UNIX and C Programming (COMP1000)

Lecture 4a: Pointers to Functions

Updated: 19th August, 2019

Department of Computing
Curtin University

Outline

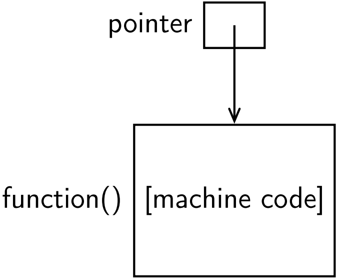
Pointers to Functions

Typedef

Using Pointers to Functions

Pointers to Functions

- Functions are stored in memory, just like variables.
- Pointers can point anywhere in memory, including to functions.
- There are special pointer types to represent this.
- These pointers can point to a function with specified parameter/return types.



Pointers to Functions — Why?

- Used to implement "callbacks":
  - You call one function, and give it a pointer to *another* function.
  - The first function calls the second, in some fashion beyond your control.
  - The second function, which you write yourself, is the "callback" function.
- Callbacks are used a lot in "Event-Driven Programming". For instance:
  - Mouse clicks (and their consequences, such as button presses).
  - Stopwatch timers.
  - Network communication.
- With callbacks, you control *what* happens, but you let something else decide *when* it should happen.

Pointers to Functions – Declaration

- To declare a pointer to a function:

  ```
  return-type (*variable-name)(parameters);
  ```

  For example:

  ```
  int (*ptr)(float x, int y);
  ```

  The parameter names are optional (and just for show):

  ```
  int (*ptr)(float, int);  /* Same as above */
  ```

- Looks a bit like a function, but this is actually a variable.
- `ptr` holds the memory address of *any* function that:
  - Takes a `float` and `int` parameters.
  - Returns an `int`.

Pointers to Functions – Assignment

- Consider this function:

```
int myFunction(float abc, int xyz) {
    return ...;
}
```

- The address-of (&) operator works on functions (as well as variables).
- So, `&myFunction` is the memory address of `myFunction` (where its machine code is stored).
- We use this to initialise pointers to functions:

```
int (*ptr)(float, int);
ptr = &myFunction;  /* ptr points to myFunction */
```

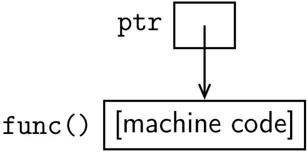
- Like all variables, we can combine declaration and initialisation:

```
int (*ptr)(float, int) = &myFunction;
```

Pointers to Functions – Usage (1)

- Pointers to functions (like all pointers) are just *values*.
- They can be copied and assigned like other pointers.
- However, dereferencing a pointer to a function gives you a function.
- Consider this:

```
int (*ptr)(float, int);
ptr = &func;
```



- The expression `*ptr` is now equivalent to `func`.
- And so `(*ptr)(...)` is now equivalent to `func(...)`.
- i.e. we can take a pointer, and call the function it points to.
  - And remember it could be *any* function (with the right parameter and return types).

Pointers to Functions – Usage (2)

To complete the example:

```
int myFunction(float abc, int xyz) {
    return ...;
}
...

/* Declare ptr as a pointer to a function. */
int (*ptr)(float, int);

/* Make ptr point to myFunction. */
ptr = &myFunction;

/* Call the function it points to. */
int result = (*ptr)(7.0, 3);
```

Pointers to Functions — Another Example

```c
void printHello(void) {
    printf("Hello world\n");
}

/* A function that takes a pointer to another
   function, and calls it n times. */
void callNTimes(int n, void (*funcPointer)(void)) {
    int i;
    for(i = 0; i < n; i++) {
        (*funcPointer)();
    }
}
...
/* Prints "Hello world\n" 10 times. */
callNTimes(10, &printHello);
```

Typedef

- The “typedef” keyword can be placed before any declaration.
- It converts the declaration into a “type declaration”.
- The name being declared instead becomes a new data type — an alias.
- You can then use that name in place of the type it was declared as.
- Normally used in header files.

Simplistic Example

```
typedef int INTEGER;
...
INTEGER num = 15;
```

Typedef — Pointer Example

```c
typedef void* MagicData;

MagicData getMagic(void);
void doMagic(MagicData magic);
```

- `MagicData` is equivalent to `void*`.
- The new name can serve as a form of documentation.
- `void*` could mean anything, but `MagicData` might indicate something specific about the data.
- It can also be a primitive form of information hiding.
- Other code *doesn't need to know* what `MagicData` really is.

Typedef — Pointers to Functions

- typedef can simplify pointers to functions.
- You only need *one* convoluted declaration (in a header file):

```
typedef int (*MyType)(float, int);
```

- MyType is now shorthand for this convoluted pointer datatype:

```
~~int (*ptr)(float, int) = &myFunction;~~
MyType ptr = &myFunction;    /* Equivalent */
```

- You can also return pointers to functions:

```
MyType function2(char a, double b) {
    return &myFunction;
}
```

  - Without typedef, the syntax for this would be very strange.

Functions as Data Types

- With pointers to functions, you treat functions as data types!
- As a result, they can *look* bizarre.
- However, they follow the *same rules* as other declarations.
  - (Those rules may be more subtle than you realised!)

Consider this ordinary function declaration:

```
int myFunction(float, int);
```

- **Rule 1:** all declarations consist of a name and a type [1].
- Here, the type is "`int...(float,int)`" (not just "`int`").
- "`myFunction`" has the *type* "`int...(float,int)`".
- Part of the type goes on the left, and part goes on the right!

[1] Except for parameters, where the name can be omitted in a forward declaration.

Pointers to Functions – Declarations (1)

- Say we want a pointer to "`int...(float,int)`" (i.e. a pointer to a function with those parameters and return type).
- Where does the * go?
- **Rule 2:** the * goes on the left of the name.
- Where does the name go?
- In the middle! (Since part of the type goes on the left, and part on the right.)

Pointers to Functions – Declarations (2)

Almost correct (but not quite)

```
int* myPointer(float,int);
```

- Everything is (basically) in the right place; the name is surrounded by the type.
- However, this is a *function* returning a pointer, not a *pointer* to a function.
- Why?
- "(...)" (the parameter list) has a higher *precedence* than "∗".
- **Rule 3:** If there's "(...)" immediately to the right, you have a function.

Pointers to Functions – Declarations (3)

Correct

Rule 4: Brackets override operator precedence.

```
int (*myPointer)(float,int);
```

This declares a *variable*, pointing to a function that:

- imports a `float` and an `int`; and
- returns an `int`.

This declaration simply obeys the rules of C that you already know.

Returning Pointers to Functions (1)

- Functions can return any data type, including pointers to other functions.
- What would the declaration look like?
- Normally, a return type goes on the left. . .
- . . . but pointers to functions have separate parts on the left and right.
- We also need *two* parameter lists!
    - One for the function we're declaring, and
    - One for the pointer to a function it returns.
- **Rule 5:** Remember all the other rules.

Returning Pointers to Functions (2)

1. First, write the function *without* a return type:

```
      myFunction(char a, double b)
```

Returning Pointers to Functions (2)

1. First, write the function *without* a return type:

```
      myFunction(char a, double b)
```

2. It returns a pointer, so add a ∗ on the left (Rule 2):

```
     *myFunction(char a, double b)
```

Returning Pointers to Functions (2)

1. First, write the function *without* a return type:

```
  myFunction(char a, double b)
```

2. It returns a pointer, so add a * on the left (Rule 2):

```
 *myFunction(char a, double b)
```

3. Add brackets to keep it that way (Rule 4):

```
(*myFunction(char a, double b))
```

Returning Pointers to Functions (2)

1. First, write the function *without* a return type:

```
myFunction(char a, double b)
```

2. It returns a pointer, so add a ∗ on the left (Rule 2):

```
*myFunction(char a, double b)
```

3. Add brackets to keep it that way (Rule 4):

```
(*myFunction(char a, double b))
```

4. Add the second parameter list, turning the returned pointer into a pointer to a function (Rule 3):

```
(*myFunction(char a, double b))(float,int)
```

Returning Pointers to Functions (2)

1. First, write the function *without* a return type:

```
      myFunction(char a, double b)
```

2. It returns a pointer, so add a ∗ on the left (Rule 2):

```
     *myFunction(char a, double b)
```

3. Add brackets to keep it that way (Rule 4):

```
    (*myFunction(char a, double b))
```

4. Add the second parameter list, turning the returned pointer into a pointer to a function (Rule 3):

```
    (*myFunction(char a, double b))(float,int)
```

5. Add the return type for the returned pointer to a function:

```
int (*myFunction(char a, double b))(float,int)
```

Returning Pointers to Functions (3)

- Compare myFunction to myPointer (declared earlier):

```
int (*myPointer)(float,int)
```

```
int (*myFunction(char a, double b))(float,int);
```

(The type of myPointer and the return type of myFunction are in red.)

- See the similarities and differences?
- myPointer *is* a pointer to a function.
- myFunction *returns* a pointer to a function.
- The brackets after the name make the difference between a variable and a function (rule 3).

Returning Pointers to Functions — Example

```
int simpleFunction(float x, int y) {
    return 10;
}

int (*myFunction(char a, double b))(float,int) {
    return &simpleFunction;
}


...
int (*myPointer)(float,int);
int result;

myPointer = myFunction(’A’, 2.5);
result = (*myPointer)(7.0, 3);
```