Function Pointers

Function Pointers

Functions are treated slightly differently by the compiler:

- Functions are basically blocks of code (instructions) in memory.
- A function's name is the same as it's address.
- To call a function, you simply use the function call operator after the function's name: ()
- You don't need to dereference (*) or take the address of (&) a function. The compiler always converts the function name to an address whenever it is used.

```
int f(void)
{
  return 255;
}

int main(void)
{
  printf("%p, %p, %p, %p\n", f, *f, &f, f());
  return 0;
}
```

Output:

```
00401028, 00401028, 00401028, 000000FF
```

Since functions are very similar to pointers, we can assign them to other pointers. Specifically, we can assign them to function pointers (variables which can point to functions.)

```
int f(void)
{
  return 255;
}

int main(void)
{
  int (*pf)(void);
  int i;

  pf = f;   /* Ok */
  pf = &f;  /* Ok */
  pf = *f;  /* Ok */
  pf = f(); /* Error: 'int (*)(void)' differs in levels of indirection from 'int' */
  i = f();  /* Ok */
  f = pf;   /* Error: '=' : left operand must be l-value */

  printf("%p, %p, %p, %p\n", f, *f, &f, f());
  printf("%p, %p, %p, %p\n", pf, *pf, &pf, pf());

  return 0;
}
```

Output:

```
0040102D, 0040102D, 0040102D, 000000FF
0040102D, 0040102D, 0012FF7C, 000000FF
```

Calling the function *f* can be accomplished in different ways:

```
int f(void)
{
  return 255;
}

int main(void)
{
  int value;
  int (*pf)(void) = f; /* Initialize pf with address of f */

    /* All statements are equivalent                             */
  value = f();      /* call function "normally"                 */
  value = pf();     /* call function through pointer to function */
  value = (*pf)();  /* dereference pointer to function          */

  return 0;
}
```

Type compatiblity is important:

```
/* a function that takes nothing and returns an int */
```

```c
int f(void)
{
  return 255;
}

/* a function that takes nothing and returns an int */
int g(void)
{
  return 0;
}

/* a function that takes nothing and returns a double */
double h(void)
{
  return 0.5;
}

int main(void)
{
  int value;
  int (*pf)(void);      /* declare function pointer */
  double (*ph)(void);   /* declare function pointer */

  pf = f;   /* Ok, pf and f are same type */
  pf = g;   /* Ok, pf and g are same type */
  pf = h;   /* Error: incompatible types  */
  ph = h;   /* Ok, ph and h are same type */

  pf = (int (*)(void)) h;   /* Only if you know what you're doing! (Unlikely in this case.) */
  value = pf();             /* Value is -858993460, not 0. */

  return 0;
}
```

Using Function Pointers

Given these math functions:

```c
  /* From math.h */
double sin(double);
double cos(double);
double tan(double);
```

and this declaration:

```c
double (*pMathFns[])(double) = {sin, cos, tan};
```

it is easy to invoke the functions pointed to:

```c
void TestFnArray1(void)
{
  int i;
  for (i = 0; i < 3; i++)
  {
    double x = pMathFns[i](2.0);
    printf("%f  ", x);
  }
  printf("\n");
}

Output:
0.909297  -0.416147  -2.185040
```

Function Pointers as Callbacks

qsort is a function that can sort an array of *any* data. Even types that haven't been invented yet!

```c
void qsort(void *base,
           size_t num,
           size_t width,
           int (*compare)(const void *elem1, const void *elem2)
          );
```

Parameters

- *base* - Start of target array
- *num* - Array size in elements
- *width* - Element size in bytes
- *compare* - Comparison function
  - *elem1* - Pointer to the key for the search
  - *elem2* - Pointer to the array element to be compared with the key

From MSDN documentation:

The **qsort** function implements a quick-sort algorithm to sort an array of num elements, each of width bytes. The argument base is a pointer to the base of the array to be sorted. **qsort** overwrites this array with the sorted elements. The argument compare is a pointer to a user-supplied routine that compares two array elements and returns a value specifying their relationship. **qsort** calls the compare routine one or more times during the sort, passing pointers to two array elements on each call:

```
compare((void *) elem1, (void *) elem2);
```

The routine must compare the elements, then return one of the following values:

```
Return Value          Description
-------------------------------------------
   < 0           elem1 less than elem2
     0           elem1 equivalent to elem2
   > 0           elem1 greater than elem2
```

The array is sorted in increasing order, as defined by the comparison function. To sort an array in decreasing order, reverse the sense of "greater than" and "less than" in the comparison function.

Example

This is the comparison function that will be used to determine the order:

```
int compare_int(const void *arg1, const void *arg2)
{
  int left = *(int *)arg1;  /* Can't dereference a void * */
  int right = *(int *)arg2; /* Can't dereference a void * */

  if (left < right)
    return -1;
  else if (left > right)
    return 1;
  else
    return 0;
}
```

This is usually written in a more compact way:

```
int compare_int1(const void *arg1, const void *arg2)
{
  return *(int *)arg1 - *(int *)arg2;
}
```

This will work nicely as the last parameter to the **qsort** function:

```
void qsort(void *base,
           size_t num,
           size_t width,
           int (*compare)(const void *elem1, const void *elem2)
          );
```

A program using the function:

```
void PrintInts(int array[], int size)
{
  int i;
  for (i = 0; i < size; i++)
    printf("%i ", array[i]);
  printf("\n");
}

void TestInts(void)
{
  int array[] = {5, 12, 8, 4, 23, 13, 15, 2, 13, 20};

  PrintInts(array, 10);               /* print the array        */
  qsort(array, 10, 4, compare_int1);  /* sort the array         */
  PrintInts(array, 10);               /* print the sorted array */
}

Output:
5 12 8 4 23 13 15 2 13 20
2 4 5 8 12 13 13 15 20 23
```

By creating another comparison function, we can sort in descending order:

```
int compare_int2(const void *arg1, const void *arg2)
{
```

```
  return *(int *)arg2 - *(int *)arg1;
}
```

How could we have written the above to take advantage of code reuse?

Now we can do:

```
void TestInts(void)
{
  int array[] = {5, 12, 8, 4, 23, 13, 15, 2, 13, 20};

  PrintInts(array, 10);              /* print original array             */
  qsort(array, 10, 4, compare_int1); /* sort in ascending order          */
  PrintInts(array, 10);              /* print sorted array (ascending)   */
  qsort(array, 10, 4, compare_int2); /* sort in descending order         */
  PrintInts(array, 10);              /* print sorted array (descending) */
}

Output:
5 12 8 4 23 13 15 2 13 20
2 4 5 8 12 13 13 15 20 23
23 20 15 13 13 12 8 5 4 2
```

Given a POINT structure we can code comparison functions. What does it mean for one structure to be *greater* or *less* than another?

```
struct POINT
{
  int x;
  int y;
};
```

A comparison function for comparing the **x** member: (note the function name)

```
int compare_ptsx(const void *arg1, const void *arg2)
{
  return ((struct POINT *)arg1)->x - ((struct POINT *)arg2)->x;
}
```

A comparison function for comparing the **y** member: (note the function name)

```
int compare_ptsy(const void *arg1, const void *arg2)
{
  return ((struct POINT *)arg1)->y - ((struct POINT *)arg2)->y;
}
```

Now we can use them in a program:

```
void PrintPts(const struct POINT pts[], int size)
{
  int i;
  for (i = 0; i < size; i++)
    printf("(%i,%i) ", pts[i].x, pts[i].y);
  printf("\n");
}

void TestStructs1(void)
{
    /* Array of 5 POINT structs */
  struct POINT pts[] = { {3, 5}, {1, 4}, {7, 2}, {2, 5}, {1, 8} };

    /* These values are calculated at compile time
  int count = sizeof(pts) / sizeof(pts[0]);
  int size = sizeof(pts[0]);

  PrintPts(pts, count);                     /* print the points        */
  qsort(pts, count, size, compare_ptsx);    /* sort the points (on x)  */
  PrintPts(pts, count);                     /* print the sorted points */
  qsort(pts, count, size, compare_ptsy);    /* sort the points (on y)  */
  PrintPts(pts, count);                     /* print the sorted points */
}

Output:
(3,5) (1,4) (7,2) (2,5) (1,8)
(1,4) (1,8) (2,5) (3,5) (7,2)
(7,2) (1,4) (3,5) (2,5) (1,8)
```

We can do something more "exotic" with the POINTS like sorting by the distance from the origin. Here's one way of doing that:

```
int compare_ptsd(const void *arg1, const void *arg2)
{
  struct POINT *pt1 = (struct POINT *)arg1;  /* first point
  struct POINT *pt2 = (struct POINT *)arg2;  /* second point
```

```
    /* calculate distances from origin
  double d1 = sqrt( (pt1->x * pt1->x) + (pt1->y * pt1->y) );
  double d2 = sqrt( (pt2->x * pt2->x) + (pt2->y * pt2->y) );
  double diff = d1 - d2;

    /* return -1, 0, 1 depending on the difference
  if (diff > 0)
    return 1;
  else if (diff < 0)
    return -1;
  else
    return 0;
}
```

Then test it:

```
void TestStructs1(void)
{
    /* Array of 5 POINT structs: [A,B,C,D,E]
  struct POINT pts[] = { {3, 5}, {1, 4}, {7, 2}, {2, 5}, {1, 8} };

    /* These values are calculated at compile time
  int count = sizeof(pts) / sizeof(pts[0]);
  int size = sizeof(pts[0]);

  PrintPts(pts, count);                     /* print the points
  qsort(pts, count, size, compare_ptsd);    /* sort the points (by distance from 0,0)
  PrintPts(pts, count);                     /* print the sorted points
}

Output:
(3,5) (1,4) (7,2) (2,5) (1,8)    [A,B,C,D,E]
(1,4) (2,5) (3,5) (7,2) (1,8)    [B,D,A,C,E]
```

Diagram:

Distances from origin: A(5.83), B(4.12), C(7.28), D(5.38), E(8.06)

Jump Tables

A *jump table* is simply a table (array) of function pointers. Instead of searching through the list of functions using an **if-then-else** paradigm, we just index into the table.

Assuming we have a function for each operation on a calculator:

```
double add(double operand1, double operand2)
{
  return operand1 + operand2;
}

double subtract(double operand1, double operand2)
{
  return operand1 - operand2;
}

double multiply(double operand1, double operand2)
{
  return operand1 * operand2;
}

double divide(double operand1, double operand2)
{
  return operand1 / operand2;
}
```

We can create a calculator program around these functions:

```
enum OPERATION {ADD, SUB, MUL, DIV};

void DoMath1(double operand1, double operand2, enum OPERATION op)
{
  double result;
  switch (op)
  {
    case ADD:
      result = add(operand1, operand2);
      break;

    case SUB:
      result = subtract(operand1, operand2);
      break;
```

```c
    case MUL:
      result = multiply(operand1, operand2);
      break;

    case DIV:
      result = divide(operand1, operand2);
      break;

    // many other cases ....

  }

  printf("%f\n", result);
}
```

Calling the function:

```c
int main(void)
{
  DoMath1(3, 5, ADD);
  DoMath1(3.14, 2, MUL);
  DoMath1(8.4, 8.4, SUB);

  return 0;
}

Output:
8.000000
6.280000
0.000000
```

We can be much more efficient by using a jump table instead:

```c
  /* create a "jump table" of calculator functions
double (*operation[])(double, double) = {add, subtract, multiply, divide};

void DoMath2(double operand1, double operand2, enum OPERATION op)
{
    /* replace the entire switch statement with this one line:
  double result = operation[op](operand1, operand2);
  printf("%f\n", result);
}
```

The calling function, **main**, doesn't have to change. Extending the operations to include a `power` function:

```c
/* Implement the new functionality
double power(double operand1, double operand2)
{
  return pow(operand1, operand2);
}

/* Update the table
enum OPERATION {ADD, SUB, MUL, DIV, POW};
double (*operation[])(double, double) = {add, subtract, multiply, divide, power};
```

Use it:

```c
DoMath2(3, 5, ADD);
DoMath2(3.14, 2, MUL);
DoMath2(8.4, 8.4, SUB);
DoMath2(5, 3, POW);  /* new function
```

Output:

```
8.000000
6.280000
0.000000
125.000000
```