CS10003: Programming & Data Structures

Dept. of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Autumn 2020

Pointers in Function

Passing Pointers to a Function

- Pointers are often passed to a function as arguments
  - Allows data items within the calling function to be accessed by the called function, altered, and then returned to the calling function in altered form
  - Useful for returning more than one value from a function
  - Still call-by-value, but now the address is copied, not the content

Passing Pointers to a Function

- Pointers are often passed to a function as arguments.
  - Allows data items within the calling program to be accessed by the function, altered, and then returned to the calling program in altered form.
  - Called *call-by-reference* (or by *address* or by *location*).
- Normally, arguments are passed to a function *by value*.
  - The data items are copied to the function.
  - Changes are not reflected in the calling program.

Example: Swapping

```
void  swap (int x, int y)
{
   int  t;
   t = x;
   x = y;
   y = t;
}

int main()
{
   int  a, b;
   a = 5;  b = 20;
   swap (a, b);
   printf ("\n a=%d, b=%d", a, b);
   return 0;
}
```

Output

```
a=5, b=20
```

Parameters passed by value, so changes done on copy, not returned to calling function

Example: Swapping using pointers

```
void swap (int *x, int *y)
{
   int  t;
   t = *x;
   *x = *y;
   *y = t;
}

int main()
{
   int  a, b;
   a = 5;  b = 20;
   swap (&a, &b);
   printf ("\n a=%d, b=%d", a, b);
   return 0;
}
```

Output

```
a=20, b=5
```

Parameters passed by address, changes done on the value stored at that address

- While passing a parameter to a function, when should you pass its address instead of the value?
  - Pass address if both these conditions are satisfied
    - The parameter value will be modified inside the function body
    - The modified value is needed in the calling function after the called function returns

- Consider the swap function to see this

Homework

Sort three numbers using swap function.

Pointers and Arrays

When an array is declared,

- The compiler allocates a *base address* and sufficient amount of storage to contain all the elements of the array in contiguous memory locations.

- The *base address* is the location of the first element (*index 0*) of the array.

- The compiler also defines the array name as a *constant pointer* to the first element.

Passing Arrays as Pointers

Both the forms below are fine in the function body, as arrays are passed by passing the address of the first element. Calling function calls it the same way

```
float  average  (int a, float x[])
{
    :
    sum = sum + x[i];
}

int main()
{
    int  n;
    float   list[100], avg;
    :
    avg  =  average (n, list);
    :
}
```

```
float  average  (int a, float *x)
{
    :
    sum = sum + x[i];
}

int main()
{
    int  n;
    float   list[100], avg;
    :
    avg  =  average (n, list);
    :
}
```

Example: function to find average

```
float avg (int array[], int size)
{
  int  *p, i , sum = 0;

  p = array;

  for (i=0; i<size; i++)
      sum = sum + *(p+i);

  return ((float) sum / size);
}
```

```
int main()
{
  int x[100], k, n;

  scanf ("%d", &n);

  for (k=0; k<n; k++)
     scanf ("%d", &x[k]);

  printf  ("\nAverage is %f",avg(x,n));
  return 0;
}
```

```
int main()
{
   int x[100], k, n;

   scanf ("%d", &n);

   for (k=0; k<n; k++)
       scanf ("%d", &x[k]);

   printf  ("\nAverage is %f",avg(x,n));
   return 0;
}
```

```
float avg (int array[], int size)
{
   int  *p, i , sum = 0;

   p = array;

   for (i=0; i<size; i++)
         sum = sum + p[i];

   return ((float) sum / size);
}
```

The pointer p can be subscripted also just like an array!

Arrays and pointers

An array name is an address, or a pointer value.

Pointers as well as arrays can be subscripted.

A pointer variable can take different addresses as values.

An array name is an address, or pointer, that is fixed.
It is a CONSTANT pointer to the first element.

Arrays

Consequences:

`ar` is a pointer

`ar[0]` is the same as `*ar`

`ar[2]` is the same as `*(ar+2)`

We can use pointer arithmetic to access arrays more conveniently.

Declared arrays are only allocated while the scope is valid

```
char *foo() {
    char string[32]; ...;
    return string;
}
```
is incorrect

Arrays

Array size `n`; want to access from `0` to `n-1`, so you should use counter AND utilize a constant for declaration & incr

Wrong

```
int i,n=10;
int ar[n];
for(i = 0; i < n; i++){ ... }
```

Right

```
#define ARRAY_SIZE 10
int i, a[ARRAY_SIZE];
for(i = 0; i < ARRAY_SIZE; i++){ ... }
```

Why? Single Source of Truth

Arrays in Functions

An array parameter can be declared as an array <u>or</u> a pointer; an array argument can be passed as a pointer.

Can be incremented

```
int strlen(char s[])        int strlen(char *s)
{                           {


}                           }
```

Arrays and pointers

int a[20], i, *p;

The expression a[i] is equivalent to *(a+i)

p[i] is equivalent to *(p+i)

When an array is declared the compiler allocates a sufficient amount of contiguous space in memory. The base address of the array is the address of a[0].

Suppose the system assigns 300 as the base address of a. a[0], a[1], ...,a[19] are allocated 300, 304, ..., 376.

Arrays and pointers

#define N 20

int a[N], i, *p, sum;

p = a; is equivalent to p = &a[0];

p is assigned 300.

Pointer arithmetic provides an alternative to array indexing.

p=a+1; is equivalent to p=&a[1]; (p is assigned 304)

```
for (p=a; p<&a[N]; ++p)
    sum += *p ;
```

```
p=a;
for (i=0; i<N; ++i)
    sum += p[i] ;
```

```
for (i=0; i<N; ++i)
    sum += *(a+i) ;
```

Returning multiple values from a function

- Return statement can return only one value
- What if we want to return more than one value?
- Use pointers
  - Return one value as usual with a return statement
  - For other return values, pass the address of a variable in which the value is to be returned

Example: Returning max and min of an array

Both returned through pointers (could have returned one of them through return value of the function also)

```
void MinMax(int A[], int n, int *min,
int *max)
{
   int i, x, y;
   x = y = A[0];
   for (i=1; i<n; ++i) {
     if (A[i] < x) x = A[i];
     if (A[i] > y) y = A[i];
   }
   *min = x; *max = y;
}
```

```
int main()
{
   int n, min, max, i, A[100];
   scanf("%d", &n);
   for (i=0; i<n; ++i)
     scanf("%d", &A[i]);
   MinMax(A, n, &min, &max);
   printf("Min and max are %d, %d", min, max);
   return 0;
}
```

Structures and Functions

A structure can be passed as argument to a function.

A function can also return a structure.

The process shall be illustrated with the help of an example.

- A function to add two complex numbers.

Example: Passing structure pointers

```
void add(struct cplx *x, struct cplx *y, struct cplx *t)
{
   t->re = x->re + y->re;
   t->im = x->im + y->im;
}
```

```
struct cplx  {
     float  re;
     float  im;
};

int main()
{
   struct cplx a, b, c;
   scanf("%f%f", &a.re, &a.im);
   scanf("%f%f", &b.re, &b.im);
   add(&a, &b, &c) ;
   printf("\n %f %f", c.re, c.im);
   return 0;
}
```

Remember: The Actual Mechanism

When an array is passed to a function, the values of the array elements are not passed to the function.

- The array name is interpreted as the address of the first array element.
- The formal argument therefore becomes a pointer to the first array element.
- When an array element is accessed inside the function, the address is calculated using the formula stated before.
- Changes made inside the function are thus also reflected in the calling program.

Typecasting

Typecasting is mostly not required in a well written C program. However, you can do this as follows:

```
char c = ‘5’
char *d = &c;
int *e = (int*)d;
```

Remember (sizeof(char) != sizeof(int))

Typecasting

void pointers

Pointers defined to be of specific data type cannot hold the address of another type of variable.

It gives syntax error on compilation. Else use a void pointer (which is a general purpose pointer type), which can point to variables of any data type.

But while dereferencing, we need an explicit type cast.

Example

```
#include<stdio.h>
int main()
{
      float pi=3.14128;
      int num=100;
      void *p;
      p=&pi;
printf("First p points to a float variable and access pi=%.5f\n",  *((float *)p));
      p=&num;
printf("Then p points to an integer variable and access num=%d\n",*((int *)p));
      return 0;
}
```

Output

```
First p points to a float variable and access pi=3.14128
Then p points to an integer variable and access num=100
```

Pointers to Pointers

Pointer is a type of data in C

hence we can also have pointers to pointers

Pointers to pointers offer flexibility in handling arrays, passing pointer variables to functions, etc.

General format:

<data_type> **<ptr_to_ptr>;

<ptr_to_ptr> is a pointer to a pointer pointing to a data object of the type <data_type>

This feature is often made use of while passing two or more dimensional arrays to and from different functions.

Example

```c
#include<stdio.h>
int main() {
    int *iptr;
    int **ptriptr;
    int data;
    iptr=&data;
    ptriptr=&iptr;
    *iptr=100;
    printf("variable 'data' contains %d\n",data);
    **ptriptr=200;
    printf("variable 'data' contains %d\n",data);
    data=300;
    printf("variable 'data' contains %d\n",**ptriptr);
    return 0;
}
```

Output

```
variable 'data' contains 100
variable 'data' contains 200
variable 'data' contains 300
```

scanf Revisited

```
int   x,  y ;
printf (“%d %d %d”,  x, y, x+y) ;
```

What about scanf ?

```
scanf (“%d %d %d”, x, y, x+y);
```

NO

```
scanf (“%d %d”, &x, &y);
```

YES

Thank You!