Lecture 14
Function Pointers

In this lecture

- Functions with variable number of arguments
- Introduction to function pointers
- Example of passing a function pointer to a function (qsort)
- Defining a function pointer
- Generic Data Types

Functions with variable number of arguments

Sometimes we may want to define a function that can take variable numbers of arguments. For example printf and scanf are two IO functions that take variable number of arguments.
The prototype of printf is

printf(char*, …);

where first argument char* is a string that contains the format statement for the print string (eg: "%d %x %i ") and the 3 dots (…) indicate a variable number of arguments equal to number of format symbols in the print string.

Let us consider writing our own function sum that can take a variable number of arguments. That is, we can call the function as **sum(2,x,y)** or **sum(3,x,y,z)** etc.

```
#include <stdarg.h>          /* header for variable argument list */
int  sum(int argcnt, …)      /* argcnt supplies count of other args */
{    va_list  ap;            /* argument pointer */
     int ans = 0;
     va_start(ap, argcnt);   /* initialize ap */
     while (argcnt-- > 0)    /* process all args */
       ans += va_arg(ap, int);
     va_end(ap) ;            /* clean up before function returns */
     return (ans);
 }
```

The ***<stdarg.h>*** is a macro library that contains a set of macros, which allows **portable functions,** that, accepts variable argument lists to be coded. Functions that have variable argument lists (such as *printf*()) do not use these macros are inherently non-portable, as different systems use different argument-passing conventions. There are few things in the above code that we need to understand.

va_start() This is a macro that is invoked to initialize *ap, a pointer to argument list.* This macro must be called to initialize the list before any calls to *va_arg()*.

va_arg() This macro returns the next argument in the list pointed to by *ap*. Each invocation of *va_arg()* modifies *ap* so that the values of successive arguments are returned in turn. The *type* parameter is the type the argument is expected to be. This is the type name specified such that the type of a pointer to an object that has the specified type can be obtained simply by suffixing a * to type. Different types can be mixed, but it is up to the routine to know what type of arguments are expected.

va_end() This macro is used to clean up; it invalidates *ap* for use (unless *va_start()* is invoked again).

Introduction to Function Pointers

Function Pointers provide an extremely interesting, efficient and elegant programming technique. You can use them to replace *switch/if*-statements, and to realize *late-binding*. Late binding refers to deciding the proper function during runtime instead of compile time. Unfortunately function pointers have complicated syntax and therefore are not widely used. If at all, they are addressed quite briefly and superficially in textbooks. They are less error prone than normal pointers because you will never allocate or deallocate memory with them.

What is a Function Pointer?

Function Pointers are pointers, i.e. variables, which point to an address of a function. The running programs get a certain space in the main memory. Both, the executable compiled program code and the used variables, are put inside this memory. Thus a function in the program code is also an address. It is only important how the compiler and processor interpret the memory the function pointer points to. A function can take many types of arguments including the **address of another function**. This can be an extremely elegant way to bind a function to a specific algorithm at runtime. For example, let us assume that we plan to use a sorting algorithm in one of our functions. At the compile time, we are not sure which sorting algorithm to use since we may not know the size of the data set n. For example, if **n < 100** we may use something like **insertion sort**, or if n is large, then we may decide to use something like **quicksort**. C provides an interesting way to achieve this by allowing the programmer to decide the algorithm at runtime.

Example of Passing a Function Pointer to a function

Perhaps a good example to understand the role of function pointers is to study the qsort utility in unix.

% man qsort

NAME

qsort - sorts an array

SYNOPSIS

#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size,
int(*compar)(const void *, const void *));

DESCRIPTION

The qsort() function sorts an array with nmemb elements of size size. The base argument points to the start of the array.

The contents of the array are sorted in ascending order according to a comparison function pointed to by compar, which is called with two arguments that point to the objects being compared.

The comparison function must return an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second. If two members compare as equal, their order in the sorted array is undefined.

RETURN VALUE

The qsort() function returns no value.

Examples of using qsort

Using qsort function requires defining the compare function that will be used to sort the array in order. For example, suppose we have an array of ints and we are to sort the integers in ascending order. Then we can define a compare function as follows.

```
int intcompare(const void* x, const void* y){
  int val = *((int*)x) - *((int*)y);
  if (val<0) return -1;
  else if (val>0) return 1;
  else return 0;
}
```

If the array is an array of strings and we sort the array using just the first character, then we can define a compare function as follows.

```
int namecmp(const void* s1, const void* s2){
  return (((char*)s1)[0] - ((char*)s2)[0]);
}
```

To use the functions the following program can be used.

```
int A[1000],i;
for (i=0;i<1000;i++)
   A[i] = rand()%1000;
qsort(A,1000,sizeof(int),intcompare);


char B[][10]={"guna","mccain","obama","paul","barr"};
qsort(*B,5,10,namecmp);
for (i=0;i<5;i++)
   printf("%s \n",B[i]);
```

Defining a Function Pointer

Functions like variables, can be associated with an address in the memory. We call this a **function pointer**. A specific function pointer variable can be defined as follows.

```
  int  (*fn)(int,int) ;
```

Here we define a function pointer fn, that can be initialized to any function that takes two integer arguments and return an integer. Here is an example of such a function

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

Now to initialize fn to the address of the sum, we can do the following.

```
fn = &sum;   /* make fn points to the address of sum */
```
or simply
```
fn = sum;      /* just ignore the & . Function names are just like array names
                    they are pointers to the structure they are referring to */
```

So we use the sum function in two ways.

```
int  x = sum(10,12);  /* direct call to the function  */
```
or
```
int x = (*fn)(12,10);  /* call to the function through a pointer */
```

More Examples
Let us consider the following mystery function that takes 3 arguments, two ints a and b and a function pointer fn that tells mystery what to do with the two ints

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

Now let us define two functions as follows.
int gcd(int a, int b) { ….}
int sumofsquares(int x, int y) { return (x*x + y*y);}

We can call mystery function with gcd or sumofsquares or sum function.

```
int main(){
   printf("%d \n", mystery(10,12,sum));
   printf("%d \n", mystery(10,12,gcd));
   printf("%d \n", mystery(10,12,sumofsqures));
}
```

Using Typedef's
The syntax of function pointers can sometimes be confusing. So we can use a typedef statement to make things simpler.
typedef <return_type> (* fpointer)(argument list);

so we can define a variable of type fpointer as follows
fpointer fp;

Another Example

```
typedef int (*Ptr)(int,int);
Ptr  fp = sum;
```

This can also be very useful in writing functions that return function pointers such as follows.

```
int  (*Convert(const char code)) (int, int) {
    if (code = = '+') return &Sum;
    if (code = = '-') return &Difference;
}
```

The above function takes a char as an argument and returns a function pointer of type Ptr (as defined above) based on if the char is + or –

Now we can simplify the Convert definition as follows.

```
Ptr Convert(const char code) {
    if (code = = '+') return &Sum;
    if (code = = '-') return &Difference;
}
```

Use the Convert function as follows

```
int main ( ) {
   int (*ptr)(int,int); /* or:  Ptr ptr; if you have a typedef */
   ptr = Convert('+');
   printf( "%d \n", ptr(2,4));
}
```

Function pointer is the address where the function begins in the memory. Function name can be used to refer to this address just as we use the name of an array to refer to the address of the first element of the array. Although function name can be preceded by &, we can ignore the & when referring to a function pointer. As an example, consider the function average defined as

```
double average(double n, double m){
  return (n+m)/2;
}
```

Now we can define a function pointer variable **fn** to hold the address of average as:

```
double (*fn)(double, double);
```

and then assign **fn** to the function average as follows.

```
fn = average;
```

Now fn is an alias for average and we can compute the value of the function at 10.5 and 20.5 as, for example using the code,

```
double x = (*fn)(10.5, 20.5);
```

In fact any function, that takes two doubles as arguments and return a double can be assigned this function pointer fn. Many programmers try to avoid function pointers because of the complex syntax. However, to make things easy, we can define a function pointer type alias using

```
typedef double (*ftype)(double, double);
```

and then define fn as follows.

ftype fn;

This makes it easy to deal with the confusing function pointer notation.

Passing Function Pointers to a Another Function

Function pointers can be passed in or return from function. For example, the function,

void foo(void* A, int n, ftype fp);

takes a function pointer of type ftype as an argument. Similarly, we can return a function pointer from a function as follows. The function foo

ftype foo(void* A, int n);

returns a function pointer of type ftype from the function foo. Next we will look at an important application of the function pointers. We do know that Java 5 and C++ Standard Template Library (STL) contains constructs to define generic functions. Generic functions are great programmer tool, as the programmer needs to write the code once for any data type and bind the type at the run time. Unfortunately C does not have any built in API's or libraries to handle algorithms that take generic data types. But we can easily define generic functions using C. Generic functions are quite useful, as they can be adapted to data of many types. In other words, we can implement an algorithm in the most general way so that any data type can be used with the function.

Generic Functions

One of the major benefits of a function pointer is that it can be passed to another function as one of the arguments. This can be quite useful when we try to write generic functions. Suppose we want to write a function to find how many elements are in an array (not the capacity, but how many places are taken). The Array can be an array of ints or array of characters or array of any other type. All we need to know is how to find the end of the array. For example, if we have an array of ints that has a sentinel (say ends with -999 or something like that), we can write a function that returns "true" when array element is equal to -999. Otherwise it returns false. Consider the following function.

```
int intfunc(int A[], int i){
  return (A[i] == -999);
}
```

The above function returns "true" if A[i] = -999 for some i. So how do we use such a function?

Suppose we build a generic function arraylen as follows. The purpose of the function is to find the size of the array (not the capacity) regardless of the data type. For example, an array of ints may have -999 as a sentinel elements, an array of characters may have '\0' as the sentinel, or an array of strings may have something like "end" as the final string. So regardless of the sentinel we need to write a function that can find the array length (not

the capacity). So we write the following function arraylen. The function takes two arguments, any array and a function pointer that defines sentinel criteria. We start with length = 0 and continue to apply term function until term(array, length) returns something other than 0.

```
typedef int (*fn)(void*, int);
int arraylen(void* any, fn term){
  int length = 0;
  while ( (*term)(any, length) == 0) length++;
  return length;
}
```

Convince yourself that this function returns the length of the array when called with

arraylen(A, (fn)intfunc)

Exercise: Write a function stringfunc as in intfunc that returns "true" if S[i] ='\0'
We can give more examples of how to use generic functions for doing lots of interesting things. Generally, if we write a sort function, using bubble sort algorithm for an array of ints, it would look something like:

```
void bubblesort(int A[], int n) {
int i, j;
for (i=0; i<n; i++)
  for (j=0; j<n-i-1; j++)
     if (A[j]>A[j+1]){
         int tmp = A[j];
        A[j] = A[j+1];
        A[j+1] = tmp;
      }
}
```

The problem with the above function is that it works only for an array of ints. Now let us suppose we need to write a sort function using bubble sort algorithm for a generic array. We now not only pass the array and its size, we also pass another function cmp, that will define how to compare two values from the array. We may also need to pass a swap function that defines how to swap two elements in the array. This is important since comparison and swap can be very specific to a data type. For example, while we can compare ints or doubles using < or >, we need to use strcmp to compare strings.

First let us develop a compare functions that returns 1, 0 or -1 based on if the first argument is greater, the same or smaller than the second argument. As an example, we will develop a compare function that compares strings using just its first character.

```
int stringcmp(char* s1, char* s2){
  if (s1[0] > s2[0]) return 1;
  else if (s1[0] < s2[0]) return -1;
  return 0;
}
```

The above function can be passed to any other function that can use the algorithm of how to compare two strings using just the first characters of the string.
Now let us suppose that we have an array of ANY TYPE and we need to sort it using bubble sort algorithm. We will need to send four arguments to this function.

- **An array of any type - void* A**
- **The size of the array – int n**
- **The compare function – a function pointer that compares two array elements**
- **The swap function – a function pointer that swaps two array elements**

We note that this is needed since the way we compare and swap integers may be different from way we compare and swap strings. No matter what data type we have in the array, we need to define a generic bubble sort algorithm.
First we will define two type aliases cmpfunc and swapfunc. The cmpfunc, takes any array and compares two elements and return 1, 0 or -1. The swapfunc, takes an array and two indices, swap the content of the array. Given below are the two typedefs that can be used.

```
typedef int (*cmpfunc)(void*, int, int);
typedef void (*swapfunc)(void*, int, int);
```

Now we define our generic bubble sort as follows.

```
void bubblesort(void* A, int n, cmpfunc cp, swapfunc sp) {
  int i, j;
  for (i=0; i<n-1;i++)
   for (j=0; j<n-i-1;j++)
     if ( (*cp)(A,j,j+1) > 0)
        (*sp)(A,j,j+1);
}
```

We note that the function takes any array, its size, and a compare function and a swap function that can help compare and swap array elements.

Let us now consider the functions defined for array of ints.

```
int intcomp(int A[],int i, int j){
  return (A[i]-A[j]);
}

void intswap(int A[], int i, int j){
  int temp = A[i];
  A[i] = A[j];
  A[j] = temp;
}
```

The above functions clearly define how to compare and swap two elements in an integer array. How do we use these functions in our bubble sort?
We can define an array of ints and simply call the bubble sort as follows.

```
int A[] ={34,12,67,45,90,16,46,99,10};
bubblesort(A,9,(cmpfunc)intcomp,(swapfunc)intswap);
```

This technique can be used to write any function for an array of any type. We only need to write the algorithm once and send the proper functions based on the type of the array.