Quick Reference - Function Pointers

How to Survive in EECS 280

An Example

We often think of data being stored in a computer's memory while a program is running, but it's equally true the program *itself* must be stored somewhere as well. Specifically, each function you wrote is stored at some particular address in memory. Let's look at an example. First, we'll define a few simple functions.

```
void printGreeting(){
  cout << "hey ";
}

void printFarewell(){
  cout << "goodbye" << endl;
}
```

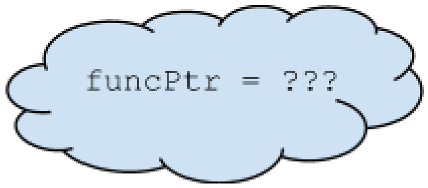
```
void printYes(){
  cout << "sure ";
}

void printNo(){
  cout << "nah ";
}
```

The standard way to call one of the functions is just to use that function's name.

```
printGreeting();
```

You could think of this as hard-coding the address of `printGreeting` for the function call. But we could also use a **function pointer**, which is basically a variable that stores the address of a function. This is more flexible because we don't necessarily need to decide what function it is when we write the code.

```
void (*funcPtr)();   // funcPtr is a
                     // pointer
                     // to a function
                     // that takes no parameters
                     // and returns nothing.
```



```
funcPtr();           // call the function at the address stored
                     // in funcPtr, whatever that ends up being
```

Why is this useful? There are a number of reasons, but one of the most common benefits is they can be used to write **higher-order functions** that can take other functions as parameters or return them as a result. Consider this example, which calls another function several times:

```
void repeat(void (*func)(), int n){

  while(--n >= 0){  // loop n times.

    func();         // call the function at address passed in
                    // through func.
  }
}
```

Now we could use the `repeat` function to write code like the following.

```
int main(){
  repeat(printNo, 4);
  repeat(printGreeting, 3);
  repeat(printFarewell, 1);
}
```

Which prints the classic "`nah nah nah nah hey hey hey goodbye`".

Understanding Function Pointer Type Declarations

One of the trickiest parts about function pointers is that their type declarations can be very complex. As we saw in lecture, the best way to read function pointer declarations (or any type declarations, really) is to start at the name of the variable and work your way from the inside out.

Here's an example.

```
10   double * (*func[3])(int)   // func is an
20   double * (*func[3])(int)   // array of 3
30   double * (*func[3])(int)   // pointers
40   double * (*func[3])(int)   //
50   double * (*func[3])(int)   // to functions that take an int
60   double * (*func[3])(int)   // and return a pointer
70   double * (*func[3])(int)   // to a double
```

A couple parts of this are worth noting. In order to determine the order of lines 20 and 30, we use the fact that the [] operator has higher *precedence* than the * operator. Similarly for lines 50 and 60, () has higher precedence than *. This handy table shows the precedence for all operators in C++.

http://en.cppreference.com/w/cpp/language/operator_precedence

Typedef (to the Rescue!)

It quickly becomes cumbersome to write out really long types like this when writing a program. The **`typedef`** keyword allows us to define new names for complicated types. The easiest way to master the syntax of typedefs is to realize they look just like variable declarations.

For example, recall the repeat function above. Originally, if we wanted to declare a *variable* `funcPtr` that is a pointer to a function with no parameters and that returns nothing, we write:

```
void (*funcPtr)();
```

But we want to declare a new name for this type. In the context of this example, such functions are used to print stuff, so a reasonable name might be `printer`. We would define this with:

```
typedef void(*printer)();
```

Now we can just write `printer` anywhere we would have used `void (*funcPtr)()`. For example, we can now rewrite the `repeat` function from above like this:

```
void repeat(printer p, int n){
  while(--n >= 0){  // loop n times.
    p();          // call the function at address passed in
                    // through func.
  }
}
```