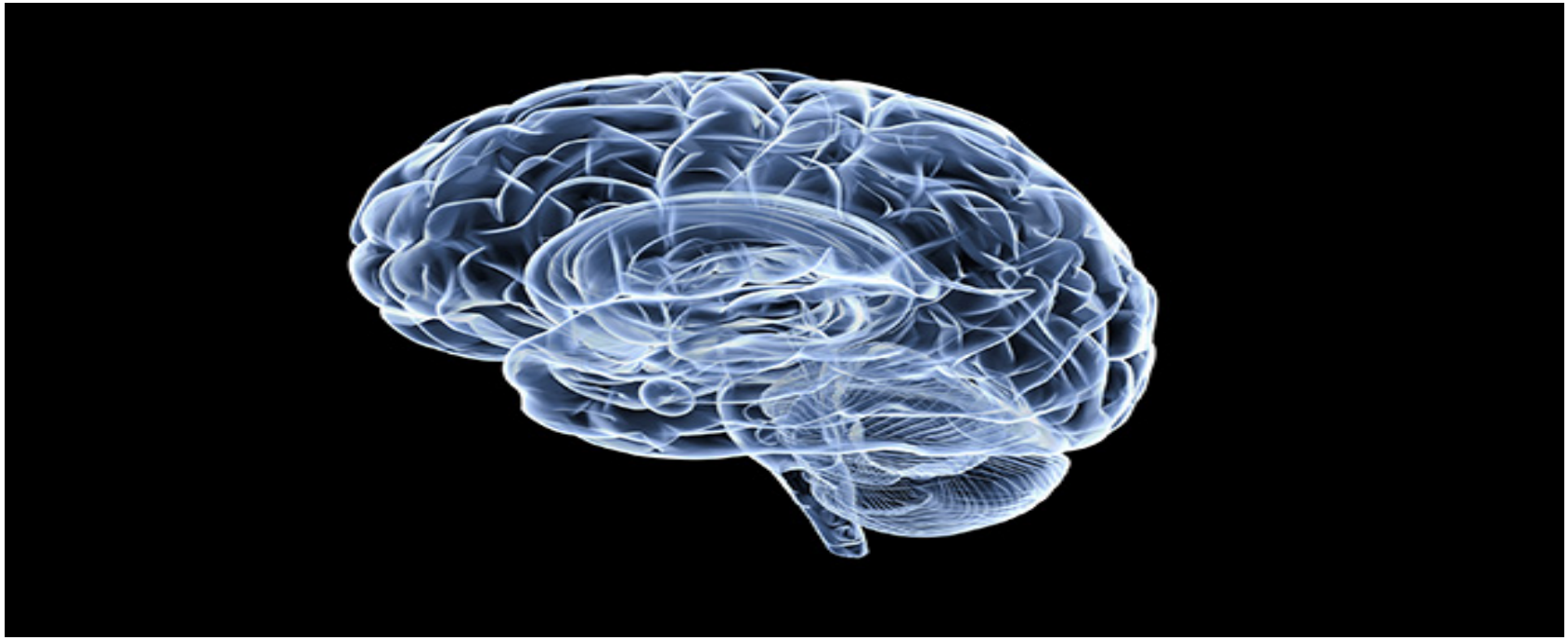

http://www.wholehealthinsider.com

C++ Memory Management

Yngvi Björnsson

Overview

- Scope, Duration, and Linkage
- Dynamic Memory Allocation
- Garbage Collection vs. RAII
- Smart Pointers

Scope, Duration, and Linkage

- **Scope**
  - Where can variable (objects) be referred to?
  - Local, Class, Global, Namespace, Statement
- **Storage Duration (lifetime)**
  - When are variables (objects) created and destroyed.
  - Automatic, Dynamic, Static, Thread [since C++11]
- **Linkage**
  - A variable's **linkage** determines whether multiple instances of an identifier refer to the same variable or not.
  - No, Internal or external

Scope

- Variables (and functions) can only be referred to in certain parts of a program --- its **scope**.
- **Local Scope**
  - Extends from declaration within { …} to end of the block, e.g. local variables
- **Global Scope**
  - Extends from declaration (outside blocks) to end of the file, e.g. global variables. May or may not be accessible from other files.
- **Other:**
  - **class scope, namespace scope, and statement scope**

(Storage) Duration

- **Automatic**
  - Allocated when program flow enters scope, deallocated when leaving score
  - Local variables, stored on the stack
    - Not initialized by default
- **Dynamic**
  - Allocated/deallocated when explicitly constructed/destructed (with new/delete)
  - Stored on the heap
- **Static**
  - Allocated at beginning of program execution (or first use), maintain their value throughout execution (independent of scope), deallocated at end of execution.
  - Global variables and variables explicitly defined as static
    - Zero-initialized by default
  - Note: Thread duration since [C++11] (tied to thread's lifetime, not program's)

Linkage

- **No Linkage**
  - Name can only be reference in the scope it is defined in.
  - E.g. names defined at block scope, including local variables.
- **Internal Linkage**
  - Name can only be seen in the file (translation unit) it is defined in.
  - E.g., static and const variables
- **External Linkage**
  - Name can be seen in files (translation units) other than where defined.
    - By using the **extern** quantifier in the other files.
  - E.g., non-static global variables

Programming example (two files)

```
const int cg = 5;
int nsg;
static int sg;
namespace ns { int c; }
extern void some_func_in_another_file();
int main() {
   int a = 10;
   for ( int i=0; i<2; ++i ) {
      int a = 0, b = 0;
      static int c = 0;
      ++a; ++b; ++c; ++ns::c; ++nsg;
      some_func_in_another_file();
      cout << "in: " << a << ' ' << b << ' ' << c << endl;
   }
  cout << cg << ' ' << nsg << ' ' << sg << endl;
  cout << a << ' ' << ns::c << endl;
}
```

```
const int cg = 10;

extern int nsg;
static int sg = 100;

void some_func_in_another_file() {
   nsg += cg;
   sg += cg;
}
```

```
in: 1 1 1
in: 1 1 2
5 22 0
10 2
```

Dynamic Memory Allocation

- Allocated memory

```
Obj* a = new Obj();
Obj* arr = new Obj[10];
```

Deallocate memory

```
delete [] arr;
delete a;
```

- Allocated memory **must** be deallocated, otherwise a memory leak
- C++ memory management:
  - ***No default garbage collection***
  - Rather, "manual" *Resource Acquisition Is Initialization* (RAII) technique:
    - Bind lifecycle of a resource to that of the acquiring object, e.g., for memory, allocate in constructor and deallocate in destructor

Example

```
int a;  // static duration

void foo() {
   int b = 0;  // automatic duration
   int* c = new int;    // dynamic duration of int (pointer automatic duration).
   int* d = new int[3]; // dynamic duration of int[] (pointer automatic duration).

   delete[] d;  // note the [], needed for arrays.
   delete c;    // important to delete d and c, otherwise memory leak.
}
```

Memory Allocation --- error check

```
// Method one.
try {
   int* data = new int[1000];
   // ...
   delete[] data;
}
catch (std::bad_alloc& ba) {
   std::cerr << "bad_alloc caught: " << ba.what() << '\n';
}
// Method two.
int* data = new (std::nothrow) int[1000];
if ( data != nullptr ) {
   // ....
   delete[] data;
}
```

Smart Pointers

- Make memory management more manageable
- **unique_ptr**
  - Represent unique ownership.
  - Implemented as a light-weight wrapper
- **shared_ptr**
  - Represent shared ownership
  - Sometimes need to also use associated **weak_ptr** some avoid circular issues
  - Implemented using reference counting (“*last one out turns the lights off”*)
- **auto_ptr**
  - Deprecated, will be removed in future versions **Do not use!**

Example

```
void foo() {
    std::unique_ptr<int> sp(new int[10]);

    // ... do something ...

    // No need to delete, sp destructor does that!
}
```

Questions?