CPNM Lecture 11 - Pointers

Mridul Sankar Barik

Jadavpur University

2022-23

Pointers - Introduction I

- A pointer is a variable that holds a memory address of another variable
- Declaration

```
int *p;
int i, j, a[10], b[20], *p, *q;
char *p;
float *q;
```

- Type of a pointer variable defines the type of variables to which the pointer will point
- Pointer Operators
  - Unary address of operator (&): returns the memory address of its operand
  - Unary indirection operator (*): returns the value stored at the address specified by its operand

Pointers - Introduction II

- On the right hand side of an assignment statement a pointer can be used to assign its value to another pointer

```
int x=99;
int *p1, *p2;
p1=&x;
p2=p1;
printf("%p, %p %d %d\n", p1, p2, *p1, *p2);
```

- Declaring a pointer type variable only allocates space for it
- A pointer type variable must be initialized before used with an indirection operator
- `int *p = &i;` $\Rightarrow$ `*p` is an alias of `i` $\Rightarrow$ changing the value of `*p` also changes the value of `i` and vice versa

Pointers - Introduction III

- if `p` and `q` are pointer type variables, then
  - `q=p;` $\Rightarrow$ value of the pointer type variable `p` is copied as the value of another pointer type variable `q`; here, both the values are addresses
  - `*q=*p;` $\Rightarrow$ value stored at the address pointed to by the pointer type variable `p` is copied as the value to be stored at the address pointed to by the pointer type variable `q`

Pointer as Arguments I

- **Call by Value**: Any changes made to the formal parameters inside a function is not reflected to the corresponding actual parameters outside the function

```
void swap(int a, int b){
    int t; t=a; a=b; b=t;
}
main(){
    ...
    printf("%d %d\n", x, y);
    swap(x, y);
    printf("%d %d\n", x, y);
}
```

- In C, we can achieve effects of **Call by Reference**, by using pointer type variables. Although, C does not support Call by Reference in its true semantics.

```
void swap(int *a, int *b){
    int t; t=*a; *a=*b; *b=t;
}
main(){
    ...
    printf("%d %d\n", x, y);
    swap(&x, &y);
    printf("%d %d\n", x, y);
}
```

Pointers as Return Values I

```
int *max(int *a, int *b){
    if(*a > *b)
        return a;
    else
        return b;
}
int main(){
    int *p, i=2, j=5;
    p=max(&i, &j);
}
```

- A function can return a pointer which is passed as an argument
- A function could also return a pointer to an external variable or to a local variable that's been declared `static`

Pointers as Return Values II

```
int *f(void){
    int i;
    ...
    return &i;
}
```

- Once `f()` returns memory location corresponding to local variable `i` is de-allocated; some compilers issue warning messages

Pointer and Arrays I

```
int a[3] = {1, 2, 3}, *p;
p = a; p = &a[0];
printf("%p", &a[1]);
printf("%p", a+1);
printf("%d", a[1]);
printf("%d", *(a+1));
```

- Name of an array is the address of the first element
- Address of `i`th element is `(a+i)` or `&a[i]`

Pointer and Arrays II

```
int *p[3]; /*array of 3 pointers to int*/
int (*q)[3]; /*pointer to array of 3 int*/
int a[3]={1, 2, 3};
int *e;

q=a;/*produces warning: assignment from incompatible
        pointer type*/
q=&a;

printf("a=%p &a=%p q=%p, *q=%p\n", a, &a, q, *q);
    /*All four prints the same address*/
e=&a /*produces warning: assignment from incompatible
        pointer type*/
e=a;
printf("e=%p, *e=%d\n", e, *e);
```

Pointer Conversion I

- A `void*` pointer can be assigned to any other type of pointer or any other pointer to a void* pointer
- No explicit cast is required to convert to or from a void* pointer
- Conversion from one type of pointer to another may create undefined behaviour

Dynamic Memory Allocation I

- Static memory allocation: storage is allocated by the compiler before the program is run
- To create new storage when the program is running use `malloc` $\Rightarrow$ returns a pointer to the allocated memory, or `NULL` if the request fails

```
int *p = (int *) malloc(5*sizeof(int));
for(i=0;i<5;i++)
    scanf("%d", (p+i));
for(i=0;i<5;i++)
    printf("%d", *(p+i));
free(p);
```

- `free` $\Rightarrow$ deallocates memory

Dynamic Memory Allocation II

- Creating 2D arrays dynamically

```
#define ROWS 5
#define COLS 8
...

int **b, i, j;
b=(int **)malloc(ROWS*sizeof(int *));

for(i=0; i<ROWS; i++){
    *(b+i) = (int *) malloc(COLS*sizeof(int));
}

for(i=0; i<ROWS; i++){
    for(j=0; j<COLS; j++){
        *(*(b+i)+j)=0;
    }
}

for(i=0; i<ROWS; i++){
    for(j=0; j<COLS; j++){
       printf("%d ", b[i][j]);
    }
    printf("\n");
}
```

Dynamic Memory Allocation III

- Dynamic memory is allocated from **Heap** area which is accessible to all the functions in program $\Rightarrow$ You can allocate space dynamically inside a function and return a pointer to it to the caller.

```
int **createMat(int r, int c){
    int **b, i;
    b=(int **)malloc(r*sizeof(int *));

    for(i=0; i<r; i++)
        *(b+i) = (int *) malloc(c*sizeof(int));
    return(b);
}
...
int **x;
x=createMat(ROWS, COLS);
```

Dynamic Memory Allocation IV

- Passing 2D array to a function

```
int printMat(int (*p)[3]){
  int i, j;
  for(i=0;i<3;i++){
    for(j=0;j<3;j++)
      printf("%d ", p[i][j]);
    printf("\n");
  }
}
...
int b[3][3]={{1,2,3},{4,5,6},{7,8,9}};
printMat(b);
```

Dynamic Memory Allocation V

- `calloc()` allocates memory for an array of `n` elements of `b` bytes each and returns a pointer to the allocated memory, additionally, the memory is set to zero.

```
int *p = (int *) calloc(5, sizeof(int));

for(int i=0; i<5; i++)
    printf("%d ", *(p+i)); //five 0's printed
```

- `realloc()` function changes the size of the memory block pointed to by `p` to `b` bytes. The contents will be unchanged in the range from the start of the region up to the minimum of the old and new sizes.

```
p=realloc(p, 10*sizeof(int));
```

Pointer Arithmetic I

- C supports three forms of pointer arithmetic
  - Adding an integer to a pointer
  - Subtracting an integer from a pointer
  - Subtracting one pointer from another
- Adding an integer `j` to a pointer `p` yields a pointer to the clement `j` places after the one that `p` points to
- If `p` points to the array element `a[i]`, then `p+j` points to `a[i+j]` (provided, `a[i+j]` exists)
- If `p` points to the array element `a[i]`, then `p-j` points to `a[i-j]` (provided, `a[i-j]` exists)
- If `p` points to the array element `a[i]` and `q` points to the array element `a[j]`, then `p-q` is equal to `i-j`
- Performing arithmetic on a pointer that doesn't point to an array element causes undefined behavior
- The effect of subtracting one pointer from another is undefined unless both point to elements of the same array

Pointer Arithmetic II

- Pointers can be compared using relational operators ($<, <=, >, >=$) and the equality operator ($==, !=$)

```
int a[10], *p, *q, i;

p=&a[2];
q=&a[4];
printf("%d %d", *(p+2), *q);
printf("%d", p-q);
```

Pointers and Multidimensional Array I

- Consider two dimensional array

```
int a[ROWS][COLS];

int row, col;
for(row = 0; row < ROWS; row++)
    for(col = 0; col < COLS; col++)
        a[row][col]=0;
```

- The same action can be achieved using pointer

```
int *p;
for(p = &a[0][0]; p<=&a[ROWS-1][COLS-1]; p++ )
    *p = 0;
```

Pointers and Multidimensional Array II

- To set the elements to zero in `i`th row of a two-dimensional array

```
int a[ROWS][COLS], *p, i;
...
for(p = a[i]; p < a[i] + COLS; p++)
    *p = 0;
```

- Two statements `p = &a[i][0]` and `p = a[i];` are equivalent
  - For any array `a`, the expression `a[i]` is equivalent to `*(a+i)`
  - Thus, `&a[i][0]` is same as `&(*(a[i]+0))` or `&*a[i]` or `a[i]`
- To set the elements in column `i` of a two-dimensional array to zero

```
int a[ROWS][COLS], (*p)[COLS], i;

for(p = &a[0]; p < &a[ROWS]; p++)
   (*p)[i] = 0;
```

Pointers and Multidimensional Array III

- Here, `p` is declared as a pointer to an array of length `COLS` whose elements are integers

Function Pointers I

- Function Pointers are pointers, i.e. variables, which contain addresses of functions

```
/*function returning int and taking one int argument*/
int func(int a);

/* function returning pointer to int
and taking one int argument*/
int *func(int a);

/* pointer to function returning int
and taking one int argument*/
int (*func)(int a);
```

- Assign the address of the right sort of function just by using its name
- Like an array, a function name is turned into an address when it's used in an expression

Function Pointers II

- You can call the function using one of two forms

```
(*func)(1);
/* or */
func(1);
```

- Example

```
#include <stdio.h>
#include <stdlib.h>
void func(int);
main(){
    void (*fp)(int);
    fp = func;
    (*fp)(1);
    fp(2);
}
void func(int arg){
    printf("%d\n", arg);
}
```

Function Pointers III

- Array of pointers to functions

```
void f1(int, float);
void f2(int, float);
void f3(int, float);

void (*fparr[3])(int, float) = { f1, f2, f3 };

/* then call one */

fparr[2](1, 3.4);
```