

More C

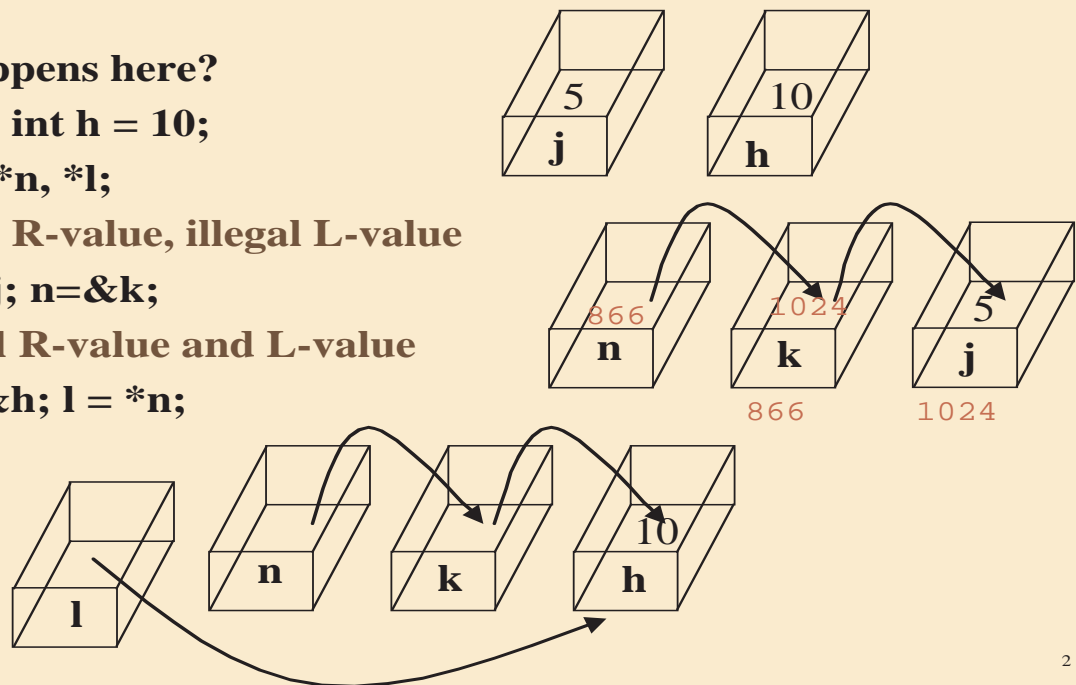
- **Pointer expressions as L-values or R-values**
- **More on pointer problems**
- **Casting**
 - To simulate subtyping
 - Unsafe capabilities
- **Pointer arithmetic**
- **Input with scanf()**
- **Arrays**
- **Strings**

1

Spr2001, BGR/AB

Pointer Expressions

- **What happens here?**
`int j = 5; int h = 10;`
`int *k, **n, *l;`
`&j, legal R-value, illegal L-value`
`k = &j; n=&k;`
`*n, legal R-value and L-value`
`*n = &h; l = *n;`



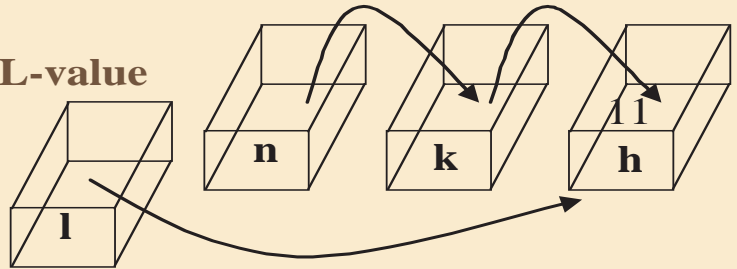
2

Spr2001, BGR/AB

Pointer Expressions

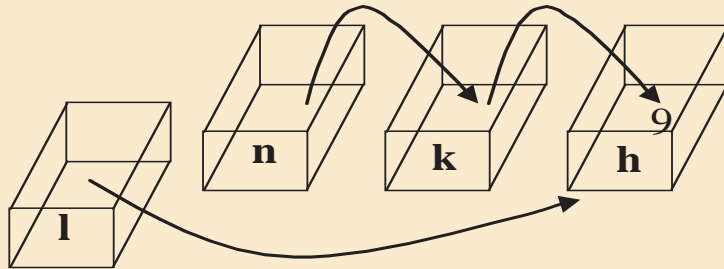
****n, legal R-value, illegal L-value**

$h = **n + 1;$



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

$h = *k - 2;$



3

Spr2001, BGR/AB

More Problems with Pointers

- **Dangling pointer**
 - Storage pointed to is freed, but pointer is not set to NULL
 - Then you are able to access storage whose values are not meaningful
- **Garbage**
 - Pointer itself is freed (perhaps by execution going out of its declaring scope) but heap locations pointed to are not freed
 - Then, there is no way to access this heap storage

4

Spr2001, BGR/AB

Managing the Heap

- **Explicit allocation and deallocation**
 - Garbage and dangling pointers
 - Memory leaks: not freeing storage appropriately

Casting

- **Safe uses of casting**
 - For pointers returned from *malloc*
*p = (int *) malloc (4);*
 - For simulating subtyping safely in C

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

```
struct t{  
    int a;  
    int b;  
}
```

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 dx*/
}
```

7

Spr2001, BGR/AB

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);
}
```

8

Spr2001, BGR/AB

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>
*/
```

9

Spr2001, BGR/AB

Pointer Arithmetic

```
int j ;
int *k;
k=&j; /* k is legal L-value */
*k+2 means ((*k)+2) == 5+2 == 7
==(*k+2): legal R-value (h = *k+2;), illegal L-value
(because its type is int)
```

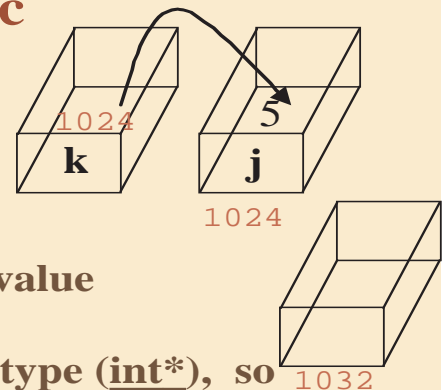
```
k+2 ==1024 + 2*sizeof j == 1024+2*4 = 1032 has type (int*), so
it is a legal (but not meaningful) L-value
```

```
/*need to know layout of storage to see to what (k+1)
points: the byte that is 8 bytes beyond the L-value of k,
since adding 2 is like adding storage for 2 int (4 bytes
each)*/
```

```
*(k+2) == *(1032) == ??
```

```
has type int: so again an illegal L-value
```

```
k++ has same properties when used with *
```



10

Spr2001, BGR/AB

Pointers and Arrays

- An array name is considered pointer to first element
 - **a** is pointer to **a[0]**
 - **pa = &a[0]** and **pa=a** mean the same thing
 - **a+1** means L-value of **a[0]** plus as many bytes as are needed to store value of elements of **a**'s type
 - Pointer arithmetic is an address calculation with respect to the underlying architecture
- An array name is not a variable
 - **a++** and **a = pa** are **ILLEGAL!**

Arrays & pointer examples

```
int array[10];
int *ap; *ap = array +2;
```

Express each of the following expressions in terms of array[]

```
ap // = &array[2]
*ap // = array[2]
ap[0] // = (*array + (0)) = *ap+0 = array[2]
ap + 6 // = (array + 2+6) = &array[8] pointer arithmetic
*ap + 6 // = (*ap) + 6 = array[2] + 6 // an integer
*(ap + 6) // = *(ap+6) = array[8] //an int
&ap // =?? No expression in terms of array
ap[-1] // = array[1]
ap[6] // = array[8]
ap[8] // = array[10] !past end of declarations so it points who-knows where !!!
```

Array/pointer processing

```
for(a=0; a<10; a++){  
    array[a] = 0;  
}
```

Requires computing $\text{array} + a * \text{sizeof array}$; this cannot be done at compile time (a is not known) so it requires run-time multiplication;

```
int *ap;  
for(ap=array; ap < array + 10; ap++){  
    *ap = 0  
}
```

The `ap++` only requires computing $+ 1 * \text{sizeof array}$, which is known at compile time. (`*ap` is just one dereference).

So the later saves on integer multiplication every time through the loop.

Spr2001, BGR/AB

newpointerarith.c

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

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

```
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 */
```

Spr2001, BGR/AB

newpointerarith.c

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

```
int a[5], j;
int *pa, *pb;
```

These two loops print the same elements.

12

Spr2001, BGR/AB

newpointerarith.c

```
int j=0;
while (!feof(stdin) && j<3)
{scanf("%d%d%d", &(people[j].age),
    &(people[j].socsec), &(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
*/
```

```
struct person{
    int age;
    int socsec;
    int phoneno;
} people[3],*zz;
```

13

Spr2001, BGR/AB

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->socsec, 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 */
```

14

Spr2001, BGR/AB

Don't do this!

```
/* c is wonderful; look at it breaking strong typing
easily*/
int j=3;
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)*/
```

15

Spr2001, BGR/AB

Strings

Strings are arrays of characters = pointers to
By convention terminated by special character NUL
(the length of the string should not include t

```
char greet[] = "hi";
    results in greet[0]='h', greet[1]='i', ???greet[2]='\0'??? (books disagree)
size_t
strlen(char const *str){
    int I;
    for (I=0, *(str+I)!=NUL , I++){};
    return I; //starting from 0 leads to omitting \0
}

char
*strcpy( char *destination, char const *source);
char *strcat(char *dest, char *src);
int  strcmp(char const *s1, char const *s2);
```

Spr2001, BGR/AB

Summary: why pointers in C?

- Call-by-value parameter passing means this is the only way you can affect params
- Pointers can be passed efficiently as arguments. Since arrays are pointers, it allows them to be passed in by value without copying. Also, this allows arrays to be returned.
- char pointers (arrays of characters) are the standard way of providing *strings*: a non-implemented data type
- **char* array[10]** permits each entry (**array[i]**) to be of different length while **char array[10][20]** makes all strings have length 20
- Makes possible recursive structures
- Allows space to be allocated dynamically, at run time, usually for data structures that can grow or shrink

Spr2001, BGR/AB

Pointers to Functions

- `int foo(int x) { ...}`
`int (*pf)(int x); //pf points to a function of type int (int)`
`pf = &foo;`
(beware `int *pf (int x)` is parsed as `(int*) pf (int x)`)
- **Function names are always converted to function pointers (!),** so `f(5)`, `(*pf)(5)`, and `pf(5)` all do the same thing.
- **Typical use:**
`Node *`
`Search_list(Node *node, void const *lookFor,`
`int (*compare) (void const *, void const *)) {`
`while (node!=NUL){`
`if (*compare)(&(amp;node-->value), lookFor) break;`
`else node = node --> link; }`
`return node;`
`}`

Spr2001, BGR/AB

Pointers to Functions

- **So if you define**
`int compare_ints (void const *a, void const *b) {`
`if (* (int*) a < * (int*)b) return 0; else return 1; }`
you can invoke
`Node *where;`
`where = Search_list(head, desired_value, compare_ints)`
- **In general, use `void *` when you want a *generic* pointer** (because it is automatically cast into the type of the argument, **IF** it is also a pointer)

Spr2001, BGR/AB