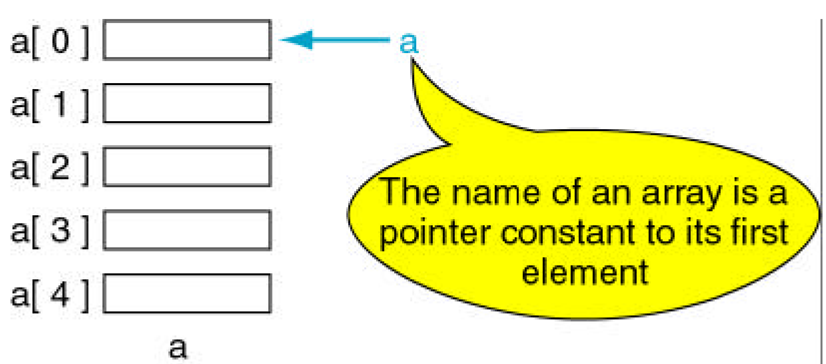
Arrays, Pointers and Arithmetic

COMP 1402/1002

The Name of an Array



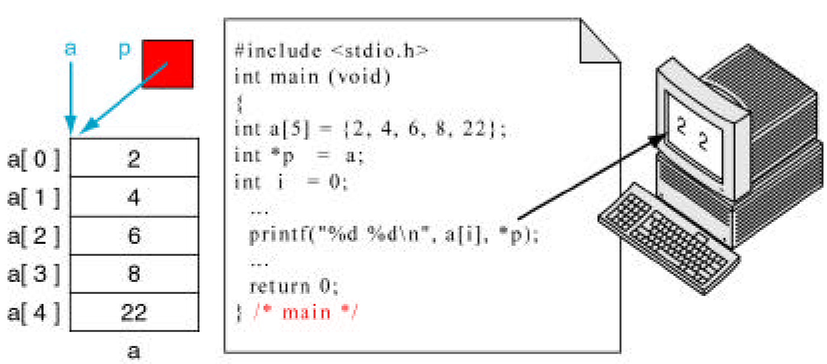
The Name of an Array

```
/* the name of an array is a pointer
  value */
int a[5];
printf("%p %p", &(a[0]), a);
```

The two values printed are identical

The value depends on memory

Using Array Names



The Meaning of Indices

a[0]	Contents at **a**
a[4]	Four elements after address **a**
a[i]	**i** elements after address **a**
a[-1]	One element before address **a**

Computes appropriate address
Automatically dereferences (no need for *****)

Access Array with any pointer!

```
#include <stdio.h>
int main (void)
{
  int a[5] = {2, 4, 6, 8, 22};
  int *p;
  p = &(a[1]);
  printf(“%d %d\n”,a[0],p[-1]);
} /* main */
```

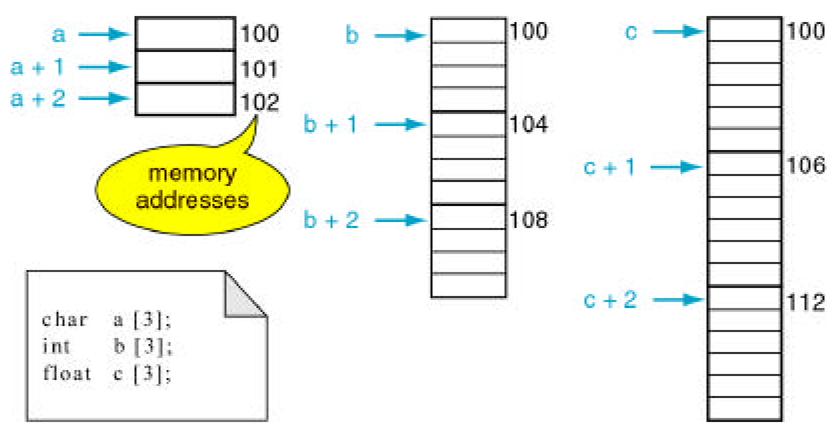
Pointer Arithmetic and Arrays

Consider the following code:

```
char a[3];
int b[3];
float c[3];
char *pa = a + 1;
int *pb = b + 1;
float *pc = c + 1;
```

What is the meaning of `+1` in each case???

Different Types and Pointer Arithmetic



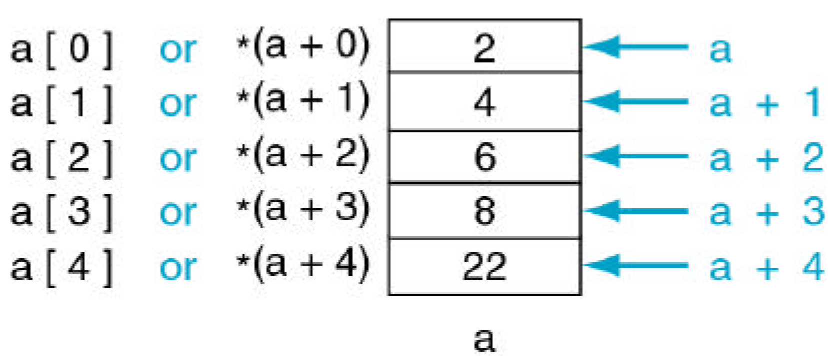
Pointer Arithmetic

C “knows” to add the correct number of bytes

Thus a[1] is **always** equivalent to *(a+1)

This is why type conversion is needed for pointers

Dereferencing arrays



What the C compiler does

`a[i]` is the pointer arithmetic **`(*a+i)`**

`i=1` results in the following “assembler code”:

```
  a + sizeof(one element)
```

Otherwise the “assembler code” is:

```
  a + i * sizeof(one element)
```

Searching With Array Indices

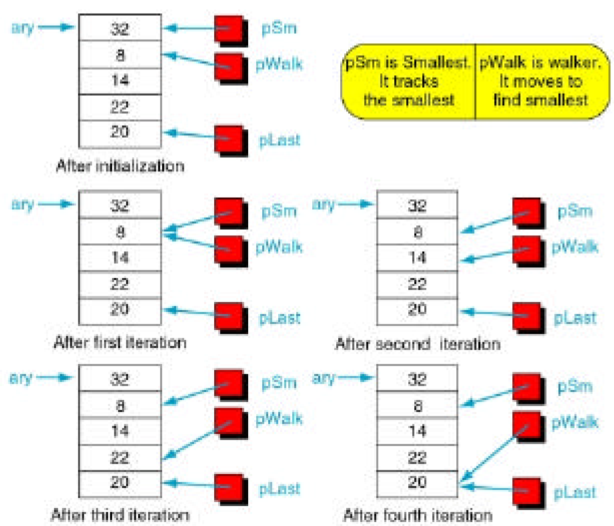
```
/* ary is an array of arySize */
int walk, last, sm;

last = arySize –1;
sm = 0;
for(walk=1; walk<=last;walk++) {
  if (ary[walk] < ary[sm])
     sm = walk;
}
```

Searching With Pointers

```
/* ary is an array of arySize */
/* pwalk, plast, psm are pointers to type
  in ary */
plast = ary + arySize –1;
psm = ary;
for(pwalk=ary + 1; pwalk<=plast;pwalk++){
  if (*pwalk < *psm)
     psm = pwalk;
}
```

Using Pointers



Comparing Pointers

`p1 >= p2`

`p2 < p3`

`p1 != p2`

`ptr1 == ptr2`

`if(ptr==NULL)` (same as) **`if (!ptr)`**

`if(ptr!=NULL)` (same as) **`if (ptr)`**

Passing Parameters

Consider the following code:

```
void f(int x) {x = x+1;}
int main (void)
{
  int a=3;
  f(a);
  printf("%d",a); /* prints 3 */
  return 0;
}
```

Passing Parameters

Consider the following code:

```
void f(int *x) { x = x+1;}
int main (void)
{
  int a[]={1,2}, *pa=a;
  f(pa);
  printf("%d",*pa); /* prints 1 */
  return 0;
}
```

Passing Parameters

Consider the following code:

```
void f(int **x) { *x = *x+1;}
int main (void)
{
  int a[]={1,2}, *pa=a;
  f(&pa);
  printf("%d",*pa); /* prints 2 */
  return 0;
}
```

C is pass by value!!

C even passes the pointer by value

The second code passes a copy of pointer

The third code passes the pointer to pa

Allows manipulation of pointers

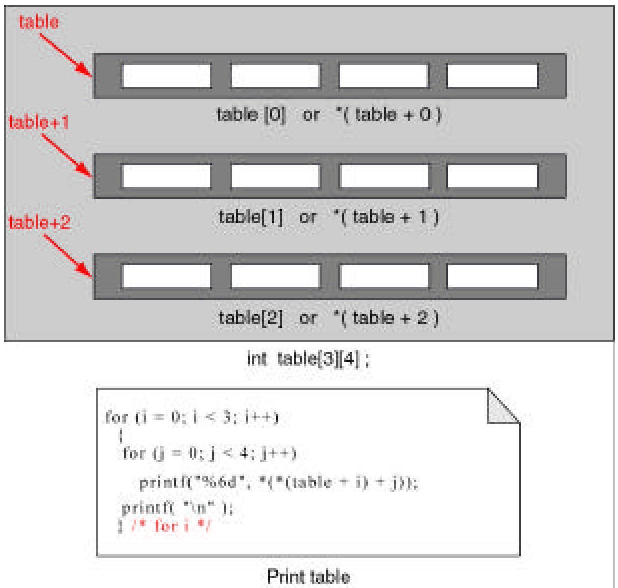
What about 2D arrays?

A[i] is equivalent to *(a + i)

Q. What is b[i][j] equivalent to?

Ans. *(b[i] + j)
((b+i) + j)

2D Arrays



Passing Arrays as Parameters

The following are identical to C

```
int func1 (int array[]);
int func1 (int *array);
```

But not to the human reader!

Passing High Dimension Arrays

```
float funct (int big[][3][5]);
```

Suppose we have the following code:

```
int ***p;
p = big + 1;
```

How many bytes to add?

We only know because of the 3 and 5!

Passing High Dimension Arrays

```
float funct (int big[][3][5]);
...
int ***p;
p = big + 1;
```

15 bytes * 4 bytes per integer

Complex Declarations

What do these declarations mean:

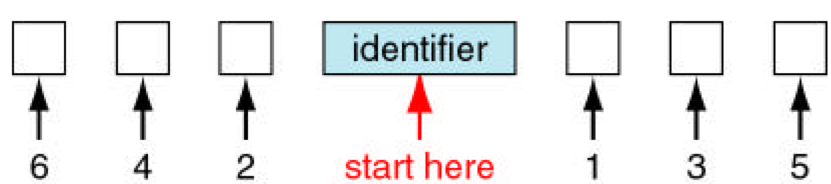
```
float x;
float * p;
float arrayOne[5];
float arrayTwo[4][5];
float * arrayThree[5];
float (* arrayFour)[35];
float const * a;
float * const b;
const float * c;
```

Start at Identifier…

Start at Identifier

Alternate right then left

Brackets take priority over above rule



Complex Declarations

```
int x;
```

x is a int

```
int * p;
```

p is a pointer to an int

```
int aOne[5];
```

aOne is an array of ints

Complex Declarations

```
int aTwo[4][5];
```

aTwo is a an array of ints

```
int * aThree[5];
```

aThree is a an array of pointers to ints

```
int (* aFour)[35];
```

aFour is a pointer to an array of ints

Complex Declarations

```
int * const b;
```

b is a constant pointer to an int

```
int const * a;
```

a is a pointer to a constant int
(an unreliable declaration!!)

Memory Allocation

Types of Memory Allocation:
Static – done at compile time

Dynamic – get memory during runtime

COBOL, FORTRAN are Static only!!
C has two types of Dynamic: Stack and Heap

Dynamic Memory in C

Stack:

Declare arrays with variables of larger scope

Problem:

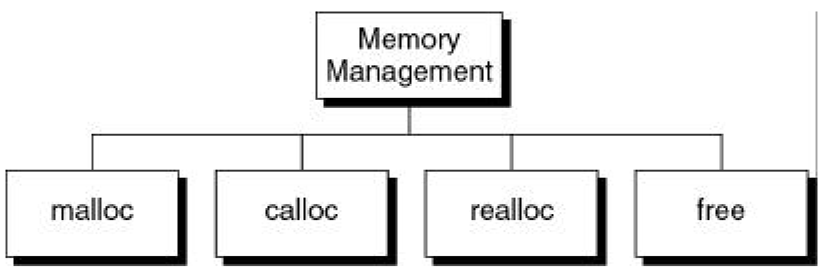
Only good in the current scope!

```
float * func(int n) {
  float array[n]; /* good */
  …
  return array; /* severe mistake */
}
```

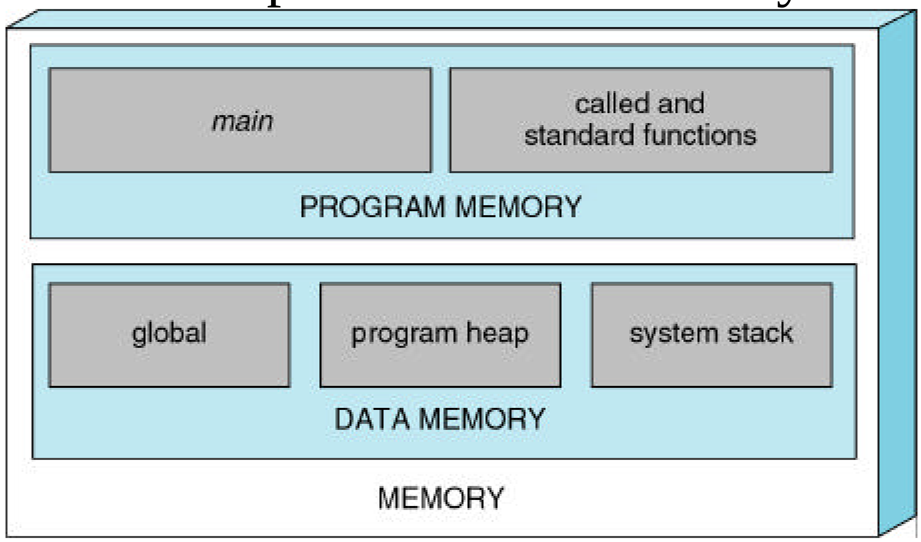
Heap Allocation

Ask the system for a portion of the memory

`stdlib.h` has 4 functions to deal with this



Conceptual View of Memory



`malloc`

```
void * malloc (size_t size);
```

`malloc` allocates number of bytes specified in **`size`**

`size_t`

an integer big enough to hold the max address

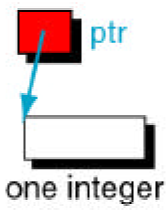
Normally this is an **`unsigned int`**

Casting and Error Check

```
pFloat = (float *)malloc (sizeof(float));
```

Cast to `(float *)` not required by ANSI

Error check also recommended:




```
if (!(ptr = (int *)malloc(sizeof(int))))
  /* No memory available */
  exit (100) ;
/* Memory available */
...
```

`calloc`

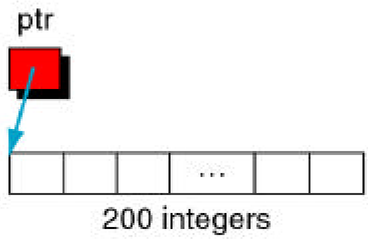
```
void * calloc (size_t count,
               size_t elem_size);
```

`calloc` allocates **`count`** elements of **`elem_size`**

Returns consecutive elements!

If unsuccessful returns **`NULL`** (like **`malloc`**)

calloc




```
if (!(ptr = (int *)calloc (200, sizeof(int))))
   /* No memory available */
   exit (100) ;

/* Memory available */
...
```

realloc

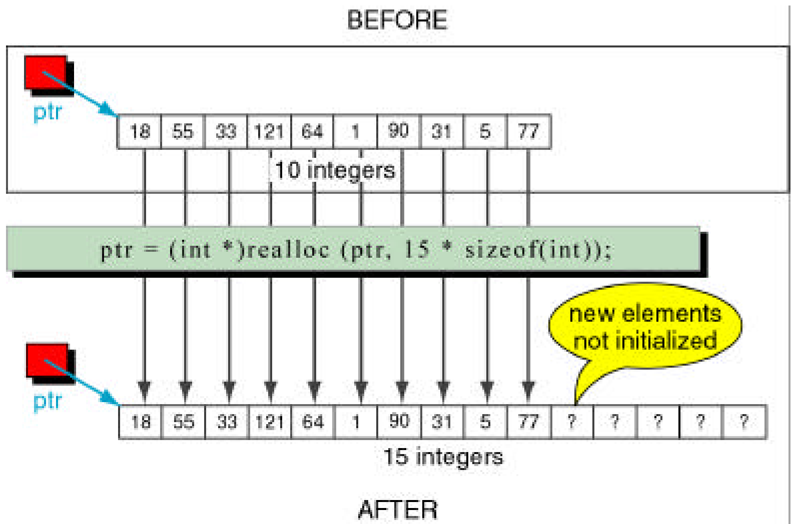
Resizes old block

Or declares new, copies old and deletes old.

Often expensive operation

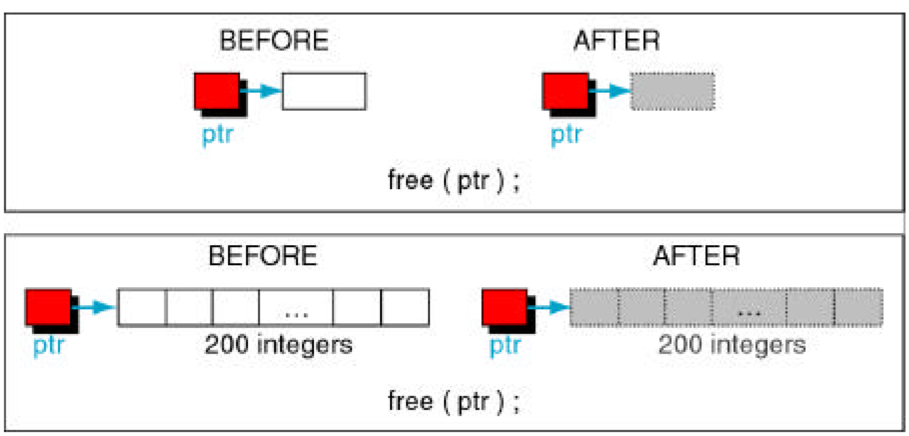
```
void * realloc (void * ptr,
                size_t new_size);
```

realloc



free

```
void free(void *ptr);
```



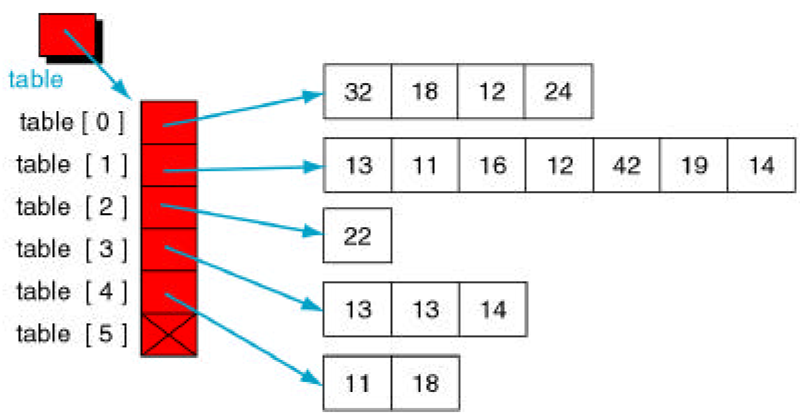
`calloc` & Array of Pointers

The next slide details the full power
of an array of pointers

`int table[4][5];`

`table` is an array of pointers
each pointing to an array of the same size

`calloc` removes that restriction




```
table = (int **)calloc (rowNum + 1, sizeof(int*));

table[0] = (int*)calloc (4, sizeof(int));
table[1] = (int*)calloc (7, sizeof(int));
table[2] = (int*)calloc (1, sizeof(int));
table[3] = (int*)calloc (3, sizeof(int));
table[4] = (int*)calloc (2, sizeof(int));
table[5] = NULL;
```