So far, system calls have had easy syntax

Integer, character string, and structure arguments.

But this is not always true.

Today, we begin to explore the C types necessary to invoke system calls. These include:
  - pointer and pointer casts.
  - const pointers
  - function pointers
  - typedefs
void *malloc(size_t size) ;
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg
);

E.g., consider these library functions:
For a pointers \( p, q \) and an item \( a \):
- \( p = q \) means make \( p \) point where \( q \) points.
- \( p + 1 \) is a pointer one cell above \( p \) (treating memory as an array).
- \( p - 1 \) is a pointer to one cell below \( p \) (ditto).
- \( *p \) refers to the item pointed to by \( p \).
- \&\( a \) is the address of an item \( a \).
- \&\&\( p \) \( \equiv \) \( p \) for a pointer \( p \).
- \&\&\( a \) \( \equiv \) \( a \) for an object \( a \).
- \( a[i] \) \( \equiv \) \( *(a+i) \)
- \( a->b \) \( \equiv \) \( (*a).b \)
A **box** refers to a **memory location**.

An **unboxed item** refers to a **symbol** that does not correspond to a memory location.

Example: `int a[5];`

Example: `int *p; p=a;`
Problems and opportunities with Pointers

Monday, September 15, 2014    3:39 PM

Pointers and arrays
  Why is there no bounds checking?
Pointer casts
  Why do we want to be able to change the type of memory?
Controlling whether to write to a pointer
  What is const?
  How are strings implemented in C?
Special pointer types
  What is void * and what can I do with it?
Pointers and ambiguity
  How does one really read a linux man page?
Caveats: for p, q pointers; i,j integers:

\[ p + i = \text{pointer}, \]
\[ p - i = \text{pointer}, \]
\[ p - q = \text{integer} \text{ (distance between } p \text{ and } q, \text{ if } p, q \text{ are same type)}, \]
\[ p + q \text{ is meaningless} \]

One can set a pointer to anything provided that one doesn't access it:

\[ p = \text{NULL}; \quad \text{// } \#\text{define NULL 0} \]
\[ *p = 2; \quad \text{// guaranteed seg fault.} \]
A **pointer cast** determines what kind of data a pointer refers to: If p is a pointer,

- (int *)p is another pointer that points to an array of ints.
- (struct foo *)p is another pointer that points to an array of things of type struct foo.

Pointer casts are a **semantic operation**
- The pointer doesn't change.
- It is simply thought of in a different way.

**Operations on the pointer change meanings!**

A pointer cast
- Redefines pointer reference (to a potentially larger or smaller thing).
- Redefines pointer addition (to increment or decrement by the size of the thing).
Pointer operations are only defined if objects are appropriately aligned. This means that the address is some multiple of a power of 2. Most processors will generate a bus error for a reference to an unaligned pointer.

long x;
short *p;
p = (short *)((char *)x)+1);
*p=2; // guaranteed bus error!

// I violated alignment constraints.
// p is an odd address, must be even.

// ANSI Standard layout: every primitive type
// is aligned on a multiple of its length.
There are many cases in which C defines something by function.

E.g., at runtime,
    There are no type descriptors.
    There is no way to determine the type of a particular memory element. Except to look at how it's used.

If it's used as an int, it's probably an int. If it's used as a pointer, it's probably a pointer.

These are attributes of the machine code.

There is no procedure for determining whether a piece of memory p is a pointer, other than how it is used (by evaluating *p)
Why pointer casts are important

The operating system, unlike a user program, has to deal with data whose type it does not know.

Often, a block of memory can have multiple meanings.

Basic strategy: treat unknown types as `void`, or `void *`, and then later, cast them to types they are.

More generally,

Whenever it is necessary to use a block of memory in a new way, can cast a pointer to it to the appropriate type.

This is the root of how dynamic memory allocation is done.

In C++, \( p = \text{new } x \); creates a new instance of the object \( x \)

In C, there is no "new", instead, \( p = (\text{int }*)\text{malloc(sizeof(int)*5)} \) creates an array of 5 ints.

`malloc` returns a pointer of type "`void *`": type unknown.
Two things about malloc

Monday, September 15, 2014  6:09 PM

It is smart enough to align memory elements on a boundary divisible by the size of the largest primitive type.

-> you can cast it to any type
+ ANSI standard layout

-> you can cast it to any struct type.
void * is C's main information hiding mechanism
    Tells you where something is.
    Doesn't tell you how to write or read it.
Example:
    void * malloc (unsigned int size);

More advanced example: driver information tables.
    Referenced as void *.
    Cast to appropriate types inside driver.
    OS doesn't know the difference.

struct driver_table {
    ...
    void * data;
    ...
} table[100];

Inside driver functions, e.g., write:
    struct device_data {
        char buffer[4096];
        ...
    };
    ...
    struct device_data * d
    = (struct device_data *) table[30].data;
    // ... do things with d ...
Dynamic memory allocation: allows an application to get new memory to use for its own purposes.

- Global variable
- Arbitrary size
- Arbitrary lifetime

`malloc` grants access, `free` revokes it.
### C versus C++

<table>
<thead>
<tr>
<th>C</th>
<th>C++</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int *p=(int *)malloc(10 *sizeof(int));</code></td>
<td><code>int *p=new int[10];</code></td>
<td><code>int p[] = new int[10];</code></td>
</tr>
<tr>
<td><code>free(p);</code></td>
<td><code>delete p;</code></td>
<td>No equivalent</td>
</tr>
</tbody>
</table>

**new calls malloc!**  
**delete calls free!**
Malloc caveats

Data returned is a **pointer to storage**.
Data is **uninitialized**; contains seemingly **random values** (impractical to predict in advance)
No constructors; **you** manually initialize data.
Everything is done by changing type of data **after** malloc (by pointer casts)

Why no initialization?
Many data structures don't need to be initialized (Example: large hash tables)
Without initialization, malloc executes in $O(1)$ in most cases (constant time)
With initialization, it executes in $O(n)$ ($n=$number of bytes)
Why bother to initialize if you don't always need it?

Common practice: encapsulate initialization

```c
struct foo { int i; double d; }
struct foo *foo_create() {
    struct foo *m =
        (struct foo *)
        malloc(sizeof(struct foo));
    m->i=0; m->d=0;
    return m;
}
```
return m;
}

void foo_delete(struct foo *f) { free(f); }
main() {
    struct foo *f = foo_create();
    ...
    foo_delete(f);
}
Must explicitly free() storage so it can be reused. **free(p) does not change p!**

Subsequent allocations can reuse memory. This can cause a **pointer alias** and havoc can ensue.

Typical horror story:

```c
int *p=(int *)malloc(sizeof(int));
...
free(p);
int *q=(int *)malloc(sizeof(int)); // reuse p's data!
*p=5; // change *q!
```

Typically, one should free(p) every time a pointer is reset. Thus, one could write:

```c
*p=5; free(p); // always set to NULL
```
When an OS gives your process memory to use, it does not know the types you assign to it.

That information is not just unknown, but also unknowable.
C is not reflective (unlike Java)!
const

If you have a pointer to a thing, you can write to it. The const declaration allows you to decide you aren't going to write to a pointer.

Example

```c
void foo(const int *p) { printf("%d\n", *p); }
```

Rules for const

Non-const implicitly casts to const. E.g.,
```
int *p; const int *q; q=p; // works
```

const does not cast to non-const. E.g.,
```
const int *p; int *q; q=p; // compiler error
```

position of const in the declaration describes exactly what is constant and what remains mutable:
```
const int *p; // *p is const
int * const p; // p is const, *p remains mutable!
const int * const p; // both are const
```

To be nerdy about this:
```
const int * const * const r;
```

means that
```
r is const
* r is const
** r is const
```
Note the pattern: the thing immediately after the const is the thing that is constant.

The point: many subroutines have const arguments that are automagically cast from non-const to const to guarantee that the subroutine won't write on the arguments.

But if I want to be really antisocial, I can write:

```c
int a;
int *p = &a;
const int *q = p;
*(int *)q = 7; // violate the agreement!
```
Const is a **social** contract

The function that declares and argument as const promises not to touch it.

By convention, a function that takes a non-const argument does touch it.

It is considered sloppy to write a function that doesn't touch a non-const argument.
#include <string.h>
char *strcpy(char *dest, const char *src);

says that:
we are going to change dest
we are not going to change src

How this is typically implemented:

char *strcpy(register char *dest, register const char *src)
{
    char *ret = dest;
    while (*src) *dest++=*src++;
    *dest='\0';
    return ret;
}

Explanation:
register char *dest: store this in a register, not a memory location
while (*src): until you get to a '"0'
*dest++=*src++: copy from src to dest, then increment
The Linux manual pages are actually rather difficult to read. The main problem is **ambiguity**: the declarations of functions do not fully describe their requirements.

Consider, e.g., the following declaration:
```
char *strcpy(char *, const char *);
```
What exactly does that mean?

**A pointer argument p is ambiguous**, and always has two regular meanings:
- A pointer to a single thing *p
- A pointer to an array of things p[i]
  (other more strange interpretations are also possible, but uncommon)

For this example,
- We don't know whether the things are arrays.
- We don't know whether the things are strings (a particular kind of array).
Ambiguity

- When you get a pointer p you do not know:
  - whether it points to an array or a single thing?
  - Equivalently, whether p[i] makes sense, or just *p?
  - appropriate range for i for p[i] to make sense?
  - Equivalently, what length of array it points to.

Example:

```c
int *foo(int *a) { return a+3; }
main() {
  int b[5];
  int *p=foo(b);
  p[-3] = 2; /* ok! */
  p[-5] = 4; /* not ok! */
}
```
int printf(const char *fmt, ...);
Q: Is fmt a pointer to a char or a pointer to a character array?
A: Turns out it is an array.
Q: How large is fmt?
A: There's a \0 at the end.
Q: what does const mean?
A: agrees not to change *fmt (or, e.g., fmt[i]).

printf("%s","hi there");
printf("hi there");

const char *t = "hello";
printf("%s", t);
printf(t);
A string constant is a pointer to pre-allocated storage
"hello" is
    const char *
not writable
readable as an array
    "hello"[0] is 'h'
terminated by a null character
    "hello"[5] is '\0' = 0.
stored in the text segment.
protected by hardware memory protection.
try to write => bus error
End of lecture on 09/25/2017
Monday, September 25, 2017  5:37 PM
Pointer ambiguities arise
  In manual pages.
  In declarations.
  In header files.
int **p; // can mean
  a pointer to nothing
  a pointer to a pointer to a thing.
  a pointer to an array of pointers to things.
  a pointer to a pointer to an array of things.
  a pointer to an array of pointers to arrays of things.
int main(int argc, char **argv, char **envp)
{
}

Declaration of main:

Thursday, September 09, 2004       6:16 PM
What is the structure of envp?