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 Pointer Can Pointer to a Function

What else could be stored at address $A$?

A function!

There’s nothing special
- about the bits used
- to name the address
- of the first instruction
- in a function.

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

```c
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*)`.

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;`
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).

*Sorry.

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,” \( t_1 \) and \( t_2 \).

Let’s use \texttt{void*} for those.

We get back an answer:
- \( t_1 < t_2 \), or
- \( t_1 == t_2 \), or
- \( t_1 > t_2 \).

*Apologies to Dr. Theodor Geisel.

Function Signature for Our \texttt{isort} Function

\[
\text{\texttt{isort}} \quad (\text{\texttt{void* base, int32_t n_elts, size_t size, int32_t (*is_smaller)(\text{\texttt{void* t1, void* t2}}))});
\]

- \texttt{base} points to an array
- with \texttt{n_elts} elements,
- each of \texttt{size} bytes.
- \texttt{is_smaller} compares two array elements, \( t_1 \) and \( t_2 \), returning -1 if \( t_1 < t_2 \), 0 if \( t_1 == t_2 \), and 1 if \( t_1 > t_2 \).
- \texttt{isort} returns 1 on success, or 0 on failure.