - Upon insertion, if space is exhausted, allocates larger space, moves items, deallocates old storage
- Internal resources are part of the implementation, and the user of an abstraction should not have to manage its internal resources
- Internal memory automatically handled in garbage-collected languages

Dispose Pattern

- Extend the interface of an abstraction to provide functions that must be called when the abstraction is created and when it is no longer needed

- Example:

```
typedef struct { ... } vector;

void vector_init(vector *);
void vector_destroy(vector *);
```

- Relies on user to remember to call both functions at the appropriate times
  - Analogous to `malloc()`, `free()`

Constructors and Destructors

- Initialization functions formalized in object-oriented languages as *constructors*

- Some languages formalize destruction functions as *destructors*
  - Garbage-collected languages provide finalizers instead; more on this later

- A language can ensure that a constructor is always called when an object is created, and the destructor when it is destroyed
  - Static objects: upon program start and end
  - Automatic objects: when they go in and out of scope
  - Dynamic objects: when `new` or `delete` is applied to them

In some languages, constructor on first use for static objects.

Resource Acquisition is Initialization (RAII)

- General pattern for resource management using constructors and destructors
  - Also called *scope-based resource management*

- Example:

```
int main() {
  vector<int> values;
  {
    ifstream input("some_file");
    int x;
    while (input >> x)
      values.push_back(x);
  }
  ...
}
```

Better name would be *lifetime-based resource management*, since dynamic objects can also manage resources.

Scope-Based Resource Management

- RAII generally does not work for non-memory resources in garbage-collected languages

- Some languages provide specific constructs for scope-based resource management

- Example:

```
with open('some_file') as f:
    values = [int(x) for x in f.read().split()]
```

Agenda

- Storage Duration
- Value and Reference Semantics
- RAII and Scope-Based Resource Management
- **Garbage Collection**

Garbage Collection

- Languages that provide automatic memory management must implement a means of detecting when objects are no longer in use and collecting them

```
def foo():
    a = list() # object A
    b = a
    b = list() # object B
    a = None   # A no longer in use
    return     # B no longer in use after return
```

- Main types of garbage collection:
  - Reference counting
  - Tracing collectors

Reference Counting

- Each object has a count of the number of pointers or references to it

```
def foo():
    a = list()
    b = a
    b = list()
    a = None
    return
```

Object A

Assignment to a sets A's count to 1

Assignment to b sets A's count to 2

Object B

Assigning B to b sets B's count to 1 and A to 1

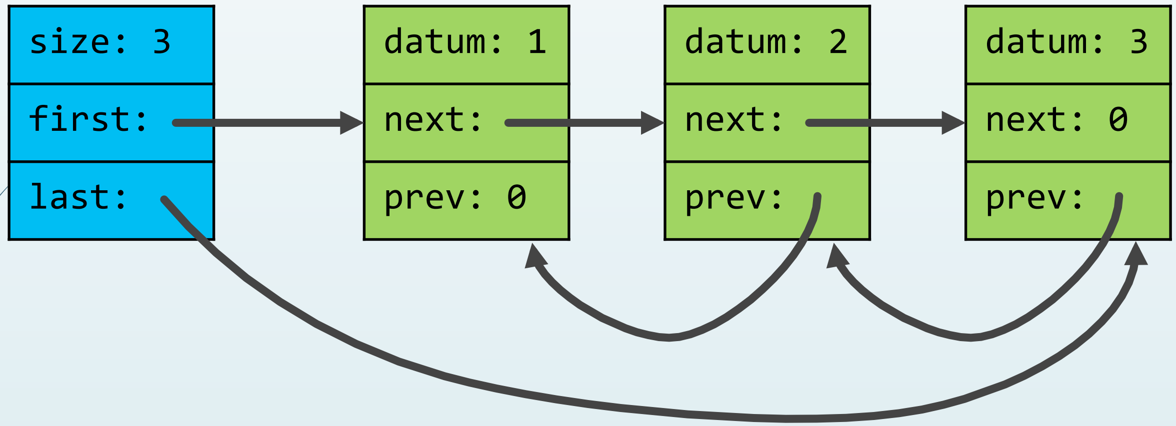
Assignment to a sets A's count to 0

Return sets B's count to 0

- When the count of an object reaches 0, it is garbage and is collected

Circular References

- Reference counting fails to detect garbage with circular references



- Implementations such as CPython include cycle detection algorithms
- Languages also might provide *weak pointers* (or references) that do not increment reference count

C++ Smart Pointers

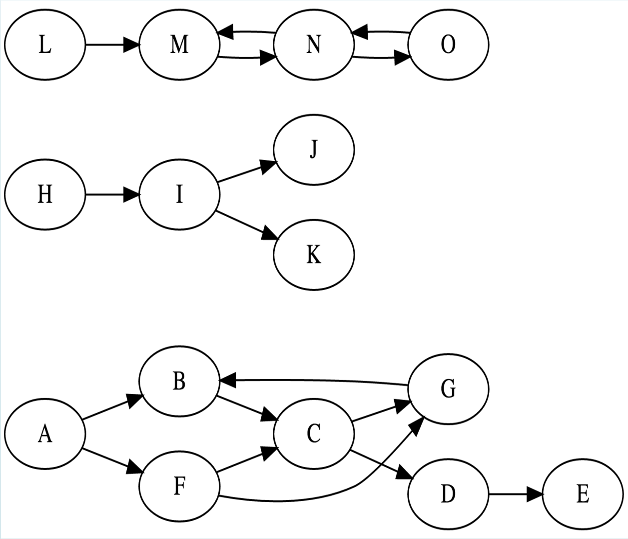
- Pointer-like objects that do reference counting

```
shared_ptr<Object> p1(new Object());
shared_ptr<Object> p2 = p1;  // count is now 2
p1.reset();   // count decremented to 1
p2 = nullptr; // count decremented, object deleted
```

- `shared_ptr`: reference counting pointer, deletes an object when count is 0
- `weak_ptr`: weak pointer that does not increment count
- `unique_ptr`: ensures that only one pointer to an object exists at a time

Tracing Collectors

- Periodic collection
- Start out from *root set* of objects
  - Generally those with static and automatic duration
- Recursively follow all pointers/references
- Objects that are reached are live, rest are dead
- *Mark and sweep*: mark all objects reached during search, then sweep rest
- *Stop and copy*: copy objects to new locations as they are encountered
  - Need to change pointers



Finalizers

- Analogous to destructor, called when an object is about to be collected
  - Java: `finalize()`
  - Python: `__del__()`

- Problems
  - May not be called in a timely manner, particularly with tracing collectors
  - Can lead to object resurrection of a reference to the object is leaked
  - Do not run in a well-defined order
  - Are not guaranteed to run in many implementations