Arrays

Arrays are just sequential chunks of memory:

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

Arrays and Pointers

Array variables are secretly pointers:

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

```cpp
cout << *p << endl; // prints 'H'
cout << *s << endl; // also prints 'H'
```

Pointers are also secretly array variables:

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

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

To the compiler:

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

POINTER ARITHMETIC
**Pointer Arithmetic: char**

Suppose:

```cpp
cchar s[] = {'H', 'e', 'l', 'l', 'o'};
```

char* p = s;

We’ve stated that:

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

Another way to look at it is:

```
p[j] == s[j] when p == s
```

Thus:

```
```

**Pointer Arithmetic: int**

Now, suppose we have:

```cpp
int arr[] = {42, 17, 33, 6};
```

```cpp
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.

**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 - 4 → go back 4 bytes
  - q + 3 → q plus 12 bytes
- Just keep in mind the array(pointer equivalence:
  ```cpp
  sometime* ptr;
  *(ptr + k) == ptr[k]
  // k * sizeof(something)
  // bytes after ptr
  ```

**C-style Strings**

In C, strings are simply arrays of char:

```cpp
char *s = "Hello!";
```

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

```
\0
```

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

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

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

```cpp
string foo = "Hello";
```

or

```cpp
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

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

Inside foo, you cannot do

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

You just do

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

Sources of confusion:
  - & denotes a reference parameter
  - & also used as address-of operator
  - References use pointers "under the covers"
Array Limitations in C++

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

```cpp
text
int sz;
cout << "What size do you need?" << endl;
cin >> sz;
int 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)?

```cpp
text
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**
  - 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.

- **Heap**
  - Dynamically allocated memory (using `new`). Grows "up".

- **Data Segment**
  - Global and static variables, constants.

- **Text Segment**
  - Program code. Read only!

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 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’ hunk 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];`

  Memory for these is allocated on the stack when they come into scope, is returned to the stack when they go out of scope (e.g., when function returns).

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

  All of these live on the heap. They will exist until explicitly deallocated by user code.

Dynamic Arrays

Allocate dynamic arrays using **new**:

```cpp
double* darray = new double[1024];
```

Use the array pointer just like a regular array:

```cpp
for (int j = 0; j < 1024; j++)
    darray[j] = j;
```

Always clean up (**deallocate**) when you are done:

```cpp
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**:

- `student student1;`
- `student student2("Kirk");`

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

```cpp
student1 is created using the default constructor:
    student();
student2 is created using another constructor:
    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`?

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.
```

WRAPPING UP
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)

Up Next

- Friday, February 9
  - Lab 5 – TBD
  - APT 3 Due
  - Project 2 – Postfix Calculator assigned (due in 2 weeks)
- Monday, February 12
  - Midterm Review
  - Lab 5 Due
- Wednesday, February 14
  - Midterm 1 (in class)