# Function Pointers in C

CSE 2421

Recommended Reading: *C Primer Plus*, Chapter 14, Section on Function Pointers
This will be especially interesting to anyone who plans to complete Lab 3

## Pointers – Review

- We saw pointers to simple types before, e.g.:

```
int *ptr1;
float *ptr2;
char *ptr3;
```

- We also saw examples of pointers to pointers (these can be used as pointers to arrays, for example):

```
float **sets_ptr; /* sets_ptr is a pointer to a pointer to float*/
```

- What about the following? What does it declare?

```
int *findValue(int value, int size, const int *array);
```

## Declaration of pointer as return type

- This declares a function which returns a pointer
- It might do something such as the following:

```
/* This function is passed a value which may be an
*  element of array, and if the value is found in the
*  array, it returns a pointer to it. If the element is
 *   not found, it returns a null pointer.
*/
int *findValue(int value, int size, const int *array);
```

## How about this one?

- What do you think this declares?

```
int (*func_p)();
```

- It’s clearly different from the declaration of a function that returns a pointer, but what does it declare?
- Compare it to this declaration:

```
int *func();
```

# Function pointer declaration

- This declares **a pointer to a function** (or a function pointer)!

  ```
  int (*func_p)();
  ```
- A pointer to a function makes sense, because a function consists of code, and that code commences, or begins, at some address.
- Therefore, we can declare a pointer which can be assigned that address (but we haven't done that yet). This will cause the function pointer to point to the beginning of the code for the function (the first instruction in the function).
- Notice that the function pointer is ***a variable***; that is, it can be assigned the addresses of different functions at different points in the program.
- **The parentheses around *func_p are necessary!** Otherwise, the compiler will treat this as a declaration of *a function which returns a pointer*, and not *a function pointer*.

## Using function pointers

- Just as a pointer to anything is not useful until it points to something, a function pointer is not useful either until it actually points to some function that we wish to call.
- Example of assignment to function pointer:

```
/* Declaration of function which returns the max value in an array*/
int returnMax(int size, const int *array);
/* Declare fp, and make it point to function returnMax */
int (*fp)(int, int*) = &returnMax;
```

# Type of a function pointer

- The type of a function pointer includes the return type, and the number and types of any parameters, in a certain order.
- For example, consider the declaration of the function pointer on the preceding slide:

```
int (*fp)(int, int*) = &returnMax;
```

- This declares fp to be a pointer to a function which returns an int, and which has two parameters, the first an int, and the second an int pointer.
- Because the function returnMax returns an int, and has two parameters, the first an int, and the second an int pointer, the address of returnMax can be assigned to fp.

## Type of a function pointer cont.

- It would not be valid to assign the address of a function which has a different return type to fp.
- Neither would it be valid to assign the address of a function which has a different number of parameters, or which has two parameters, but of different types (that is, the first one not an int, or the second one not an int pointer).
- In short, the signatures must match.

# Note on assignment to function pointers

- **NOTE:** We do not have to use the address operator, &, when we assign the address of a function to a function pointer, but we can.
- Therefore, after the declaration of returnMax, the following two assignments to fp are **equivalent**:

  ```
  int returnMax(int size, const int *array);
  int (*fp)(int, int*) = returnMax;
  int (*fp)(int, int*) = &returnMax;
  ```
- This is because the compiler always converts the function name to a function pointer (an address) before using it (You can compare this to the name of a variable which is an array, which the compiler also treats as a pointer or address).

## Note on assignment to function pointers cont.

- The initialization can also be accomplished with a separate assignment statement:

```
int returnMax(int size, const int *array);
int (*fp)(int, int*);
fp = returnMax;        /*  OR   fp = &returnMax; */
```

- Notice that function returnMax must be *declared* (but not necessarily defined) before assigning its address to the function pointer.
- This is so that the compiler can do checking for type compatibility.

## How do we call a function with a function pointer?

- For the declarations above, the call would be (where int i has been previously declared):

  ```
  i = (*fp)(argument1, argument2);
  ```
- If the function to which fp points has no parameters, the call would be:

  ```
  i = (*fp)();
  ```
- If the function to which fp points has no parameters, and no return value, the call would be:

  ```
  (*fp)();
  ```

NOTE: All of these calls explicitly dereference the function pointer. We can also make the call as shown on the next slide, **without** dereferencing. Again, the parentheses are necessary!

## Function call using a function pointer without dereferencing (more usual)

- Without dereferencing fp (where int i has been previously declared):

```
i = fp(argument1, argument2);
```

- If the function to which fp points has no parameters, the call without dereferencing would be:

```
i = fp();
```

- If the function to which fp points has no parameters, and no return value, the call without dereferencing would be:

```
fp();
```

## Why use function pointers?

- Function pointers provide some extremely interesting, efficient and elegant programming techniques, though they are not very common in C programs.
- You can use them to:

  – Replace switch/if statements

  – Implement *callbacks* (used in labs 3 & 4)

  – Realize your own *run-time-binding* (we will not look at this).
- Which leads to:

  Greater flexibility and better code reuse.

## Arrays of function pointers

- C treats pointers to functions just like pointers to data; therefore, we can have an array of pointers to functions (example below).
- This offers the possibility of selecting a function using an index, and this can be used to replace an if, or a switch statement ***when the functions all pass the same number and type of parameters.***
- For example:

  Suppose we're writing a program that displays a menu of commands for the user to choose from. We can write functions that implement these commands, and then store pointers to the functions in an array.

## Replacing switch/if statements

- Suppose the following code:

```
float math_op() { /* begin function math_op */
    int option;
    float a, b;
    printf("Enter 1 to add, 2 to subtract, 3 to multiply, or 4 to divide:\n");
    scanf("%d", &option);
    printf("Enter the two operands:\n");
       scanf("%f  %f", &a, &b);
/* function code continued on next slide */
```

## Replacing switch/if statements cont.

```
    switch (option) {
    case 1: return (a + b);
            break;
    case 2: return (a – b);
            break;
    case 3: return (a * b);
            break;
    case 4: return (a / b);
            break;
    default: printf(“Invalid choice entered. Must be 1, 2, 3, or 4!\n”);
    } /* end switch */
} /* end function math_op */
```

# Using function pointers instead

- First, define four functions to perform the four operations, add, subtract, multiply and divide, each returning a float, and taking two float parameters; e.g. (only 2 functions shown; the other 2 are similar),

  ```
  float add(float a, float b) {return (a + b);}
  float subtract(float a, float b) {return (a - b);}
  ```

- Then, declare an array of four function pointers, and assign addresses of the functions:

  ```
  float (*fp[4])(float, float);
  fp[0] = add;          /* make fp[0] point to add function */
  fp[1] = subtract;               /* make fp[1] point to subtract function */
  fp[2] = multiply;     /* make fp[2] point to multiply function */
  fp[3] = divide;                 /* make fp[3] point to divide function */
  ```

## Using function pointers instead cont.

Now, rewrite function math_op as follows (Notice *a pointer* to the array of function pointers passed as a parameter to the function):

```
float math_op(float (**fp)(float, float)) {
    int option;
    float a, b;
    printf(“Enter 1 to add, 2 to subtract, 3 to multiply, or 4 to divide:\n”);
    scanf(“%d”, &option);
    printf(“Enter the two operands:\n”);
    scanf(“%f  %f”, &a, &b);
    if ( (option > 0) && (option <= 4)) {
        return ( fp[option – 1](a, b) );
    }
    else printf(“Invalid choice entered. Must be 1, 2, 3, or 4!\n”);
}
```

## Using function pointers instead cont.

```
/* #include <stdio.h> and 4 function definitions here */
int main () {
   float result;
   float (*fp[4]) (float, float) = {add, subtract, multiply, divide};
   result = math_op(fp);
   printf("The result is: %f\n", result);
   return 0;
}
```

# typedef might be your friend

```
typedef float (*binOp)(float, float);
/* #include <stdio.h> and 4 function definitions here */
int main () {
   float result;
   /* the function pointer declaration is much easier */
   binOp fp[4]= {add, subtract, multiply, divide};
   result = math_op(fp);
   printf("The result is: %f\n", result);
   return 0;
}
```