C - 3

- Pointer expressions as L-values or R-values
- How to pass back values from functions?
- Casting
  - To simulate subtyping
  - Unsafe capabilities
- Pointer arithmetic
- Input with scanf()

Pointer Expressions

What happens here? Legality of usage depends on type declaration of pointer.

```
int j = 5; int h = 10;
int *k, **n, *l;
```

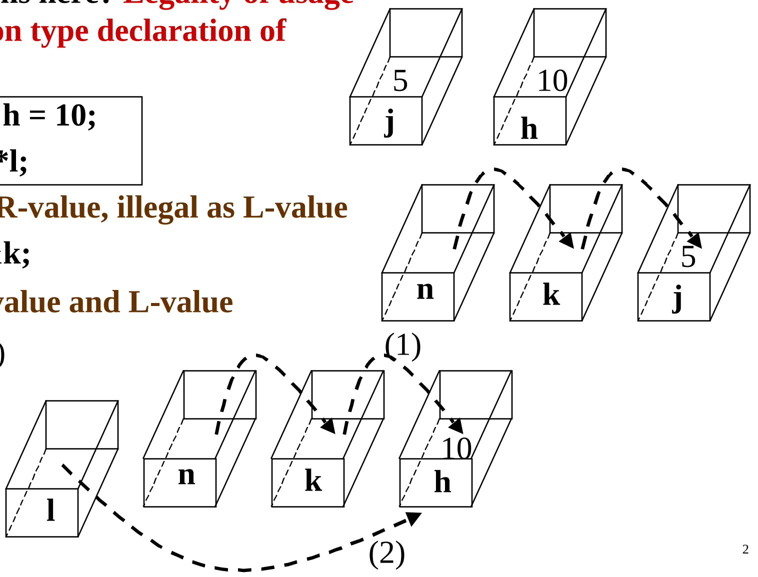
&j, legal as R-value, illegal as L-value

k = &j; n=&k;

*n, legal R-value and L-value

*n = &h; (1)

l = *n; (2)

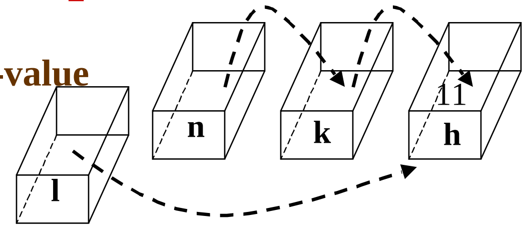


Pointer Expressions

int *k, **n, *l;

**n, legal R-value, legal L-value

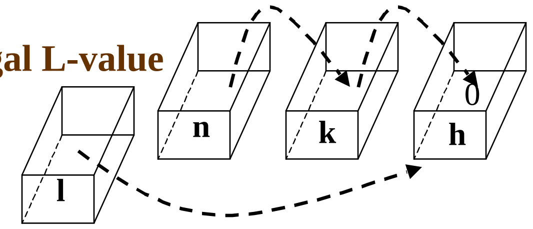
**n = **n+1;



*k, legal R-value, legal L-value

h = *k - 2;

*k = 0;



Side Effects in Functions

- All parameter passing in C is call-by-value
  - Means parameter values are copied into a called function but NOT copied back out at return
- To accomplish side effects, need to use pointer valued parameters
  - Address of actual variable is passed into the function
  - Variable is always accessed indirectly through the corresponding pointer parameter

ret_vals.c

Example

```
main(void)
{       int i,j;
        i= 1; j = 2;
        printf("%d %d\n",i,j);
        j= incr(&i);
        printf("%d %d\n",i,j);
}

int incr(int *a)
{       int z;
        if ((*a)%2 != 0) {z = 1; (*a)++;}
        else z= 0;
        return z;
}
```

```
remus!c> a.out
1 2
2 1
```

incr() increments its argument by 1 and then returns 1 if the value of its parameter was odd.

employee.c

Example

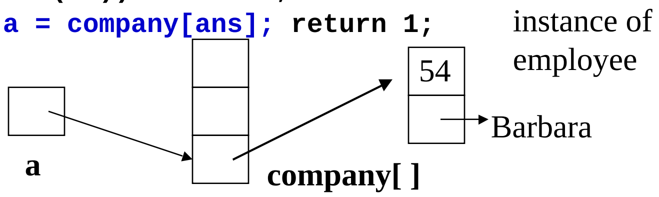
```
#include<stdio.h>
/*this makes these definitions and variables globals*/
/*this is a user-defined type in C*/
typedef struct cell employee;
struct cell{
  int age;
  char *name;
  employee *next;
};
employee *company[2];/*defines an array of pointers to
  employee's*/
/*each element of this array points to a struct cell*/
```

Example

```
main(void)
{ employee *z;
/* create storage for employee records*/
  company[0] = (employee *)malloc(sizeof(employee));
  company[1] = (employee *)malloc(sizeof(employee));
/* initialize company array */
  (company[1])->age = 54;
  (company[1])->name = "Barbara Ryder";
  (company[0])->age = 28;
  (company[0])->name = "Beth Ryder";
  if (find_over49(&z) != 0) printf(" %s \n",z->name);
}
```

Example

```
/* z is of type employee *, so address(z) is value of
   an employee ** var*/
int find_over49(employee **a)
{
  int ans, i;
  ans = -1;
  for (i=0; i<2; i++)
    if (((company[i])-> age) > 49){ans = i;break;}
  /*example of multiple return values from a c
  function*/
  if (ans == (-1)) return 0;
  else { *a = company[ans]; return 1;
  };
}
```



Example

```
/* a sample run
128 remus!c> gcc employee.c
129 remus!c> a.out
 Barbara Ryder
*/
```

Casting

- **Safe uses of casting**
  - **For pointers returned from *malloc***

    p = (int *) malloc (4);
  - **For simulating subtyping safely in C**

```
struct  s{            struct  t{
   int  a;              int  a;
   int  b;             int  b;
   double  c;          }
}
```

s is like a subtype of t because it has same fields as t plus an extra field.

newcasting.c

```
/*example due to satish chandra of bell labs
  this is a use of casting that is like subtyping*/
#include<stdio.h>
typedef struct{
  int x,y;
 }point;
typedef enum{
    RED, BLUE
  }color;
typedef struct{
    int x,y;
    color c;
  }colorpoint;
void translateX(point *p, int dx){
    p ->x += dx; /*translates x co-ordinate by 1*/
}
```

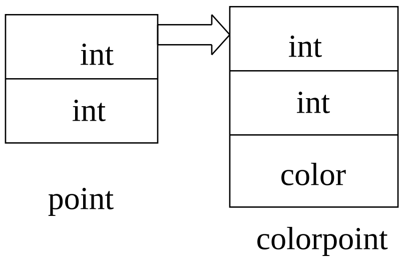
newcasting.c

```
main(){
   point p;
   colorpoint cp;
/* initialize p to (0,0) and cp to (1,1) */
   p.x = 0;
   p.y = 0;
   cp.x = 1;
   cp.y = 1;
   cp.c = RED;
   printf(" p= %d,%d cp= %d,%d\n",p.x,p.y,cp.x,cp.y);
/* move x co-ordinate by 1 for both points*/
   translateX(&p, 1);
   translateX((point *) &cp, 1);
   printf("after translation, p= %d,%d cp= %d,%d\n”,
      p.x,p.y,cp.x,cp.y);
 }
```

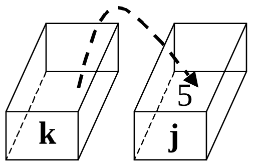
Output

```
/* output resulting
20 scherzo!c> gcc newcasting.c
21 scherzo!c> a.out
 p= 0,0 cp= 1,1
after translation, p= 1,0 cp= 2,1
22 scherzo!c>
*/
```

Why the cast works?



Pointer Arithmetic



int *k; k=&j;

(*k+1), legal R-value, illegal L-value

(*k+1) means ((*k)+1)

h = *k +1;

***(k+1) legal (but not meaningful) L-value, legal R-value**

/*need to know layout of storage to see to what (k+1) points, to the byte that is 4 bytes beyond the L-value of k, since adding 1 is like adding storage for 1 int (4 bytes)*/

k++ has same properties when used with *

newpointerarith.c

```
struct person{
  int age;
  int socsecnum;
  int phoneno;
};

main( )
{ int a[5], j;
  int *pa, *pb;
  struct person people[3];
  struct person *zz;

  for (j = 0; j < 6; j++)
      a[j] = j;/* initialize a */
```

a, array of ints;
people, array of struct persons

newpointerarith.c

```
for (j = 0; j < 6; j++)
  printf(" %d",a[j]);
printf("\n");
pa = &a[0];
for (pb = a; pb < &a[6]; pb++)
{  printf(" %d %d", *pa,*pb);
   pa = pa + 1;
}
printf("\n");
```

```
33 1 scherzo!c> a.out
0 1 2 3 4 5
0 0 1 1 2 2 3 3 4 4 5 5
```

newpointerarith.c

```
   j=0;
   while (!feof(stdin) && j<3)
   {   scanf("%d%d%d", &(people[j].age),
   &(people[j].socsecnum), &(people[j].phoneno));
       printf("output with array elements %d %d %d\n",
              (people[j]).age,(people[j]).socsecnum,
              (people[j]).phoneno);
       j++;
   }/* can I use people[j]->age? Why or why not? */
/* output:
   52 999 3699 26 111 5430 24 222 3361 --I typed this at the
   output with array elements 52 999 3699 terminal
   output with array elements 26 111 5430
   output with array elements 24 222 3361
*/
```

newpointerarith.c

```
/* wow. this works! */
   printf("\n");
   zz = people;/*remember people is an array*/
   for (j = 0; j<3; j++)
     { printf("output with pointer %d %d %d\n",
              (*zz).age,zz->socsecnum,zz->phoneno);
       zz = zz + 1;
     }
   printf("\n");
/*  output:
   output with pointer 52 999 3699
   output with pointer 26 111 5430
   output with pointer 24 222 3361  */
```

Don’t do this!

```
/* c is wonderful; look at it breaking strong typing
   easily*/
   zz=(struct person *) &j;
   printf(" %d\n",zz->age);

/* output
   3
--so c actually interprets the value of j (an int) as
   the value of the first field of a person struct, age
   (an int)*/
```

Pointers and Structs

- **Field access uses “.” operator**

  `e.g., (people[j]).age`
- **-> is shorthand for * .**

  for listcell * p, p -> num means (*p). num
- **that’s why this works when zz is a pointer:**

  `(*zz).age,zz->socsecnum`

C Functions

- Prototype - often found in a *.h (header)file; used for compiler for type checking function calls

```
int mult(int k, int n);
```

- Definition - contains code of the function

```
 int mult(int k,int n){
   return k*n;}
```

- Invocation

```
int j,n=50; … j = mult(4,n);
```

Strings

- Strings are arrays of chars, as in Pascal
  - Because of special relation between array names and pointers, often see strings defined as char *
- String library contains useful functions

  int strcmp(char *s,char *t): returns value < 0 if s is less than t (in lexicographic order), 0 if s == t and >0 if t is less than s.

  char * strcpy(char *s, char *t): copies the string pointed to by t into the string pointed to by s; to work this needs t to be declared big enough to store the string.

Strings

```
int strcpy (char *s, char *t){
 for (; *t != ‘\0’; s++,t++)
    *t = *s;
}
```

On return, t points to a copy of the string pointed to by s. Means you had to have allocated storage for t BEFORE calling strcpy().

Q: What is the difference in meaning of

t = s versus *t = *s ??

Strings and Chars

- **Ascii collating sequence encodes characters representing letters as consecutive integers; therefore this works:**

```
char s,t;
scanf( “ %c %c “, &s, &t);
if (s < t) ….
```