Dynamic Memory Allocation

November 30, 2010

Remember Memory?

We've disussed:

- Variables that associate portions of program memory with a name in our program.
  ```
  int x(10);
  ```
- References that pass the memory locations of variables to functions, instead of a copy of the variable value.
  ```
  ostream& print( ostream& os, const className& rhs );
  ```
- Arrays, that associate a region of memory *holding one or more same-type variables* with a name in our program.
  ```
  double dblArray[5] = { 3.2, 4 };
  ```

It's time we dealt with memory directly — using *dynamically allocated arrays*.

Dynamic Memory Allocation

Dynamic Memory Allocation Arbitrarily sized memory space requested by the programmer using the C++ `new` operator, allocated by the computer operating system, and de-allocated by the programmer using the C++ `delete[]` operator.

Program Free Store or Heap Name of memory region where dynamically allocated memory resides within a program (as opposed to the program stack, program disk space, CPU registers, ...).

- Allows a programmer great flexibility in the amount of data a program may process.
- Allocated and deallocated within one runtime session of the program. Not retained *across* different sessions of the same program.
- Dynamic memory is finite.

The `new(nothrow)` Operator

`new(nothrow)` does the following:

1. Allocates memory space for a specified array.
2. Calls the appropriate constructor for each element.
3. On success, returns a pointer to the newly created array.
4. On failure, returns the value `NULL` or throws an exception.

`NULL` is a symbolic constant in C++ (like a definition for PI), usually defined to be zero.

But always use `NULL`, don't assume it is zero!

We don't study exceptions in the course, this is why we specify `nothrow` in `new(nothrow)`.

Using `new(nothrow)` and `delete`

```
14 int main()
15 {
16   // allocate space for an unsigned integer
17   int size(0);
18   while( size <= 0 && cin ) {
19     cout << "Number of elements (>0)?" << flush;
20     cin >> size;
21   }
22   if( !cin ) {
23     // failure reading from user
24     cout << "Error reading input." << endl;
25     exit(1);
26   }
27
28   // allocate size doubles as an ARRAY
29   double* const dynArray = new(nothrow) double[size];
30   if( dynArray == NULL ) {
31     cout << "Yikes! not enough memory." << endl;
32     // Gotta do this now!
33     exit(1);
34   }
```

`new` can be used to allocate an array of fundamental data types or user-defined objects

Note the pattern:

1. Declare a `*const` pointer initialized with the `new` result.
2. Check against `NULL`.
3. Use the memory as if it were a 1-dimensional array.
4. `delete[]` the memory when finished.

Using `new(nothrow)` and `delete`

```
28   // allocate size doubles as an ARRAY
29   double* const dynArray = new(nothrow) double[size];
30   if( dynArray == NULL ) {
31     cout << "Yikes! not enough memory." << endl;
32     // Gotta do this now!
33     exit(1);
34   }
35
36   // use the memory space to get some work done
37   for( int i=0; i<size; i++ ) {
38     dynArray[i] = i + 1/double(i+1);
39   }
40
41   useArray( dynArray, size );
42
43   // delete[] the memory --- allow the operating
44   // system to reuse for other applications.
45   delete[] dynArray;
46   return 0;
47 }
```

What is this?

`double* const dynArray`

1. It's called a pointer.
2. It's initialized with the result of `new`.
3. It holds an address (location) of memory.
4. It can be used like a 1-dimensional array...

Using `new(nothrow)` and `delete`

```
28   // allocate size doubles as an ARRAY
29   double* const dynArray = new(nothrow) double[size];
30   if( dynArray == NULL ) {
31     cout << "Yikes! not enough memory." << endl;
32     // Gotta do this now!
33     exit(1);
34   }
35
36   // use the memory space to get some work done
37   for( int i=0; i<size; i++ ) {
38     dynArray[i] = i + 1/double(i+1);
39   }
40
41   useArray( dynArray, size );
42
43   // delete[] the memory --- allow the operating
44   // system to reuse for other applications.
45   delete[] dynArray;
46   return 0;
47 }
```

`delete[]` deallocates (“frees”) memory allocated by `new`

`delete[]` does not “erase” the pointer, only the memory the pointer references.

Use `delete[]` only on pointers initialized with `new`.

Do not use `delete[]` multiple times on the same pointer variable.

Use Dynamic Memory Like Static Arrays

```
5  // the allocated memory is used in here...
6  // AS IF IT WERE A STATIC ARRAY
7  void useArray( double array[], const int size )
8  {
9      cout << array[0] << "␣";
10     cout << array[size/2] << "␣";
11     cout << array[size-1] << endl;
12 }
```

Called from here...

```
36   // use the memory space to get some work done
37   for( int i=0; i<size; i++ ) {
38     dynArray[i] = i + 1/double(i+1);
39   }
40
41   useArray( dynArray, size );
```

Notice that `useArray` has array parameters declared, but it can be passed a pointer to dynamic memory just as well.

Common Errors Using Dynamic Memory Allocation

- Dereferencing a `NULL` pointer.
- Using `delete` instead of `delete[]` when deallocating an array.
- Forgetting to deallocate memory (with `delete[]`) when no longer needed (memory leak).
- "Losing" a pointer to allocated memory, so that it cannot be deallocated (memory leak). Use
  `int* const p( new(nothrow) int[1000] );`
  to prevent this.
- Using `delete[]` on a memory address not previously returned by `new`.

finis