Dynamic Allocation: Linked Lists

In a previous lecture, we said:

**Where do nodes come from...?**

Don’t worry about *where* just yet, but here’s *how* we do it:

```c
node* ptr = new node;
```

Declare a pointer variable (of type pointer to node) and create (dynamically allocate) a node object.

New is a C++ keyword. You can’t use it as a variable name, etc.

This Lecture

We’ll return to the topic of dynamic allocation of objects.

We’re going to cover some other stuff first, though.

Here’s the outline for today:

- Pointers and arrays
- Pointer arithmetic
- Dynamic array allocation
- Dynamic object allocation

Arrays

Arrays are just sequential chunks of memory:

```c
char s[5] = {'H','e','l','l','o'};
```

POINTERS AND ARRAYS

Arrays are just sequential chunks of memory:
Arrays and Pointers

Array variables are secretly pointers:

```cpp
char s[5] = {'H','e','l','l','o'};
char *p = s;
cout << *p << endl; // prints 'H'
cout << *s << endl; // also prints 'H'
```

Pointers are also secretly array variables:

```cpp
for (int j = 0; j < 5; j++) {
    cout << p[j] << endl;
}
```

To the compiler:

```
p[j] == *(p + j)
```

POINTER ARITHMETIC

POINTER ARITHMETIC

- Pointer arithmetic depends on type
  - char* p ─► p++ advances by 1 byte
  - int* q ─► q++ advances by 4 bytes (size of int)

- You can add or subtract:
  - q-- ─► go back 4 bytes
  - q + 3 ─► q plus 12 bytes

- Just keep in mind the array/pointer equivalence:

```
someint* ptr;
*(ptr + k) == *(ptr + k)
```

```
// k * sizeof(someint)
```

```
// bytes after ptr
```

POINTER ARITHMETIC

```
Now, suppose we have:
int arr[] = {42, 17, 33, 6};
int* q = arr;
```

It can be demonstrated that:

```
q[j] == *(q + j) == arr[j]
```

This implies that:

```
q[1] == *(q + 1) == arr[1] == 17
```

Then q + 1 is not simply 1 byte address beyond q, but must be 4 bytes beyond q.
**C-style Strings**

In C, strings are simply arrays of char:

```c
char *s = "Hello!";  // valid in C; for C++ add const
```

This array has size 7, not 6: the last entry stores byte value 0, or '\0':

```
\x00\x00\x00\x00\x00\x00\x00
```

Without this value, there’s no way to detect the end of a string! With it, though, we can do:

```c
for (char* p = s; *p != '\0'; p++) { ... }
```

String literals in C++ are still stored this way, but convert to the string type:

```c
string foo = "Hello";
```

or

```c
string("Hello")
```

---

**Pointers and Reference Parameters**

Reference parameters are not pointers! Reference parameters are not pointers! Reference parameters are not pointers!

If you have a function

```c
void foo(int &x) { ... }
```

Inside `foo`, you cannot do

```c
*x = 10;  // incorrect!
```

You just do

```c
x = 10;  // correct
```

---

**Array Limitations in C++**

Standard C++ does not let you do this*:

```c
int sz;
cout << "What size do you need?" << endl;
cin >> sz;
ext arr[sz];  // compiler error
```

*Strangely, later versions of C do allow this. Confusingly, so does g++ (some versions).

---

**Dynamic Array Allocation**

So what if you know you’ll need an array, but not the size (at compile time)?

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

Note that `new` gives us a pointer to our memory.
Where Does Memory Come From?

- **Stack**: local variables, function arguments, return values. Grows "down".
- **Heap**: dynamically allocated memory (using `new`). Grows "up".
- **Data Segment**: Global and static variables, constants.
- **Text Segment**: 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.

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 used after a function that created it returns. You can’t do that with local variables!

Stack vs Heap

- **Stack**: local aka automatic variables and arrays:
  - `int z;`
  - `foo f;`
  - `double darray[100];`

- **Heap**: dynamically allocated objects and arrays:
  - `int* p = new int;`
  - `foo* fp = new foo;`
  - `double* dptr = new double[100];`

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

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] = 0.0;
```

Always clean up (deallocate) when you are done:

```
delete[] darray;
```
Pointers, Objects, and Dynamic Memory

Consider this simple class:

```cpp
class student {
public:
    string name;
    student() { ; }
    student(string n) { name = s; }
    void eat();
    void sleep();
};
```

Creating New Objects: Stack
If we want to create a student locally:

```cpp
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:

```cpp
student();
```

student2 is created using another constructor:

```cpp
student(string s);
```

Creating New Objects: Heap
We can also create single objects dynamically:

```cpp
student* sp1 = new student;
student* sp2 = new student("Picard");
```
- These are created on the heap.
- They will live forever unless deleted:

```cpp
delete sp1;
delete sp2;
```

Note, again, the two different constructors.

Working With Object Variables
Consider:

```cpp
student student1;
student* p = new student;
```

We know that we can do:

```cpp
student1.name = "Sisko";
student1.eat();
```

What can we do with p?

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

Note that this won't work correctly:

```cpp
*p.name = "Janeway";
The . has higher precedence than *
```

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();
```
The Destructor

The counterpart to the constructor:
- No return type
- Name is ~ followed by class name, e.g., ~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.
```

Now we can do, e.g.:
```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.
```

Dynamic Memory Don’ts

Never:
- Dereference a pointer which has not been set to valid memory (using new or &)
- Dereference a pointer to memory which has already 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)

WRAPPING UP

Up Next

- Reading: Chapter 7
- Wednesday, March 13
  - ArrayList (how to implement a vector, part 1)
  - Reading: Chapter 13.1
- Friday, March 15
  - Lab 9 – TBD
  - Project 3 – Evil Hangman due
  - New assignment – TBD