CS11001/CS11002 Programming and Data Structures (PDS)

(Theory: 3-1-0)

Pointers to Pointers

Pointers

- Pointers are addresses in memory.
- In order that the user can directly manipulate memory addresses,
- C provides an abstraction of addresses.
- The memory location where a data item resides can be accessed by a pointer to that particular data type.
- C uses the special character * to declare pointer data types.
- A pointer to a double data is of data type double *.
- A pointer to an unsigned long int data is of type unsigned long int *.

Declaration of Pointers

- You may also declare pointers simultaneously with other variables.
- All you have to do is to put an asterisk (*) before the name of each pointer.
  - long int *pointer, *p;
  - float *fptr;
  - double *standard;
- pointer and p are pointers to data of type long int.
- Similarly, standard is a pointer to a double data.

Typecasting

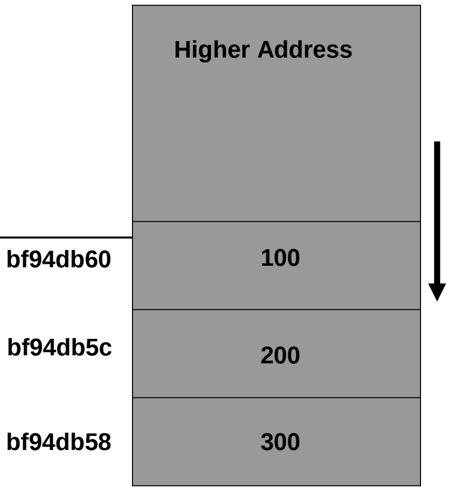
- Assume that piTo4 is a double variable that stores the value 97.4090910340
- (int *)piTo4An integer pointer that points to the memory location 97
- **Pointer constants**
  - Well, there are no pointer constants actually.
  - It is dangerous to work with constant addresses.
  - You may anyway use an integer as a constant address.
  - But doing that lets the compiler issue you a warning message. Finally, when you run the program and try to access memory at a constant address, you are highly likely to encounter a frustrating mishap known as "Segmentation fault".
  - That's a deadly enemy. Try to avoid it as and when you can!
- Incidentally, there is a pointer constant that is used widely. This is called NULL. A NULL pointer points to nowhere.

Address Operator &

```
#include<stdio.h>
main()
{
  int a=100; int b=200; int c=300;
  printf(“Address: %x contains value: %d\n”,&a,a);
  printf(“Address: %x contains value: %d\n”,&b,b);
  printf(“Address: %x contains value: %d\n”,&c,c);
   printf("size of int =%d\n",sizeof(int));
}
```

Addresses and contents of variables

```
-bash-3.2$ ./a.out
Address: bf94db60
   contains value: 100
Address: bf94db5c
   contains value: 200
Address: bf94db58
   contains value: 300
size of int =4
```



Pointer Variables

<data_type> *<ptrvar_name>;

- <data_type> could be any one of the primitive C data types, like int, char, float.
- The <ptrvar_name> is any valid C variable name.
- The symbol * differentiates a common variable from the pointer variable.
- It indicates the C compiler that the variable <ptrvar_name> is a pointer variable.
- The pointer holds the address of any variable of the specified type, <data_type>

Pointers accessing variables

```
#include <stdio.h>
main()
{
  int *iptr;
  int var1, var2;
  var1=10; var2=20;
  iptr=&var1;
  printf("Address and contents of var1 is %x and
       %d\n",iptr,*iptr);
  iptr=&var2;
  printf("Address and contents of var2 is %x and
       %d\n",iptr,*iptr);
  *iptr=125;
  printf("Value of var2 is %d\n",var2);
  (*iptr)++;
  printf("Address and contents of var2 is %x and
       %d\n",iptr,*iptr);
  *iptr++;
  printf("Value of var2 is %d\n",var2);
  printf("Address and contents of var1 is %x and
       %d\n",iptr,*iptr);

}
```

```
Address and contents of var1 is
bfc7fe8c and 10
Address and contents of var2 is
bfc7fe88 and 20
Value of var2 is 125
Address and contents of var2 is
bfc7fe88 and 126
Value of var2 is 126
Address and contents of var1 is
bfc7fe8c and 10
```

int *ptr and int* ptr are same.

However the first one helps you to declare in one statement:

    int *ptr, var1;

Dereferencing Pointers

- Dereferencing is an operation performed to access and manipulate data contained in the memory location.
- A pointer variable is said to be dereferenced when the unary operator *, in this case called the indirection operator, is used like a prefix to the pointer variable or pointer expression.
- An operation performed on the dereferenced pointer directly affects the value of the variable it points to.

Example

```
#include<stdio.h>
main()
{
  int *iptr, var1, var2;
  iptr=&var1;
  *iptr=25;
  *ipr += 10;
   printf(“variable var1 contains %d\n”,var1);
   var2=*iptr;
   printf(“variable var2 contains %d\n”,var2);
   iptr=&var2;
   *iptr += 20;
   printff(“variable var2 now has %d\n”,var2);
 }
```

Output

```
variable var1 contains 35
variable var2 contains 35
variable var2 now has 55
```

Thus the two use of * are to be noted.

int *p for declaring a pointer variable

*p=10 is for indirection to the value in the address pointed by the variable p.

This power of pointers is often useful, where direct access via variables is not possible.

Example?

Another example

```
#include<stdio.h>
main()
{
  int a=5, b=10;
  int *p;
  p=&a;
  printf(“a=%d,b=%d\n”,a,b);
  b=*p;
 printf(“a=%d,b=%d\n”,a,b);
 printf(“address of a is %x\n”,&a);
 printf(“address of b is %x\n”,&b);
 printf(“address pointed to by p is %x\n”,p);
 printf(“value pointer p accesses is %d\n”,*p);
}
```

Output

- a=5,b=10
- a=5,b=5
- address of a is bfd03f0c
- address of b is bfd03f08
- address pointed to by p is bfd03f0c
- value pointer p accesses is 5

Swapping two numbers

```
■ void main( )
{int i, j; scanf("%d %d", &a, &b);
  printf("After swap: %d %d",a,b);
  swap(&a,&b);
  printf("After swap: %d %d",a,b);
}
■ void swap(int *a, int *b)
  {
 int temp = *a;
*a = *b;
 *b = temp;
  }
```

This is an example to show the usefulness of pointer dereferencing used to access variables, which otherwise cannot be.

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>
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));
}
```

Pointer Arithmetic

```
#include <stdio.h>
main()
{
 int i, n;
 int smallest;
 int a[50];
 int *p;

 scanf("%d",&n);

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

 p=a;

 smallest=*p;
 p++;
 for(i=1;i<n;i++)
 {
  if(smallest>*p)
    smallest=*p;
  p++;
 }
printf("The smallest element is %d\n",smallest);
}
```

Pointer Arithmetic

```
#include <stdio.h>
main()
{
 int i, n;
 int smallest;
 int a[50];
 int *p;

 scanf("%d",&n);

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

 p=a;

 smallest=*p;

 for(i=1;i<n;i++)
 {
  if(smallest>*(++p))
    smallest=*p;
  }
printf("The smallest element is %d\n",smallest);
}
```

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;