CS11001/CS11002
Programming and Data Structures (PDS)
(Theory: 3-1-0)

Pointers to Pointers

Examples of pointer arithmetic

```
int a=10, b=5, *p, *q;
 p=&a;
 q=&b;
 printf("*p=%d,p=%x\n",*p,p);
 p=p-b;
 printf("*p=%d,p=%x\n",*p,p);
 printf("a=%d, address(a)=%x\n",a,&a);
```

Output:

*p=10,p=bfbe4de8
*p=10,p=bfbe4dd4
a=10, address(a)=bfbe4de8

Some more valid pointer arithmetic

```
#include<stdio.h>

main()
{
  int a=10, b=5, *p, *q;
  p=&a;
  q=&b;

  printf("*p=%d,p=%x\n",*p,p);
  p=p-b;
  printf("*p=%d,p=%x\n",*p,p);
  printf("a=%d, address(a)=%x\n",a,&a);
  //p=p-q;
  //printf("*p=%d,p=%x\n",*p,p);
  p=p+a;
  //p=(int *)(p-q)-a;
  printf("*p=%d,p=%x\n",*p,p);
  p=p-a;
  printf("*p=%d,p=%x\n",*p,p);
  p=p+b;
  printf("*p=%d,p=%x\n",*p,p);
}
```

*p=10,p=bf92f328
*p=10,p=bf92f314
a=10, address(a)=bf92f328
*p=2212752,p=bf92f33c
*p=2212752,p=bf92f314
*p=10,p=bf92f328
(Here integer variable needs 4 bytes)

If a pointer p is to a type, d_type, when incremented by i, the new address p points to is:

current_address+i*sizeof(d_type)

Similarly for decrementation

Invalid pointer arithmetic

- p=-q;
- p<<=1;
- p=p+q;
- p=p+q+a;
- p=p*q;
- p=p*a;
- p=p/q;
- p=p/b;
- p=a/p;

Subtraction of Pointers

```
#include<stdio.h>
main()
{
  int *p, *q;
  float *f, *g;
  q=p+1;
  g=f+1;
  printf("%d\n",(short int*)q-(short int*)p);
  printf("%d\n",(float *)g-(float *)f);
}
```

When two pointers are subtracted, the results are of type size_t

Both the printf statement outputs 1.

Even though the numerical values of the pointers differ by 4 in case of integers/float, this difference is divided by the size of the type being pointed to.

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

```
#include<stdio.h>

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);
}
```

Output

- variable 'data' contains 100
- variable 'data' contains 200
- variable 'data' contains 300

Pointers and Arrays

- **The elements of an array can be efficiently accessed by using a pointer.**
- Array elements are always stored in contiguous memory space.
- Consider an array of integers and an int pointer:
- #define MAXSIZE 10
  int A[MAXSIZE], *p;

The following are legal assignments for the pointer p:

- p = A; /* Let p point to the 0-th location of the array A */
  p = &A[0]; /* Let p point to the 0-th location of the array A */
  p = &A[1]; /* Let p point to the 1-st location of the array A */
  p = &A[i]; /* Let p point to the i-th location of the array A */
  Whenever p is assigned the value &A[i], the value *p refers to the array element A[i].

- Pointers can be incremented and decremented by integral values.
- After the assignment p = &A[i]; the increment p++ (or ++p) lets p one element down the array, whereas the decrement p-- (or --p) lets p move by one element up the array. (Here "up" means one index less, and "down" means one index more.)
- Similarly, incrementing or decrementing p by an integer value n lets p move forward or backward in the array by n locations. Consider the following sequence of pointer arithmetic:
- p = A; /* Let p point to the 0-th location of the array A */
- p++; /* Now p points to the 1-st location of A */
- p = p + 6; /* Now p points to the 7-th location of A */
- p += 2; /* Now p points to the 9-th location of A */
- --p; /* Now p points to the 8-th location of A */
- p -= 5; /* Now p points to the 3-rd location of A */
- p -= 5; /* Now p points to the (-2)-nd location of A */

Oops! What is a negative location in an array?

Like always, C is pretty liberal in not securing its array boundaries.

As you may jump ahead of the position with the largest legal index, you are also allowed to jump before the opening index (0).

Though C allows you to do so, your run-time memory management system may be unhappy with your unhealthy intrusion and may cause your program to have a premature termination (with the error message "Segmentation fault").

It is the programmer's duty to ensure that his/her pointers do not roam around in prohibited areas.

Accessing Array elements

```
#include<stdio.h>
main()
{
 int iarray[5]={1,2,3,4,5};
 int i, *ptr;
 ptr=iarray;
 for(i=0;i<5;i++)
  {
    printf("%x - %d\n",ptr,*ptr);
    ptr++;
  }
}
```

```
bf90e2f8 - 1
bf90e2fc - 2
bf90e300 - 3
bf90e304 - 4
bf90e308 - 5
```

- The name of the array is the starting address (base address) of the array.
- It is the address of the first element in the array.
- Thus it can be used as a normal pointer, to access the other elements in the array.

```
 main()
{ int iarray[5]={1,2,3,4,5};
  int i;
  for(i=0;i<5;i++)
      printf("%x:%d\n",(iarray+i),*(iarray+i));
}
```

Output

- bff5814c:1
- bff58150:2
- bff58154:3
- bff58158:4
- bff5815c:5

Equivalent Expressions

```
#include<stdio.h>
main()
{
 int i;
 int a[5]={1,2,3,4,5}, *p = a;
 for(i=0;i<5;i++,p++)
 {
   printf("%d %d",a[i],*(a+i));
   printf(" %d %d %d\n",*(i+a),i[a],*p);
 }
}
```

Output:

```
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3
4 4 4 4 4
5 5 5 5 5
```

Passing Arrays to functions (contd.)

```
#include<stdio.h>
#include<string.h>
#include<malloc.h>

void sortbyptrswap(char **per, int n)
{
  int i, j;
  char *temp;
  for(i=0;i<n;i++)
    for(j=0;j<n-1;j++)
     if(strcmp(per[j],per[j+1])>0)
     {
      temp=per[j];
      per[j]=per[j+1];
      per[j+1]=temp;
     }
}
```

```
main()
{
  int i, n=0;
  char *person[100]; //array of 100 pointers
  int choice;
  do{
    person[n]=(char*)malloc(40);//allocate space of 40 characters for each pointer
    printf("Enter Name:");
    scanf("%s",person[n++]);
    printf("Enter another (1/0)?\n");
    fflush(stdin);
    scanf("%d",&choice);
  }while(choice);

  printf("Unsorted List:");
  for(i=0;i<n;i++)
    printf("\n%s",person[i]);
  sortbyptrswap(person,n);
  printf("sorted List:\n");
  for(i=0;i<n;i++)
    printf("\n%s",person[i]);

}
```

Output

```
Enter Name:ram
Enter another (1/0)?
1
Enter Name:shyam
Enter another (1/0)?
1
Enter Name:jadu
Enter another (1/0)?
tom
Enter Name:Enter another (1/0)?
0
Unsorted List:
ram
shyam
jadu
tom
sorted List:
jadu
ram
shyam
tom-
```