A Pointer is a Memory Address with a Type

In C, a pointer is a memory address. Let’s say that we have an address A. As you know, we tell the compiler
- the type of data
- that we have stored
- or want to store at address A.

Examples include...
- int* A;
- float* A;
- player_t* A;

A Function’s Address is a Function Pointer

Given a function
```c
int32_t func (double d, char* s);
```
the expression &func evaluates to the function’s (starting) address.*
The type of &func is
```
int32_t (*)(double, char*)
```
a pointer to a function that takes a double and a char* and returns an int32_t.

*For historical reasons, the expression func produces the same value, but today with type int32_t (double, char*).
I/O Channel Behavior Depends on Type of Channel

But what’s the point?
Er.*
Remember that file descriptors are
◦ indices into an array of I/O channels
◦ controlled by the OS?
The behavior of each I/O channel
◦ depends on the type of the channel
◦ (keyboard, display, file, and so forth).

Behavior is Implemented with Function Pointers

In the I/O channel array,
◦ each array element
◦ includes several function pointers.
The functions define a channel’s behavior.
To implement a new channel type,
◦ implement a function for each operation
  (read, write, and so forth), then
◦ use the addresses of the new functions to form
  the array element for a channel of the new type.

Can Also Specialize Data Structures or Other Functions

Function pointers can be used to specialize
behavior in other ways.
Behavior of operations on a structure:
◦ different subtypes use different functions;
◦ called ‘virtual’ functions in C++ and Java
Behavior of a function X:
◦ pointer to Y passed as argument;
◦ X calls back to Y to execute an operation;
◦ X is said to use a callback to implement
  the operation.

Let’s Generalize Insertion Sort to Operate on Any Type

Do you remember
◦ that I said we could copy the code
◦ for the insertion sort on integers
◦ to create an insertion sort on strings?
Now let’s
◦ generalize the code to
◦ sort an array of “things.”
  We’ll use function pointers.
We Need the Size of Each Thing

What new information will our sort need?

First, how big is a “thing?”

We’ll pass the answer

* as a number of bytes
* in a parameter.

Help Prof. Lumetta ... Again

Which horse is first?

But to sort “things,” we need to have an order!

The solution? A callback!

What Signature Do We Use for the Callback?

What is the signature of the callback function?

We need to pass it two “things,” t1 and t2.*

Let’s use void* for those.

We get back an answer:

* t1 < t2, or
* t1 ≥ t2.

*Apologies to Dr. Theodor Geisel.

Function Signature for Our isort Function

int32_t isort

(void* base,
 int32_t n_elts,
 size_t size,
 int32_t (*is_smaller)(void* t1, void* t2));

points to an array number of things in array
number of bytes per thing
pointer to a function comparing two things
Return Value from `isort` Indicates Success or Failure

Returns 1 on success, or 0 on failure.

```c
int32_t isort
    (void* base,
     int32_t n_elts,
     size_t size,
     int32_t (*is_smaller)
         (void* t1, void* t2));
```

Function Pointer: a Function Signature, Plus an Asterisk

Focus now on the function pointer argument:

- Function pointer's identifier preceded by `*` and surrounded by parentheses.
- `int32_t (*is_smaller) (void* t1, void* t2)`

Function Pointer Meaning Must Be Documented

**Meaning** of the function pointer argument
- **must be documented**, just like other parameters!
- `is_smaller`
  - compares two array elements, `t1` and `t2`
  - and returns
    - non-zero (true) if `t1 < t2`,
    - 0 (false) if `t1 ≥ t2`.

Change Variables to Support “Things” of Any `size`

Now for code (based on the integer version):

```c
char* array = base;
void* current;
int32_t sorted;
int32_t index;
```
Allocate space for one "thing."

No memory? Fail.

Free the allocated space.

Return success!

Lack of Memory is the Only Reason for Failure

if (NULL == (current = malloc (size))) {
    return 0;  // Allocate space for one "thing."
}

No memory? Fail.

// do the insertion sort here
free (current);  // Free the allocated space.
return 1;  // Return success!

Loop structure is identical to the integer version of the sort.

Invariant: when loop finishes with sorted equal to N, first N entries of array are sorted.

Copy a “Thing” into and Out of current

memcpy (current, array + (sorted - 1) * size, size);

Copy one "thing" into current.

// inner loop finds correct // position for current (index)
memcpy (array + index * size, current, size);

Copy current to correct position.

Use memcpy to Copy Bytes

Need a way to copy size bytes...
In string.h (C’s string library), we find:

void* memcpy (void* dest, const void* src, size_t N);

which copies N bytes from src to dest assuming that the regions do not overlap and returns dest.
Loop structure is again identical to the integer version of the sort.

```c
for (index = sorted - 1; 0 < index; index--) {
    // inner loop body performs
    // comparison and may copy
    // one thing to adjacent place
}
```

Use `is_smaller` to compare `current` with the thing in array element `index - 1`.

```c
if ((*is_smaller) (current, array + (index - 1) * size)) {
    memcpy (array + index * size, array + (index - 1) * size, size);
} else {
    break;
}
```

Keep shifting things until `current`'s place is found.

```c
if ((*is_smaller) (current, array + (index - 1) * size)) {
    memcpy (array + index * size, array + (index - 1) * size, size);
} else {
    break;
}
```

Otherwise, we found the right place!

```c
if ((*is_smaller) (current, array + (index - 1) * size)) {
    memcpy (array + index * size, array + (index - 1) * size, size);
} else {
    break;
}
```
Write a Version of \texttt{is\_smaller} or a Specific Type

How do we use the generic sort?
Start by writing a comparison function.

Function signature must match specification for \texttt{is\_smaller}:

\begin{verbatim}
int32_t (*is\_smaller)(void* t1, void* t2)
\end{verbatim}

Example: Comparison Function for \texttt{double}

\begin{verbatim}
int32_t double\_is\_smaller(void* t1, void* t2)
{
  double\* double1 = t1;
  double\* double2 = t2;
  return ((*double1) < (*double2));
}
\end{verbatim}

Comparison Function Enables Sorting \texttt{doubles}

And now we can sort doubles!

\begin{verbatim}
double d\_array[8] = {
  5.0, -17, 39, 33,
  -222, 109, 3.1415, 60
};
isort (d\_array, 8,
  sizeof (d\_array[0]),
  &double\_is\_smaller);
\end{verbatim}

Code is on the Class Web Page

That's all!

Again, all code is on the web page.