CS 270 - Computer Organization

C Memory Operations

Dr. Stephen P. Carl

Quote of the Day

With great power there must also come -- great responsibility!

- Stan Lee

Variables and Memory Addresses

C has a powerful and dangerous way of working with memory addresses: pointer variables. A pointer variable (or just *pointer*) is declared like any other variable, but with a star (asterisk) prefix.

```
int *p;  // declares a pointer variable p
```

Note that a pointer variable can have any legal type, including user-defined types (structs), and a special type, **void.**

The type applies an interpretation to the contents of the referenced memory location.

C Memory Operators

To assign values (addresses) to pointers, we need a way to get the address of a variable in memory

The unary **&** is the "address-of" operator. Applied to a variable, it evaluates to the address where the variable is stored.

The unary ***** is the "dereference" operator. When applied to a pointer, it evaluates to the *contents* of the memory location referenced by the pointer.

The meaning of * is clear from context

C Memory Operations

Okay, we need some examples. These first are a bit contrived:

```
int *p, x = 10;

p = &x;   // p now contains the address-of x

printf("%d", p);   // prints the memory location

printf("%d", *p);  // prints 10

printf("%x %d", p, *p);
```

The output when I run this is **0xA0001234 10**

SEWANEE
The University of the South

Visualizing Memory

SEWANEE
The University of the South

Initializing Pointers

An uninitialized pointer is a system crash waiting to happen. There are three possibilities:

- Use the static address of an existing variable, as in the previous slide
- Use the result of *dynamically allocating* memory, using **malloc**
- Temporarily mark the pointer as *undefined,* using **NULL**

SEWANEE
The University of the South

Initialization Examples

SEWANEE
The University of the South

Dereferencing

Earlier I said that pointers were dangerous. C already makes the programmer responsible for checking that array bounds are not exceeded. With the * operator, we now have to make sure we can account for our memory addresses. What can happen?

Dereferencing a legal memory address returns the value stored there

Dereferencing a NULL pointer causes a run-time error called a *segmentation fault*

Dereferencing an uninitialized pointer causes a random error ranging from internal data corruption to segmentation fault

In the last case, segmentation fault is your best-case scenario!

SEWANEE
The University of the South

Dereferencing Examples

SEWANEE
The University of the South

Malloc

Dynamic memory allocation occurs when the program requires new memory to store objects created at runtime. In Java, this happens when you use **`new`**

In C, we use **malloc** which takes a size in bytes and if that much space is available, returns the *base address* of the requested region.

The return type for **malloc** is the untyped address **`void*`** which is not usually what we want, so a *cast* is required to match the type of the target pointer.

Also, an operator **`sizeof`** is provided so we don't have to remember object sizes in bytes.

SEWANEE
The University of the South

Malloc Examples

Some examples:

```
double *pd;  // single double object
pd = (double *) malloc(sizeof(double));
double *double_array;  // 100 double objects
double_array = (double *) malloc(100 * sizeof(double));
struct Packet *p;
p = (struct Packet *) malloc(sizeof(struct Packet));
```

SEWANEE
The University of the South

Malloc Counterexamples

Oops, I failed. To save space (and time) books and tutorials often leave off the check that the call to **malloc** succeeds. If there is not enough space to satisfy the request, **malloc** returns **NULL.**

```
pd = (double *) malloc(sizeof(double));
if (pd == NULL) { /* handle out-of-memory error */ }
double_array = (double *) malloc(100 * sizeof(double));
if (double_array == NULL) {
   // handle the out-of-memory error
}
```

Similarly for initializing `"struct Packet *p"`

SEWANEE
The University of the South

Freeing allocated space

Unlike Java, we have to explicitly tell the system when we are done using allocated memory.

C has no built-in garbage collector

For this reason, every `malloc` must have a matching `free`

```
free(double_array);  // no longer need those 100 double objects
```

Forgetting to free an allocated region of memory causes a situation called a *memory leak* where allocated memory is no longer accessible. It's like hoarding unusable memory, and is woefully common in production C/C++ programs like Word or Firefox.

SEWANEE
The University of the South

Freeing allocated space

Making sure your own mallocs are freed is relatively easy in a small program. However, it can be challenging in the following instances:

- Memory allocated in one file is used (and must be freed) in another
- A call to a C system (or third-party) library function allocates memory and returns the address

These situations are called *inter-module dependencies* and violate the SE principle of *low coupling* **at the language level!**

Tracking inter-module memory management dependencies is a real pain and the bane of a professional C/C++ programmer's existence.

SEWANEE
The University of the South