CSCI 262
Data Structures
15 – Dynamically Allocated Memory

Dynamic Array Allocation
What if you know you’ll need an array, but not the size?

```cpp
int *arr, sz;
cout << "What size do you need?" << endl;
cin >> sz;
arr = new int[sz];
```

Where Does Memory Come From?

The stack: local variables, function arguments, return values. Grows "down".

The heap: dynamically allocated memory. Grows "up".

Global and static variables, constants.

Program code. Read only!

The Stack

- Holds “stack frames” aka “activation records”
- Each function call results in a new stack frame
- Each stack frame contains memory for:
  - Local variables declared in the function
  - Parameters passed into function
  - Return address for function
- When the function is exited, all of this memory is returned to the stack automatically.

Data Segment/BSS

Global and static variables:
- Only ever one instance of them
- Get stored in their own special area
- Memory is pre-allocated, fixed in size

The Heap

A big ol’ chunk of memory!

- Get pieces of it (“allocate memory”) using `new`
- Pieces stay allocated until explicitly released by use of `delete`

Heap memory has a lifetime independent of scope – it can be passed out of functions, for instance. You can’t do that with local variables!
Dynamic Arrays

Allocate dynamic arrays using new:
   double *darray = new double[1024];

Use the array pointer just like a regular array:
   for (int j = 0; j < 1024; j++)
     darray[j] = j;

Always clean up (deallocate) when you are done:
   delete[] darray;

Dynamic Arrays: Rules

Never:
- Dereference a pointer which has not been set (using new or &)
- Dereference a pointer to memory which has been deallocated (a dangling pointer)
- Change or lose a pointer which is pointing to dynamically allocated memory (or you won’t be able to deallocate – this causes a memory leak)
- Use delete on a pointer which isn’t pointing to dynamically allocated memory (e.g., a dangling or NULL pointer)

Pointers, Objects, and Dynamic Memory

Recall our student class:
   class student {
      public:
         string name;
         string year;
         double gpa;
         bool is_hungry;
         student();
         student(string);
         void eat();
         void sleep();
         void program(int);
   };

Creating New Objects: Stack

If we want to create a student locally:
   student student1;
   student student2("Kirk");

These are created on the stack.
They will vanish when exiting the current scope.

student1 is created using the default constructor:
student1();
student2 is created using another constructor:
student(string);

Creating New Objects: Heap

We can also create single objects dynamically:
   student* sp1 = new student;
   student* sp2 = new student("Picard");

These are created on the heap.
They will live forever unless deleted:
   delete sp1;
   delete sp2;

Note, again, we use two different constructors.

Working With Object Variables

Consider:
   student student1;
   student* p = new student;

We know that we can do:
   student1.name = "Sisko";
   student1.eat();

What can we do with p?
Working with Object Pointers

We have:

```cpp
student* p = new student;
```

We could just dereference (perfectly fine!)

```cpp
(*p).name = "Janeway";
(*p).sleep();
```

C++ gives us another operator we can use directly:

```cpp
p->name = "Archer";
p->sleep();
```

Note that this won't work correctly:

```cpp
*p.name = "Janeway";
```

The . has higher precedence than *

The Destructor

The counterpart to the constructor:

– No return type
– Name is ~ followed by class name, e.g.,
  ```cpp
  ~student();
  ```
– Never takes a parameter!

The destructor is called automatically when:

– A local (stack allocated) object goes out of scope
– delete is called on a dynamically allocated object

Arrays of Objects

We can also use new to create arrays of objects:

```cpp
int n = 100;
student* arr = new student[n];
The default constructor is used to create every object in the array.
```

Note different syntax if we use pointers vs array indexing:

```cpp
for (int i = 0; i < n; i++)
  arr[i].gpa = 4.0;
```

As with base types, we use delete[] on dynamically allocated arrays of objects:

```cpp
delete[] arr;
The destructor is called on every object in the array.
```

2-D Arrays

For multi-dimensional arrays, you need arrays of pointers, e.g.:

```cpp
// make a 5 x 7 2-D int array
int **arr, i;
arr = new int*[5];  // allocate array of int pointers
for (i = 0; i < 5; i++)
  arr[i] = new int[7];  // allocate columns
arr[2][6] = 42;  // etc.

// clean up
for (i = 0; i < 5; i++)
  delete[] arr[i];  // deallocate columns
delete[] arr;   // deallocate array of int pointers
```