Dynamic Data Structures with C

Terry Marris October 2010

1 Pointers to Functions

A pointer to a function is a variable whose value is the address of a function. We see how to declare a pointer to a function, how to pass functions as parameters to other functions, how to store pointers to functions in arrays and structures, and how to return pointers to functions.

Why bother? Well, we will need to write data structures that stores data chosen by the user and uses functions such as *equal()* and *clone()* written by the user, and we have no idea what the user is going to choose. Who is the user? Another programmer who is using our data structures to implement their applications.

1.1 Declaration and Call

We define a pointer to a function variable. The function returns an *int* and has two *int* parameters. We initialise the pointer with *NULL*.

```
int (*ptrToFunction)(int, int) = NULL;
```

We could assign any function that returns an *int* and has two *int* parameters to our pointer. We shall assign the function *add()* to *ptrToFunction*.

```
  int (*ptrToFunction)(int, int) = add;
```

add() returns the result of adding two integers.

```
int add(int a, int b)
{
  return a + b;
}
```

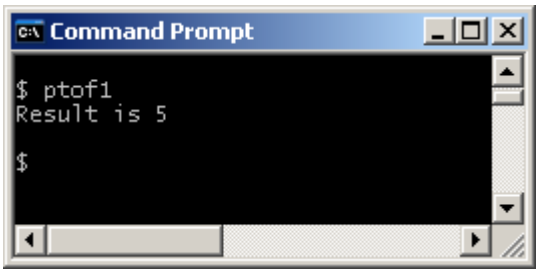
To call the pointer to function we can write

```
(*ptrToFunction)(2, 3)
```

where *2*, *3* are its arguments. Of course, you can choose any two integer values you like (provided their sum does not exceed *INT_MAX)*.

Here is the entire program and its run.

```
/* ptof1.c - declaration and call */
#include <stdio.h>

int add(int a, int b)
{
  return a + b;
}

int main()
{
  int (*ptrToFunction)(int, int) = add;
  printf("Result is %d\n", (*ptrToFunction)(2, 3));
  return 0;
}
```



1.2 Arguments and Parameters

We pass a function to another function by using pointers.

As before, our pointer to a function contains the address of *add()*.

```
int add(int a, int b)
{
  return a + b;
}

int main()
{
  int (*ptrToFunction)(int, int) = add;
  ...
```

sum() has the pointer to function as a parameter.

```
int sum(int (*ptrToFunction)(int, int),...
```

The parameters *x* and *y* provide the arguments to the pointer to function.

```
int sum(int (*ptrToFunction)(int, int), int x, int y)
{
  return (*ptrToFunction)(x, y);
}
```

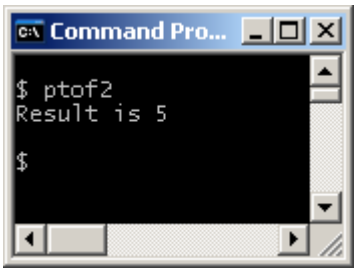
To call *sum()* and pass the pointer to function argument we write

```
sum((int (*)(int, int))(*ptrToFunction), 2, 3)
```

Notice the cast operator *(int (*)(int, int)*. This says that we have a pointer to a function that returns an *int* and has two *int* parameters.

The program, and its run, are shown below.

```
/* ptof2.c - parameters and arguments */
#include <stdio.h>

int add(int a, int b)
{
  return a + b;
}

int sum(int (*ptrToFunction)(int, int), int x, int y)
{
  return (*ptrToFunction)(x, y);
}

int main()
{
  int (*ptrToFunction)(int, int) = add;
  printf("Result is %d\n", sum(
        (int (*)(int, int))(*ptrToFunction), 2, 3));
  return 0;
}
```



Can the verbose syntax be simplified? We could define a pointer to function type.

```
typedef int (*PtrToFunction)(int, int);
```

Then *sum()* becomes

```
int sum(PtrToFunction ptrToFunction, int x, int y)
{
  return (*ptrToFunction)(x, y);
}
```

The declaration and initialisation of a pointer to function variable is

```
PtrToFunction ptrToFunction = add;
```

But in passing the pointer to function argument to *sum()*, the cast operator is still required.

```
sum((int (*)(int, int))(*ptrToFunction), (2, 3))
```

Here is the entire program.

```
/* ptof21.c - parameters and arguments */
#include <stdio.h>

typedef int (*PtrToFunction)(int, int);

int add(int a, int b)
{
  return a + b;
}

int sum(PtrToFunction ptrToFunction, int x, int y)
{
  return (*ptrToFunction)(x, y);
}

int main()
{
  PtrToFunction ptrToFunction = add;
  printf("Result is %d\n", sum(
        (int (*)(int, int))(*ptrToFunction), 2, 3));
  return 0;
}
```

1.3 Return Type

We see how to return a pointer to a function.

First create the type *PtrToFunction* that returns an *int* and has two *int* parameters.

```
typedef int(*PtrToFunction)(int, int);
```

Then we can write

```
PtrToFunction operation(const char op)
{
  if (op == '+')
    return add;
  else
    return NULL;
}
```

If *op* equals + we return *add()*, the function to return the sum of two integers.

We declare the pointer to function variable and initialise it with *NULL*.

```
PtrToFunction ptrToFunction = NULL;
```

Then we assign to it the value returned by a call to *operation()* with '+' as its argument.

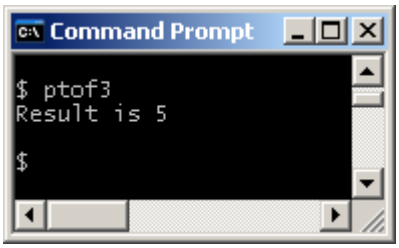
```
ptrToFunction = operation('+');
```

We could combine the declaration and assignment

```
PtrToFunction ptrToFunction = operation('+');
```

The entire program and its run are shown below.

```
/* ptof3.c - return type */

#include <stdio.h>
#include <stddef.h>

typedef int(*PtrToFunction)(int, int);

int add(int a, int b)
{
  return a + b;
}

PtrToFunction operation(const char op)
{
  if (op == '+')
    return add;
  else
    return NULL;
}

int main()
{
  PtrToFunction ptrToFunction = operation('+');
  printf("Result is %d\n", (*ptrToFunction)(2, 3));
  return 0;
}
```



1.4 Arrays of Pointers

We see how to create an array of pointers to functions.

We have two functions, *add()* and *multiply()*.

```
int add(int a, int b)
{
  return a + b;
}

int multiply(int a, int b)
{
  return a * b;
}
```

We define the *PtrToFunction* type.

```
typedef int (*PtrToFunction)(int, int);
```

And declare an array of type *PtrToFunction*.

```
PtrToFunction array[3] = { NULL };
```

The three elements of the array are initialised with *NULL*.

We assign the two functions to the array elements.

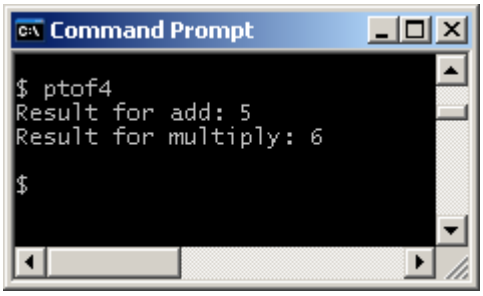
```
array[0] = add;
array[1] = multiply;
```

Then to exercise the functions we can write

```
(*array[0])(2, 3));
(*array[1])(2, 3));
```

Here is the entire program and its run.

```
/* ptof4.c - array of pointers */

#include <stdio.h>
#include <stddef.h>

typedef int (*PtrToFunction)(int, int);

int add(int a, int b)
{
  return a + b;
}

int multiply(int a, int b)
{
  return a * b;
}

int main()
{
  PtrToFunction array[3] = { NULL };
  array[0] = add;
  array[1] = multiply;

  printf("Result for add: %d\n", (*array[0])(2, 3));
  printf("Result for multiply: %d\n", (*array[1])(2, 3));
  return 0;
}
```



1.5 Structure Members

We define our pointer to structure type.

```
typedef int (*PtrToFunction)(int, int);
```

Then define a structure with just one member of type *PtrToFunction*.

```
typedef struct structure {
  PtrToFunction ptrToFunction;
} *PtrToStructure, Structure;
```

We declare a variable of type *Structure* and initialise its member with *add()*.

```
  Structure s;
  s.ptrToFunction = add;
```

To call *add()* we can write

```
s.ptrToFunction(2, 3)
```

Notice that the syntax here is straightforward. We might have expected to write *s.(*ptrToFunction)(2, 3)*, but my compiler (GNU CC) signals an error.

To declare a variable of type **PtrToStructure* and to assign *add()* to its member we can write

```
PtrToStructure ps = malloc(sizeof(struct structure));
ps->ptrToFunction = add;
```

Again, notice the straightforward syntax.

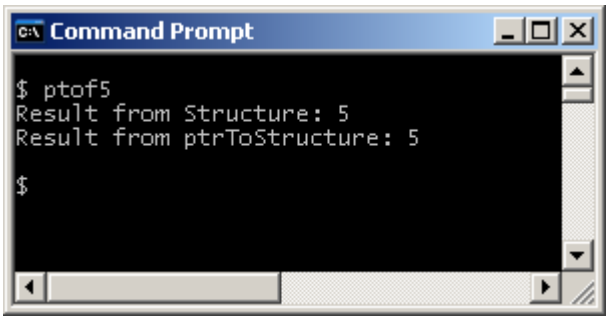
To make a call to *ptrToFunction* we can write

```
ps->ptrToFunction(2, 3)
```

Here is the entire program and its run.

```
/* ptof5.c - structure members */

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>

typedef int (*PtrToFunction)(int, int);

typedef struct structure {
  PtrToFunction ptrToFunction;
} *PtrToStructure, Structure;

int add(int a, int b)
{
  return a + b;
}
```

```
int main()
{
  Structure s;
  s.ptrToFunction = add;
  printf("Result from Structure: %d\n", s.ptrToFunction(2, 3));

  PtrToStructure ps = malloc(sizeof(struct structure));
  ps->ptrToFunction = add;
  printf("Result from ptrToStructure: %d\n", ps->ptrToFunction(2, 3));

  return 0;
}
```



Exercise 1.1

1. Write a function, *compare()*, that has two char parameters and returns zero if they are identical, one otherwise. Declare a pointer to this function and show how the pointer may be used as:

    a. a parameter to a function

    b. a structure member

Bibliography

Kernighan B and Ritchie D *The C Programming Language* Prentice Hall 1988 pp 118, 147
Haendel L *The Function Pointer Tutorials www.newty.de* accessed Aug 2010