C++: Memory Management

Announcement

- RIT Career Fair
  - Thursday, Oct 3rd 1pm – 7pm
  - Friday, Oct 4th 9am – 5pm (interviews)
- Clark Gym
- www.rit.edu/co-op/careers

Announcement

- CS Student Meeting
- Wednesday, Oct 2nd
- 5-6pm
- 08-1250
- Followed by free pizza in Building 10 (outside of ICLs)

Announcement

- Date for Exam 1 has been changed!
  - New date: Monday Oct 7th

Project

- Questions?
- Everyone have a partner?
- Please e-mail me with the name of your partner and I will assign you a group account.
- **Design diagrams due Oct 1st !!!!**

Plan for this week

- Today: Intro to Memory Management
  - … plus a bit about Templates
- Tuesday: Memory Problems
- Wednesday: Case Study: smart pointers!

Storage Types

- Global (static) data
  - come into being when the program starts and are destroyed when the program ends
- Local (automatic) data
  - come into being when a function is called (or a block entered) and are destroyed at close of scope
- Dynamic (heap) data
  - allocated by invoking the `new` operator and are destroyed by invoking the `delete` operator.

Object Data

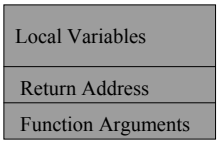
- There is also data that exists as part of an object of a class.
  - The creation / destruction of this data is dictated by the constructor / destructor of the class.

Program Memory

- The memory used by a program is generally separated into the following sections:
  - Code – Where the executable code is kept
  - Global – Where storage for global variables is kept
  - Stack – Runtime stack (where local variables are kept)
  - Heap – Free store for dynamically allocated variables.

Runtime Stack

- Every time a function is called, information about this function is pushed on the runtime stack.
  - Function Frame:



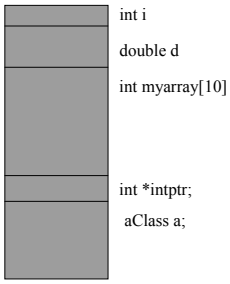
Runtime Stack

- The compiler can determine how much memory is required in the local variables area for a given function.

Runtime Stack

```
void foo ()
{
    int i;               // size of int
    double d;            // size of double
    int myarray[10];     // 10 * size of int
    int *intptr = new int [10];  // size of
  pointer
    aClass a (1,2,3);    // size of aClass object
    …
}
```

Runtime Stack

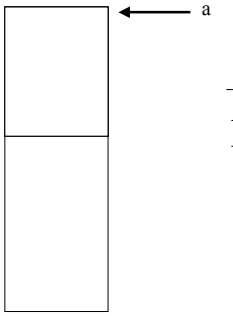


- Local variable memory for `foo()`

Runtime Stack

- When function returns, the function frame, including local variables are popped off the stack
  - Makes room for next function call.
  - Important safety tip: Do not return references or pointers to local variables!

Runtime Stack




```
void f()
{
    aClass &a = badFuct();
    ...
    anotherFunction();

    int I = a.moo;
}
```

Free Store (Heap)

- Dynamically allocated variables are stored on the heap.
- `new` – allocates space for data/ objects on heap
  - Returns pointer to newly allocated object
- `delete` – releases memory for previously allocated data/objects
  - Called on pointer previously returned by `new`.
  - Each object on the heap should be deleted once and only once!

Free Store (Heap)

```
aClass *a = new aClass (…);
delete a;

// allocates space for an array of 20 aClass objects
// Note: aClass must have default constrcutor
aClass *aArray = new aClass [20];
aClass *aArray2 = new aClass [20];
delete [] aArray;  // heap can determine size
delete aArray2  // destructor only called on aArray2[0]

// allocate local array...on stack NOT heap
aClass localArray[20];
```

Free Store (Heap)

- Although usually large, heap is not infinite.
  - It is possible to run out of heap space.
  - If your heap is full…game over!
  - Important to deallocate (delete) objects when they are no longer needed.
  - Note: if heap is full, `new` will return 0.

Stack & Free Store

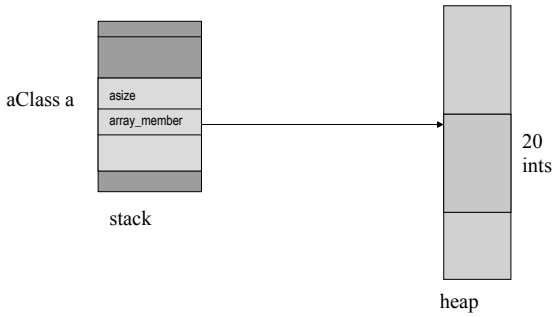
```
class Foo
{
private:
    int *array_member;
    int asize;
    ...
public:
    Foo (int size);
    ~Foo ();
}
```

Stack & Free Store

```
Foo::Foo (int size) :
 asize (size), array_member (new int[size])
{
  for (int i=0; i<size; i++)
      array_member[i] = 0;
}

void f ()
{
  // local aClass object
  aClass a (20);
  ...
}
```

Stack and Free Store



Stack and Free Store

- Questions?

A quick intro to Templates

- Problem:
  - Let's say that we need a Queue class that manages a Queue of ints:

```
class intQueue
{
 private:
    int *q;
    int n;
    …
public:
    void enqueue (int i);
    int dequeue();
   …
}
```

A quick intro to Templates

- Problem:
  - Now, let's say that we need a Queue class that manages a Queue of double:

```
class doubleQueue
{
 private:
    double *q;
    int n;
    …
public:
    void enqueue (double i);
    double dequeue();
   …
}
```

A quick intro to Templates

- Note that the code for intQueue will be almost identical to that of doubleQueue.
- Is there a way to reuse the basic Queue code but have it work on different datatypes?

A quick intro to Templates

- One solution:

```
class voidQueue
{
 private:
    void **q;
    int n;
    …
public:
    void enqueue (void *i);
    void * dequeue();
   …
}
```

A quick intro to Templates

- One solution
  - This is sort of the solution used by Java.
  - Problems:
    - Programmer must know what kind of data each queue is managing
    - Must cast the returned (void *) to correct type
      - Let's hope he/she gets it right!

A quick intro to Templates

- The template solution:
  - Define a "generic" queue class
  - Fill in the datatype of the objects managed by the queue later.
  - Can define such a class using Templates:

A quick intro to Templates

```
template <T>
class Queue
{
 private:
    T *q;
    int n;
    …
public:
    void enqueue (T i);
    T dequeue();
   …
}
```

A quick intro to Templates

- To use this template:

```
// a queue of ints
Queue<int> iqueue;

// a queue of doubles
Queue<double> dqueue;

// a queue of aClass objects
Queue<aClass> aqueue

// a queue of pointers to aClass
Queue<aClass *> aptrqueue
```

A quick intro to Templates

- C++ provides for
  - Class templates – e.g. queue
  - Function templates – define "generic" functions that can take arguments of different types
    - Sorting, Searching, etc.

A quick intro to Templates

- Standard Template Library (STL)
  - A library of commonly used templated classes and functions.
  - Much like the .util package in Java.
  - Includes
    - Collections (Vectors, Queues, Stacks, Maps)
    - Iterators
    - Functions

Templates and You

- Why bring up templates now?
  - For the project, you will be maintaining a Queue of Problem configurations
    - Use the STL queue rather than writing your own!
    - Implement your "generic" problem solving framework as a Templated class
  - On Thursday we will be talking about smart pointers which make use of Templates.

Summary

- Memory
  - Allocation types
  - Heap vs. Stack
- A bit about Templates
  - More on Templates and STL in weeks to come.
- Questions?