POINTERS

Friday, 5. August 2022 21:58

* INTRODUCTION TO MEMORY ADDRESS

```
int n=5;
char m='A';
```

100 & 200 are here address refrence.

* REFRENCING AND DEREFRENCING OPERATORS

```
int n=5
printf("%d", &n); -> 1000
printf("%d", *&n); -> 5
```

1000 -> Address or refrence

* Address would be always a whole number which OS decide.

*&n ≅ n

&	*
* Address of operator	* Indirection operator
* Refrencing operator	* Derefrencing operator
* Unary operator	* Unary operator
* & -> variable	* * -> address

```
int n=5;
&n =7;  => error
```

* &n is not a variable
* &n is just a way to represent address of variable n. &n ≅ 1000
* &n is considered as constant value

```
int n=5;
int *j;
j= &n;
```

* j is a pointer variable

```
printf("%d %d %d", &n, j, *&n);   // 1000 1000 5
printf("%d %d %d", *j, n, &j);    // 5 5 2000
```

*j ≅ n

* WHAT IS POINTER?

Pointer is a variable which contains address of another variable.

* SIZE OF POINTER

```
int a, *p;
char b, *q;
double c, *r;
```

* size of ordinary variable depends on the datatype.
* size of pointer variable doesn't depend on its datatype.

BASE ADDRESS

```
int a, *p;      // 1001 1002 1003 1004 -> BA
char b, *q;     // 2000 -> BA
double c, *r;   // 3001 3002 ... 3007 -> BA
```

BA -> base address

* Pointer variable always contain base address.

DATATYPE OF POINTER

```
int a, *p;      // 1000
char b, *q;     // 2000
double c, *r;   // 3000
```

p = &b *p ≅ b

char *q; q is not going to store character const, but its going to store address of char variable.

EXTENDED CONCEPT OF POINTERS

```
int n=5, *p, **q, ***r;
p = &n;
q = &p;
r = &q;
printf("%d %d %d %d", p, &r, *q, &n);      // 1000 4000 1000 1000
printf("%d %d %d %d", **r, &q, **&p, n);   // 1000 3000 5 5
printf("%d %d %d %d", ***r, **q, *p, **&p); // 5 5 5 5
```

n (1000) = 5, p (2000) = 1000, q (3000) = 2000, r (4000) = 3000

POINTERS ARITHMETIC

```
int a, b, *p, *q;
p= &a;
q= &b;
```

p+1 = 1004
p+2 = 1008
p+5 = 1020

p+q, p*q, p/q, p*5, p/4 => error
why error? because we are not getting any meaning by doing these operation.

p+2, p-5, q-p } valid

p+i = add of p + size(type of pointer)*i

SWAP

```
int main() {
int a,b;
printf("Enter two numbers");
scanf("%d %d", &a, &b);
   // call a function to swap
printf("%d %d", a,b);
return 0;
}
```

a = 10 (1000), b = 20 (2000) swap(&a, &b)

```
void swap (int *p, int *q)
{
  int t= *p;
  *p = *q;
  *q = t;
}
```

WHY WE USE & in scanf()?

scanf("%d", &n)

scanf(, *p)

POINTERS AND ARRAYS

```
int a[5];     // 0 1 2 3 4 ; 1000 1001 1002 1003 1004
int *p;
p= &a[0];
fun(a);
fun(int *p)
```

p+0	1000	&a[0]
p+1	1004	&a[1]
p+2	1008	&a[2]
p+i		&a[i]
*(p+0)		a[0]
*(p+1)		a[1]
*(p+2)		a[2]
*(p+i)		a[i]

a[i] -> *(a+i) -> *(i+a) -> i[a]

a is constant a = ~~ ✗
p is variable p = ✓

a+i p+i

p[i] -> *(p+i) -> *(i+p) -> i[p]

POINTERS AND STRINGS

```
int l;
char str[10];
fgets(str, 10, stdin);
l= length(str);

int length(char *p)
{
    int i;
    for( i=0; *(p+i); i++);
    return i;
}
```

ARRAYS OF POINTERS

```
int *p[4];
int a[5], b[6], c[3], d[8];
p[0] = &a[0]
p[1] = &b[0]
p[2] = &c[0]
p[3] = &d[0]
input(p)

void input(int **ptr)
```

int *p[4] : 100 | 200 | 300 | 400 (1000, 1008, 1016, 1024)

a (100), b (200), c (300), d (400)

POINTERS TO ARRAY

int (*p)[4]; p is a pointer to an array of int type with 4 blocks.

∴ p+1 means pointing next array with 4 block.

p: 100, 16 bytes

so address of (p+1) would be 116

* Difference b/w int (*p)[4] & *p
  - int (*p)[4] ⇓ points an array
  - *p ⇓ points an integer in any datatype

```
int (*p)[4];
int a[5][4];
p = a;
   a[2][2] = 10
   *(p+2)[2] = 10
 *(*(p+2)+2) = 10
 *[p[2]+2] =10
   p[2][2] = 10
```

	0	1	2	3
0	100	104	108	112
1	116	120	124	128
2	132	136	140	144
3	148	152	156	160
4	164	168	172	176

WILD POINTER

An uninitialized pointer is a wild pointer.

```
void f1() {
int *p; -> wild pointer
 *p =5; -> illegal use of pointer
}
```

NULL POINTER

* A pointer containing NULL (special address) is known as NULL pointer.
* It is a pointer containing NULL, we consider it as if it is not pointing to any location.
* As a safe guard to illegal use of pointers. You can check for NULL before accessing pointer variable.

```
int *p = NULL;
if (p != NULL)
{
    *p = 5;
}
```

DANGLING POINTER

A pointer pointing to a memory location that has been deleted is called dangling pointer.

```
int *f1()
{
    int a;
    ....
    return &a;
}
void fun()
{
    int *p;
    p= f1() -> dangling pointer
    *p = 5; -> illegal use of pointer
}
```

```
void f1()
{
  int *p;
  {
    int n;  -> Scope and life of n is limited to the block
    p = &n;
  } -> After this line p becomes dangling pointer
  *p =5 -> illegal use of pointer
```

```
void f1()
{
  int *p;
  p= (int *) malloc(sizeof(int));
  free(p); -> p becomes a dangling pointer
  *p =5; -> illegal use of pointer
```

VOID POINTER

* void pointer is a generic pointer that has no associated data type with it.
* void pointer can hold address of any type.

```
void *p;      |  void *p;
int n;        |  float y;
p= &n         |  p = &y;
```

void pointer can not be dereferenced.

*p=5; *p = 3.5f; → error

However void pointer can be derefrenced using typecasting.

(int)p = 5 | *(float*)p = 3.5f;