COMP322 - Introduction to C++

Lecture 04 - Memory management

Dan Pomerantz

School of Computer Science

29 January 2013

Memory management in C++

Named objects in C++ are stored in one of two ways:

- **Static:** Applies to global variables and local variables with the qualifier `static`. The storage is allocated permanently during execution of the program.
- **Automatic:** Applies to all other local variables. These are allocated only while the relevant variable is in scope.

```
double sqrt_by_newton(double x) {
  double delta;
  double y = x;              // initial guess

  do {
    double new_y = y - (y * y - x) / (2 * y);
    delta = y - new_y;
    y = new_y;
  } while (delta > 1e-12);  // new_y deleted here
  return (y);
} // delta and y deleted here
```

Using pointers to automatically allocated spaces

After your method finishes, any data that was stored in an automatically generated variable, will be meaningless.

```
double* foo() {
     double x = 3; //create automatically generated variable
     double* pointer = &x; //gets the address of x
     //Next line compiles but the address of x has no meaning
     //after the method leaves
     return pointer;
}

int main() {
    double* r = foo();
    cout << *r << endl; //The value pointed to by r is not known!
}
```

The free store in C

The free store, or heap, allows us to create objects with "intermediate" lifetimes.

These objects don't have names, but their addresses may be assigned to pointer variables.

The C heap is accessed by the functions `malloc()` and `free()`.

- `void *malloc(size_t nbytes)` returns a pointer to a region of memory at least 'nbytes' in size. It returns NULL if no such region is available.
- `void free(void *ptr)` marks the region as unused.

An pointer returned by `malloc()` can be used until it is explicitly released by calling `free()`.

The free store in C

- The argument to `malloc()` is the number of bytes required, often a `sizeof` expression.
- The memory returned is *not* initialized in any way.
- `malloc()` returns `void *`, so we cast the return value to the required pointer type.
- Any argument to `free()` must have been returned by `malloc()`

```
 struct MatrixElement *create_Element(int row, int column, int val
  MatrixElement* new_p;
  new_p = (struct MatrixElement *)
                   malloc(sizeof(struct MatrixElement));
  //Now initialize the data p is pointing to.
  //We need to dereference p and then set it's values
  //using dot operator.
  (*p).row = row;
  p->col = col; //same as (*p).col = col
  return (new_p);
}
void delete_Element(MatrixElement* element_p) {
  free(element_p); // Memory is no longer in use
}
```

The free store in C++

While `malloc()` and `free()` remain part of the C++ standard library, C++ introduces two operators which manipulate the free store.

- `new` - Allocate memory on the free store.
- `delete` - Free memory if the argument is non-zero.

```
MatrixElement* create_element(int row, int column, int value) {
  //note that new returns a pointer
  MatrixElement* newElement = new MatrixElement;
  newElement->row = row;
  ...
  return (newElement);
}

void delete_element(MatrixElement* element_p) {
  delete element_p; // Memory is no longer in use
}
```

Comparing two ways of coding

When we want to return a pointer, we need to return an address that was created using the new operator. Otherwise, it will go out of scope and the address will potentially store something else.

```
MatrixElement* createElement(int row) {
     MatrixElement newElement;
     newElement.row = row;
     return &newElement; //address has no meaning!
}
```

vs.

```
MatrixElement* createElement(int row) {
     MatrixElement* newElement = new MatrixElement;
     (*newElement).row = row;
     return newElement; // now the address has a meaning!
}
```

In the second example, the address is *guaranteed* not to change until we use the delete operator to *free* the memory. In the first case, anything can happen.

Returning a pointer

So if we want to return a pointer, we must use the new operator.

Why would we want to return a pointer?

- Faster (less to copy)
- Returning an array
- Easier to make modifications to (there is such thing as NULL or empty)

Details of the `new` operator

- A `new` expression implicitly calls the *constructor* for an object, which may initialize the object.
- However, the initial value of the memory is undefined.
- We can specify initial arguments for the constructor:

```
double *pd1 = new double(1.1); // Sets the initial value
```

- If a `new` operation fails, it throws an *exception*. By default this will end the program.
- We can suppress the exception with the `nothrow` parameter:

```
#include <new>
symbol *sym_p = new(std::nothrow) symbol();
if (sym_p == NULL) { // Allocation failed
```

Creating and deleting arrays

The `new` and `delete` operators also work with arrays. This allows us to set the lengths of arrays at runtime.

```
int *create_vector(int size) {
   int *vector = new int[size]; // Allocate 'size' integers
   // perform some initialization, e.g.
   return vector;
}

void delete_vector(int *vector) {
  delete [] vector; // brackets tell delete this is an array
}
```

Note that we *cannot* provide explicit initialization for the individual elements in the array.

We must specify `delete []` when deleting an array.

Creating multidimensional arrays

The syntax of C++ does not allow for easy creation of multidimensional arrays:

```
float **matrix = new float[10][10]; // Illegal!
```

Instead we have to use a more complex initialization:

```
float **matrix = new float *[10];
int i, j;
for (i = 0; i < 10; i++) {
  matrix[i] = new float[10];
}
// now we can access matrix[i][j]
```

Many of these sort of things are better handling using the standard library.

Advantages of C++ memory management

The `new` and `delete` operators provide several advantages over `malloc()` and `free()`.

1. No need to cast - New automatically returns a pointer of the correct type.
2. No need to explicitly calculate the size of the object.
3. Initialization is performed if a constructor is defined.
4. Can throw an exception on failure, which can simplify error handling.
5. Will call the *destructor* before deleting an object, if defined.
6. You can override the `new` operator and provide your own implementation for debugging or other special purposes.

Common memory management errors

There are a large number of ways in which memory management can go wrong.

- Deleting a pointer that was not returned by `new`:

```
int n;
int *p = &n;
// ...
delete p;           // error!

int *pv = new int[5];
pv++;               // alters the pointer
delete [] pv;       // error!
```

- Deleting the same pointer twice:

```
int *p = new int(10); // Initialize *p==10
// ...
delete p;
// ...
delete p;           // error!
```

Common memory management errors

- Failing to delete allocated memory (*Memory leak*):

```
char *linebuf = new char[1024];

// ...

linebuf = new char[128]; // Another object

// ...

delete linebuf;      // deleted 2nd, but not 1st object
```

- Assuming the memory is initialized:

```
float *pv1 = new float[size];
float sum;

sum += pv1[0];  // The value of pv1[0] is undefined!
```

Interesting exercise to try at home

Define a struct `HugeData` and make one of the contents `int takesSpace[10000]`

Now put into a loop the following:

```
int* vector = new HugeData;
```

The above in a loop is a memory leak. Now see how many iterations it takes until your program crashes.