

POINTERS

- Pointers are variables that contain *memory addresses* as their values.
- A variable name *directly* references a value.
- A pointer *indirectly* references a value.
Referencing a value through a pointer is called *indirection*.
- A pointer variable must be declared before it can be used.

Concept of Address and Pointers

- Memory can be conceptualized as a linear set of data locations.
- Variables reference the contents of a locations
- Pointers have a value of the address of a given location

ADDR1	Contents1
ADDR2	
ADDR3	
ADDR4	
ADDR5	
ADDR6	
*	
*	
*	
ADDR11	Contents11
*	
*	
ADDR16	Contents16

Seeing the Value of a Pointer

- Address
 - `printf(“%x”,ptr);`
 - Prints a hexadecimal number – the address that ptr points to
 - `printf(“%u”,ptr);`
 - Prints a unsigned number – the address that ptr points to

POINTERS

- Examples of pointer declarations:

```
FILE *fptr;
```

```
int *a;
```

```
float *b;
```

```
char *c;
```

- The **asterisk**, when used as above in the declaration, tells the compiler that the variable is to be a pointer, and the type of data that the pointer points to, but **NOT** the name of the variable pointed to.

Use of & and *

- When is & used?
- When is * used?
- & -- "address operator" which gives or produces the memory address of a data variable
- * -- "dereferencing operator" which provides the contents in the memory location specified by a pointer

How to declare pointer variables

- Format: type *name [= object address]
- E.g., char *c;
- E.g., int *p;
- P is variable which is going to store an address in which an integer value is present.
- int **k;
- K is a variable which is going to store an address of (int *) type variable.
- K is a variable which is going to store an address in which an address which points to integer is present.

int a = 3;

a

3

2000

```
printf("%u",&a);  
printf("%d",a);  
printf("%u",&a);
```

int *b = &a;

b

2000

3000

```
printf("%u",&b);  
printf("%u",b);  
printf("%u",&b);  
printf("%d",&b);  
printf("%d",*b);
```

int **c = &b;

c

3000

4000

```
printf("%u",&c);  
printf("%u",c);  
printf("%u",&c);  
printf("%u",&c);  
printf("%u",*c);  
printf("%d",&c);  
printf("%d",&c);
```

int *d = &c;**

d

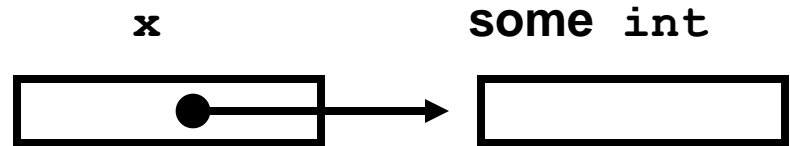
4000

5000

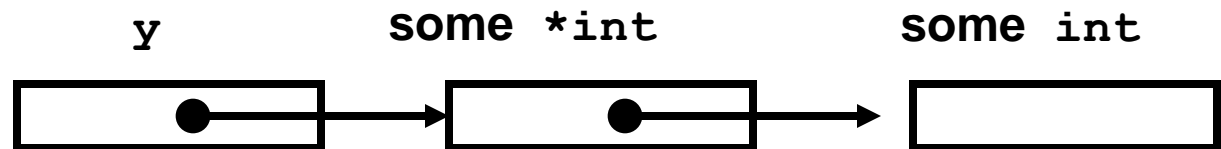
```
printf("%u",&d);  
printf("%u",d);  
printf("%u",&d);  
printf("%u",&d);  
printf("%u",*d);  
printf("%u",&d);  
printf("%d",&d);
```

Pointers to anything

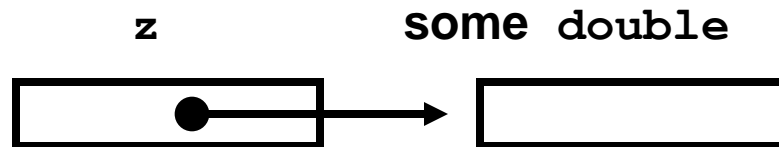
```
int *x;
```



```
int **y;
```



```
double *z;
```



Pointers and Functions

- Pointers can be used to pass addresses of variables to called functions, thus allowing the called function to alter the values stored there.
- We looked earlier at a swap function that did not change the values stored in the main program because only the values were passed to the function swap.
- This is known as "call by value".

Parameter passing (Call by value)

```
#include <stdio.h>
void swap ( int a, int b ) ;
int main ( )
{
    int a = 5, b = 6;
    printf("a=%d b=%d\n",a,b) ;
    swap (a, b) ;
    printf("a=%d b=%d\n",a,b) ;
    return 0 ;
}
```

```
void swap( int a, int b )
{
    int temp;
    temp= a; a= b; b = temp ;
    printf ("a=%d b=%d\n", a, b);
}
```

Results:

a=5 b=6

a=6 b=5

a=5 b=6

Pointers and Functions

- If instead of passing the values of the variables to the called function, we pass their addresses, so that the called function can change the values stored in the calling routine. This is known as "call by reference" since we are referencing the variables.
- The following shows the swap function modified from a "call by value" to a "call by reference". Note that the values are now actually swapped when the control is returned to main function.

Pointers with Functions (example)

```
#include <stdio.h>

void swap ( int *a, int *b ) ;

int main ( )
{
    int a = 5, b = 6;
    printf("a=%d b=%d\n",a,b) ;
    swap (&a, &b) ;
    printf("a=%d b=%d\n",a,b) ;
    return 0 ;
}
```

```
void swap( int *a, int *b )
{
    int temp;
    temp= *a; *a= *b; *b = temp ;
    printf ("a=%d b=%d\n", *a, *b);
}
```

Results:

a=5 b=6

a=6 b=5

a=6 b=5

Arithmetic and Logical Operations on Pointers

- A pointer may be incremented or decremented
- An integer may be added to or subtracted from a pointer.
- Pointer variables may be subtracted from one another.
- Pointer variables can be used in comparisons, but usually only in a comparison to NULL.
- Do not attempt the following operations on pointers.
 - Addition of two Pointers
 - Multiplying a pointer with a number
 - Dividing a pointer with a number

Arithmetic Operations on Pointers

- When an integer is added to or subtracted from a pointer, the new pointer value is changed by the integer times the number of bytes in the data variable the pointer is pointing to.
- For example, if the pointer *valptr* contains the address of a float variable and that address is 7870, then the statement:
$$\textit{valptr} = \textit{valptr} + 2;$$
would change *valptr* to 7874

Example

```
int f (void) {  
    int s = 1;  
    int t = 1;  
    int *ps = &s;  
    int **pps = &ps;  
    int *pt = &t;  
  
    **pps = 2;  
    pt = ps;  
    *pt = 3;  
    t = s;  
}
```

$s == 1, t == 1$

$s == 2, t == 1$

$s == 3, t == 1$

$s == 3, t == 3$

Rvalues and Lvalues

What does = really mean?

```
int f (void) {  
    int s = 1;  
    int t = 1;  
    t = s;  
    t = 2;  
}
```

left side of = is an “lvalue”
it evaluates to a location (address)!

right side of = is an “rvalue”
it evaluates to a value

There is an implicit * when a
variable is
used as an rvalue!

Manipulating Addresses

```
char s[6];  
s[0] = 'h';  
s[1] = 'e';  
s[2] = 'l';  
s[3] = 'l';  
s[4] = 'o';  
s[5] = '\0';  
printf ("s: %s\n", s);
```

s: hello

$A[i]$ in C is just syntactic sugar
for
 $*(A + i)$

Obfuscating C

```
char s[6];  
*s = 'h';  
*(s + 1) = 'e';  
2[s] = 'l';  
3[s] = 'l';  
*(s + 4) = 'o';  
5[s] = '\\0';  
printf ("s: %s\\n", s);
```

s: hello

Fun with Pointer Arithmetic

```
int match (char *s, char *t) {  
    int count = 0;  
    while (*s == *t) { count++; s++; t++; }  
    return count;  
}
```

```
int main (void)  
{  
    char s1[6] = "hello";    The \0 is invisible!  
    char s2[6] = "hohoh";  
  
    printf ("match: %d\n", match (s1, s2));  
    printf ("match: %d\n", match (s2, s2 + 2));  
    printf ("match: %d\n", match (&s2[1], &s2[3]));  
}
```

$\&s2[1]$
 $\rightarrow \&(* (s2 + 1))$
 $\rightarrow s2 + 1$

match: 1
match: 3
match: 2

Condensing match

```
int match (char *s, char *t) {  
    int count = 0;  
    while (*s == *t) { count++; s++; t++; }  
    return count;  
}
```

```
int match (char *s, char *t) {  
    char *os = s;  
    while (*s++ == *t++);  
    return s - os - 1;  
}
```

$s++$ evaluates to s_{pre} , but changes the value of s

Hence, C++ has the same value as C, but has unpleasant side effects.

Arrays and Pointers

- So far, we have looked at pointer variables which were declared with statements like:

```
char *cptr ;
```

```
int *aptr ;
```

- We have also used *pointer constants* when we sent the address of a variable to a function. We did this by placing an ampersand before the variable name. For instance, the address of variable **a** is **&a**, which is a pointer constant.

Arrays and Pointers

- The name of an array without a subscript (index number) is also a pointer constant. When the name is used without the subscript it references the address of element 0 of the array.

<code>int myarray[10];</code>	<code>/* Declare an array */</code>
<code>int *myptr;</code>	<code>/* Declare a pointer (not initialized)*/</code>
<code>printf ("%d\n", myarray);</code>	<code>/* print address of myarray[0] */</code>
<code>scanf ("%d", myarray);</code>	<code>/* get value from keyboard and store in myarray[0] */</code>
<code>scanf ("%d",&myarray[0]);</code>	<code>/* same thing as above */</code>

Arrays and Pointers

```
myptr = &myarray[2];          /* Assign the address of the third  
                               element of myarray to myptr */  
printf("%d", *myptr);         /* Print the value of what myptr is  
                               pointing to, i.e., the value of  
                               myarray[2] */
```

- Note that the * in front of myptr de-references the pointer. That is, it says “use the value in the address that is pointed to.”
- The following shows a small program and its output.

Arrays and Pointers

```
/* Printing array values and addresses */  
#include <stdio.h>  
int main ( )  
{  
    int k;  
    float a[4] = {1000, 2000, 3000, 4000};  
    printf ("k    a    &a[k]    a[k]\n");  
    for (k=0; k<4; k++)  
        printf ("%d %u %u %f\n", k, a, &a[k], a[k]);  
}
```


Running the Program

k	a	&a[k]	a[k]
0	62916	62916	1000.000000
1	62916	62920	2000.000000
2	62916	62924	3000.000000
3	62916	62928	4000.000000

Arrays and Pointers

- We have seen how to pass a pointer for a single valued variable to a function.
- Sometimes we want to pass an entire array to a function. The name of an array without a subscript is a pointer constant that contains the address of element [0] of the array. Therefore, if we pass the array name with no subscript to a function, we are passing a pointer to that array.
- The following program illustrates this.

Arrays and Pointers

```
/* Passing an array to a function */  
#include <stdio.h>  
#include <string.h>  
void myfunct (int , char [ ]);  
int main ( )  
{  
    char name[20] = "Michael J. Miller";  
    myfunct (20, name);  
}
```

Arrays and Pointers

```
void myfunct (int len , char text[ ])
{
    int k ;
    printf ("%d\n", strlen(text)) ;
    for (k = 0 ; k < len ; k++)
        printf ("%c", text[k]) ;
}
```

/*Program Output */

17

Michael J. Miller

Arrays and Pointers

- Given the declaration `int a[3] = { 1, 2, 3 } ;`
 - `a` is a pointer to (the address of) `a[0]`
 - `&a[0]` is a pointer to `a[0]`
 - `a[0]` is the value 1 (`*a` is also the value 1)
 - `&a[1]` is a pointer to `a[1]`
 - `a[1]` is the value 2
 - `&a[2]` is a pointer to `a[2]`
 - `a[2]` is the value 3
 - `&a[0]+1` is a pointer to `a[1]`
 - `&a[0]+2` is a pointer to `a[2]`

Arrays and Pointers

Given the declaration

```
int b[3][3] = {{1,3,5}, {7,9,11}, {13,15,17}};
```

- b is a pointer to b[0][0]
- b[0] is also a pointer to b[0][0]
- b[1] is a pointer to b[1][0]
- b[2] is a pointer to b[2][0]
- *b is a pointer to b[0] (special case)
- *b[1] is the value of b[1][0] (which is 7)
- *b[2] + 1 is the value of b[2][0] + 1 (which is 14)

```
/* Double subscript arrays and  
   user-written functions */  
#include <stdio.h>  
void printarray (int [ ] [7], int);  
int main( ) {  
    int calendar[5][7]={ {1,2,3,4,5,6,7},  
        {8, 9, 10,11,12,13,14},  
        {15,16,17,18,19,20,21},  
        {22,23,24,25,26,27,28},  
        {29,30,31,32,33,34,35}} ;  
    printarray (calendar , 5);  
}
```

```
void printarray (int cal[][7], int j)
{
    int k, n ;
    for (k = 0 ; k < j ; k++)
    {
        for (n = 0 ; n < 7 ; n++)
            printf ("%3d  ", cal[k][n]);
        printf ("\n");
    }
}
```



```
/* Double subscript arrays, user-  
   written  
   functions and pointer arithmetic  
   */  
#include <stdio.h>  
void printarray (int * , int , int);  
int main ( ) {  
    int calendar[5][7]= {{1,2,3,4,5,6,7},  
                          {8, 9, 10,11,12,13,14},  
                          {15,16,17,18,19,20,21},  
                          {22,23,24,25,26,27,28},  
                          {29,30,31,32,33,34,35}};  
    printarray (calendar[0] , 5 , 7);  
}
```

```
void printarray (int *cal, int j, int m)
{
    for (k = 0 ; k < j*m ; k += m)
    {
        for (n = 0 ; n < m ; n++)
            printf ("%3d  ", *(cal+k+n));
        printf ("\n");
    }
}
```

Array of Pointers

- `int *a[5];`
- Declares and allocates an array of pointers to `int`. Each element must be dereferenced individually.
- Each element stored in array are pointed by pointers.

0	1	2	3	4
2000	2010	2030	2080	2100
6020	6022	6024	6026	6028

Pointer to an array

- `int (*a)[10];`
Declares (without allocating) a pointer to an array of `int(s)`. The pointer to the array must be dereferenced to access the value of each element.
- `int b[10];`
- `a = &b;`
- `(*a)[0]` , `(*a)[1]`,

Multidimensional Arrays and Functions

- The most straightforward way of passing a multidimensional array to a function is to declare it in exactly the same way in the function as it was declared in the caller. If we were to call
- `func(a2);` then we might declare
`func(int a[5][7])`
`{ ... }`

Multidimensional Arrays and Functions

- However, it certainly does need to know what a is an array *of*. It is not enough to know that a is an array of "other arrays"; the function must know that a is an array of *arrays of 5 ints*.
- The upshot is that although it does not need to know how many "rows" the array has, it *does* need to know the number of columns.
- That is, if we want to leave out any dimensions, we can only leave out the first one:
- `func(int a[][7]) { ... }`
- The second dimension is still required. (For a three- or more dimensional array, all but the first dimension are required; again, only the first dimension may be omitted.)

Passing 2D array to functions

- since the first element of a multidimensional array is another array, what gets passed to the function is a *pointer to an array*. If you want to declare the function func in a way that explicitly shows the type which it receives, the declaration would be
- `func(int (*a)[7]) { ... }`
- The declaration `int (*a)[7]` says that a is a pointer to an array of 7 ints.

- Finally, we might explicitly note that if we pass a multidimensional array to a function:
- `int a2[5][7];`
- `func(a2);`
- we can *not* declare that function as accepting a pointer-to-pointer:
- `func(int **a) /* WRONG */`
- `{ ... }`

Multidimensional Arrays

- C allows multidimensional arrays by using more []
 - Example: `int matrix[5][10];`
- Some differences:
 - Because functions can be compiled separately, we must denote all but one dimension of a multiple dimensional array in a function's parameter list
 - `void afunction(int amatrix[][10]);`
 - Because arrays are referenced through pointers, there are multiple ways to declare and access 2+ dimensional arrays
 - This will be more relevant when dealing with an array of strings (which is a 2-D array)

```
int a[10][20];  
int *a[10];  
int **a;
```

`*a[4]` –first element of 5th array element
`*a[9]` –first element of 10th array element
`**a` –first element of `a[0]`

```
int *a[3];           // array of 3 pointers  
int x[2] = {1, 2};  
int y[3] = {3, 4, 5};  
int z[4] = {6, 7, 8, 9};  
*a = &x[0];          // a[0] points to x[0]  
*(a+1) = &y[0];       // a[1] points to y[0]  
*(a+2) = &z[0];       // a[2] points to z[0]  
// array a is a jagged array, it is not
```

Address calculations in

- The array elements are stored in memory row after row, so the array equation for element "mat[m][n]" of type DT is:
- $\text{address}(\text{mat}[i][j]) = \text{address}(\text{mat}[0][0]) + (i * n + j) * \text{size}(\text{DT})$
- $\text{address}(\text{mat}[i][j]) = \text{address}(\text{mat}[0][0]) + i * n * \text{size}(\text{DT}) + j * \text{size}(\text{DT})$
- $\text{address}(\text{mat}[i][j]) = \text{address}(\text{mat}[0][0]) + i * \text{size}(1 \text{ row}) + j * \text{size}(\text{DT})$

Pointers to Pointers

- As indicated in the last slide, we can have an array of arrays which is really an array of pointers or pointers to pointers
 - We may wish to use pointers to pointers outside of arrays as well, although it is more common that pointers to pointers represent array of pointers
 - Consider the following:

```
int a;  
int *p;  
int **q;  
a = 10;  
p = &a;  
q = &p;  
printf("%d", **q);
```

We dereference our pointer p with *p but we dereference our pointer to a pointer q with **q

*q is actually p, so **q is a

// outputs 10

Pointers to Functions

- Pointer to function
 - Contains address of function
 - Similar to how array name is address of first element
 - Function name is starting address of code that defines function
- Function pointers can be
 - Passed to functions
 - Stored in arrays
 - Assigned to other function pointers

Pointer to Functions

- A useful technique is the ability to have pointers to functions. Their declaration is easy: write the declaration as it would be for the function, say `int func(int a, float b);` and simply put brackets around the name and a `*` in front of it: that declares the pointer.

`/* pointer to function returning int */`

- `int (*func)(int a, float b);` Once you've got the pointer, you can assign the address of the right sort of function just by using its name: like an array, a function name is turned into an address when it's used in an expression. You can call the function using one of two forms:
 - `(*func)(1,2);`
 - `/* or */`
 - `func(1,2);`

Function Pointers

- Code for a function is in memory..
- ...so we can have a pointer to it
- Function's name is a pointer to it
(just like an array)
- Can also have variables:

```
int (*myFun) (int x) = factorial;
```

```
....
```

```
myFun(3); /*Some people prefer (*myFun)(3)*/
```

Example

```
void func(int arg)
{
    printf("%d\n", arg);
}
```

```
void main()
{
    void (*fp)(int);
    fp = func;
    (*fp)(1);
    fp(2);
}
```


Function returning pointers

- Because of precedence, if you don't parenthesize the name, you declare a function returning a pointer:
- `/* function returning pointer to int */`
- `int *func(int a, float b);`

Example: Bubble Sort

```
#define SIZE 10
int ascending( int, int );
int descending( int, int );

int main() {
int A[SIZE] = { 2, 6, 4, 8, 10,
               12, 89, 68, 45, 37 };
bubble( A, SIZE, ascending );
bubble( A, SIZE, descending );
}

void bubble( int A[],int n,int (*compare)(int,int))
{
    int pass, count;
    void swap( int *, int * );

    for (pass=1; pass<n; pass++)
    {
        for (i=0; i<n-pass; i++)
        {
            if ((*compare)(A[i],A[i+1]))
                swap(&A[i],&A[i+1]);
        }
    }
}
```

```
void swap( int *x,int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```

```
int ascending( int a, int b ) {
return b < a;
/*swap if b is less than a*/
}
```

```
int descending( int a, int b ) {
return b > a;
/*swap if b is greater than a*/
}
```

Function Pointer Usage 1

- Parameterize a function over another function:

```
void doToList(ListNode * head, void (*f)(int x)) {  
    if(head == NULL) return;  
    f(head->data);  
    doToList(head->next, f);  
}
```

Function Pointer Usage 2

- Can be nicer than switch/case when all actions use/result in the same type

```
int addition(int a,int b)
{
    return(a+b);
}
int subtraction(int a,int b)
{
    return(a-b);
}
int multiplication(int a,int b)
{
    return(a*b);
}
int division(int a,int b)
{
    return(a/b);
}
```

```
void main()
{
    int ch,r;
    int (*fp[3])(int,int);
    fp[0] = addition; fp[1] = subtraction;
    fp[2] = multiplication; fp[3] = division;
    do
    { printf("\nEnter ur choice (0/1/2/3: ");
      scanf("%d",&ch);
      r = (*fp[ch])(3,4);
      printf("Result = %d",r);
      printf("\ndo u want to continue : ");
      ch = getche();
    }while(ch=='y');
}
```

Function pointers

- Function Pointers are pointers, i.e. variables, which point to the address of a function. You must keep in mind, that a running program gets a certain space in the main-memory. Both, the executable compiled program code and the used variables, are put inside this memory. Thus a function in the program code is, like e.g. a character field, nothing else than an address. It is only important how you, or better your compiler/processor, interpret the memory a pointer points to.
- Function Pointers provide some extremely interesting, efficient and elegant programming techniques. You can use them to replace *switch/if*-statements, to realize your own *late-binding* or to implement *callbacks*.

Generic Pointers

- When a variable is declared as being a pointer to type void it is known as a generic pointer.
- Since you cannot have a variable of type void, the pointer will not point to any data and therefore cannot be dereferenced.
- It is still a pointer though, to use it you just have to cast it to another kind of pointer first. Hence the term Generic pointer.
- A void * is a pointer to a memory location without actually specifying what data that location stores.

Uses of Generic Pointer

- Useful when you want a pointer to point to data of different types at different times.
- Example

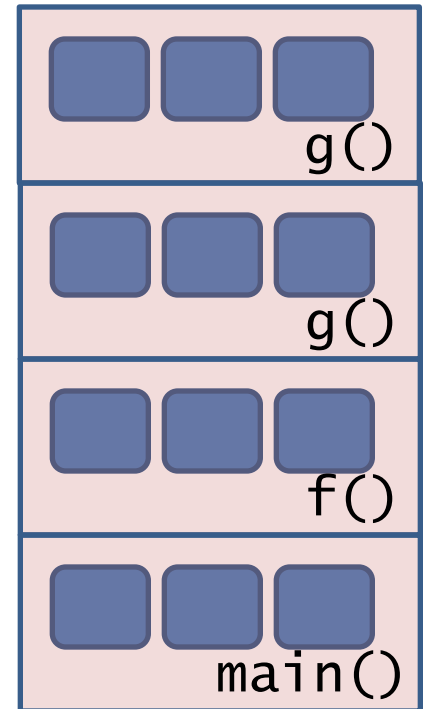
```
void main()
{
    int i = 6;    char c = 'a' ;    void *gp;
    gp = &i;
    printf("gp points to the integer value %d\n", *(int*)gp);
    gp = &c;
    printf("gp now points to the character %c\n", *(char*) gp);
}
```

Dynamic Memory allocation

- The process of allocating memory at run time is known as dynamic memory allocation.
 - Many languages permit a programmer to specify an array size at run time. Such languages have the ability to calculate and assign during executions, the memory space required by the variables in the program.
 - But c inherently does not have this facility but **supports with memory management functions**, which can be used to allocate and free memory during the program execution. The following functions are used in c for purpose of memory management.
1. **malloc** : Allocates memory requests size of bytes and returns a pointer to the 1st byte of allocated space
 2. **calloc** : Allocates space for an array of elements initializes them to zero and returns a pointer to the memory
 3. **free** : Frees previously allocated space
 4. **realloc** : Modifies the size of previously allocated space.

Overview of memory management

- Stack-allocated memory
 - When a function is called, memory is allocated for all of its parameters and local variables.
 - Each active function call has memory on the stack (with the current function call on top)
 - When a function call terminates, the memory is deallocated (“freed up”)
- Ex: `main()` calls `f()`,
 `f()` calls `g()`
 `g()` recursively calls `g()`



Overview of memory management

- Heap-allocated memory
 - This is used for *persistent* data, that must survive beyond the lifetime of a function call
 - global variables
 - dynamically allocated memory – C statements can create new heap data (similar to `new` in Java/C++)
 - Heap memory is allocated in a more complex way than stack memory
 - Like stack-allocated memory, the underlying system determines where to get more memory – the programmer doesn't have to search for free memory space!

Note: `void *` denotes a generic pointer type

Allocating new heap memory

`void *malloc(size_t size);`

- Allocate a block of `size` bytes,
return a pointer to the block
(`NULL` if unable to allocate block)

`void *calloc(size_t num_elements, size_t element_size);`

- Allocate a block of `num_elements * element_size` bytes,
initialize every byte to zero,
return pointer to the block
(`NULL` if unable to allocate block)

Allocating new heap memory

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

- ▶ Given a previously allocated block starting at `ptr`,
 - ▶ change the block size to `new_size`,
 - ▶ return pointer to resized block
 - ▶ If block size is increased, contents of old block may be copied to a completely different region
 - ▶ In this case, the pointer returned will be different from the `ptr` argument, and `ptr` will no longer point to a valid memory region
- ▶ If `ptr` is `NULL`, `realloc` is identical to `malloc`
- ▶ Note: may need to cast return value of `malloc/calloc/realloc`:

```
char *p = (char *) malloc(BUFFER_SIZE);
```

Deallocating heap memory

```
void free(void *pointer);
```

- Given a pointer to previously allocated memory,
 - put the region back in the heap of unallocated memory

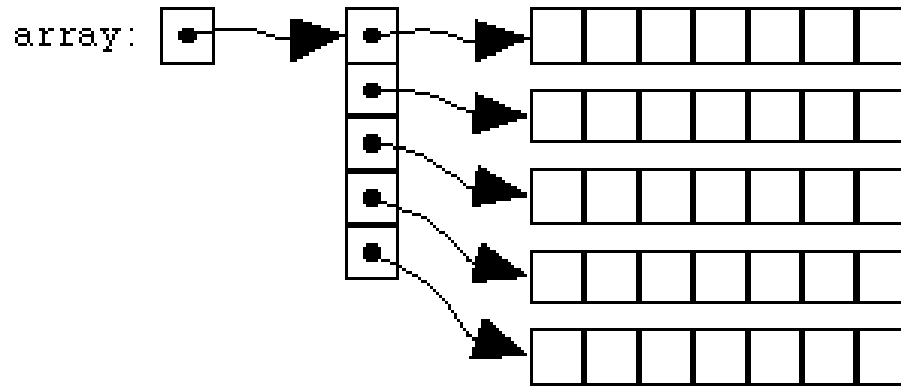
Memory errors

- Using memory that you have not initialized
- Using memory that you do not own
- Using more memory than you have allocated
- Using faulty heap memory management

Dynamic memory allocation egs.

- `int *ip;`
- `ip = (int *)malloc(sizeof(int));`
- `ip = (int *)malloc(n * sizeof(int));`
- `free(ip);`

Dynamically allocating memory for 2 D array



```
int **A;
```

```
Accept r, c
```

```
A = (int **)malloc(r * sizeof(int *));
```

```
for(i = 0; i < r; i++)
```

```
{
```

```
    A[i] = (int *)malloc(c * sizeof(int));
```

```
}
```


Freeing the allocated memory

- `for(i = 0; i < nrows; i++)`
- `free(array[i]);`
- `free(array);`

Structs and Enumeration

Data Structures (struct)

- Arrays require that all elements be of the same data type. Many times it is necessary to group information of different data types. An example is a materials list for a product. The list typically includes a name for each item, a part number, dimensions, weight, and cost.
- C and C++ support data structures that can store combinations of character, integer floating point and enumerated type data. They are called a structs.

Structures (struct)

- A *struct* is a derived data type composed of members that are each fundamental or derived data types.
- A single *struct* would store the data for one object. An array of *structs* would store the data for several objects.
- A collection of one or more variables, typically of different types, grouped together under a single name for convenient handling
- A *struct* can be defined in several ways as illustrated in the following examples:

Declaring Structures (struct)

Does Not Reserve Space

```
struct my_example
{
    int label;
    char letter;
    char name[20];
};
```

/* The name "my_example" is
called a structure tag */

Reserves Space

```
struct my_example
{
    int label;
    char letter;
    char name[20];
} mystruct ;
```

User Defined Data Types (typedef)

- The C language provides a facility called *typedef* for creating synonyms for previously defined data type names. For example, the declaration:

```
typedef int Length;
```

makes the name *Length* a synonym (or alias) for the data type *int*.

- The data “type” name *Length* can now be used in declarations in exactly the same way that the data type *int* can be used:

```
Length a, b, len ;
```

```
Length numbers[10] ;
```

Typedef & Struct

- Often, *typedef* is used in combination with *struct* to declare a synonym (or an alias) for a structure:

```
typedef struct                                /* Define a structure */
{
    int label ;
    char letter;
    char name[20] ;
} Some_name ;                                /* The "alias" is Some_name */

Some_name mystruct ; /* Create a struct variable */
```

Accessing Struct Members

- Individual members of a *struct* variable may be accessed using the structure member operator (the dot, "."):

```
mystruct.letter ;
```

- **Pointer to Structures**
- Or , if a pointer to the *struct* has been declared and initialized

```
Some_name *myptr = &mystruct ;
```

by using the structure pointer operator (the "->"):

```
myptr -> letter ;
```

which could also be written as:

```
(*myptr).letter ;
```


Structs with Union

- `/* The program on the next 3 slides creates a union and makes it a member of struct personal which is, in turn, a member of struct identity. The union uses the same memory location for either rank or a character string (deg) depending on the answer to the prompt for student status in main() */`

Structs with Union (cont.)

```
#include <stdio.h>
```

```
union status
```

```
{
```

```
    int rank ;
```

```
    char deg[4] ;
```

```
};
```

```
struct personal
```

```
{
```

```
    long id ; float gpa ;
```

```
    union status level ;
```

```
};
```

```
struct identity
```

```
{
```

```
    char name[30] ;
```

```
    struct personal student ;
```

```
};
```

Enumeration

- Enumeration is a user-defined data type. It is defined using the keyword **enum** and the syntax is:

```
enum tag_name {name_0, ..., name_n};
```

- The tag_name is not used directly. The names in the braces are symbolic constants that take on integer values from zero through n. As an example, the statement:

```
enum colors { red, yellow, green };
```

- creates three constants. red is assigned the value 0, yellow is assigned 1 and green is assigned 2.

Enumeration

```
/* This program uses enumerated data types to
   access the elements of an array */
#include <stdio.h>
int main( )
{
    int March[5][7]={0,0,1,2,3,4,5},{6,7,8,9,10,11,12},
    {13,14,15,16,17,18,19},{20,21,22,23,24,25,26},
    {27,28,29,30,31,0,0}};
    enum days {Sunday, Monday, Tuesday,
               Wednesday, Thursday, Friday, Saturday};
```

Enumeration

```
enum week {week_one, week_two,  
week_three,  
           week_four, week_five};
```

```
printf ("Monday the third week "  
        "of March is March %d\n",  
        March [week_three] [Monday] );  
}
```

Definition — *Structure*

- A collection of one or more variables, typically of different types, grouped together under a single name for convenient handling
- Known as **struct** in C and C++

struct

- Defines a new *type*
 - I.e., a new kind of data type that compiler regards as a unit
- E.g.,

```
struct motor {  
    float volts;           //voltage of the motor  
    float amps;            //amperage of the motor  
    int phases;            //# of phases of the motor  
    float rpm;             //rotational speed of motor  
};                          //struct motor
```

struct

- Defines a new *type*
- E.g.,

```
struct motor {  
    float volts;  
    float amps;  
    int phases;  
    float rpm;  
};           //struct motor
```

← Name of the type

Note:— name of type is optional if you are just declaring a single **struct** (middle p. 128 of K&R)

struct

- Defines a new *type*
- E.g.,

```
struct motor {  
    float volts;  
    float amps;  
    int phases;  
    float rpm;  
};           //struct motor
```



Members of the
struct

A *member* of a **struct** is analogous
to a *field* of a class in Java

Declaring `struct` variables

```
struct motor p, q, r;
```

- Declares and sets aside storage for three variables – `p`, `q`, and `r` – each of type `struct motor`

```
struct motor M[25];
```

- Declares a 25-element array of `struct motor`; allocates 25 units of storage, each one big enough to hold the data of one `motor`

```
struct motor *m;
```

- Declares a pointer to an object of type `struct motor`

Accessing Members of a `struct`

- Let

```
struct motor p;  
struct motor q[10];
```

- Then

<code>p.volts</code>	— is the voltage
<code>p.amps</code>	— is the amperage
<code>p.phases</code>	— is the number of phases
<code>p.rpm</code>	— is the rotational speed

<code>q[i].volts</code>	— is the voltage of the <code>i</code> th motor
<code>q[i].rpm</code>	— is the speed of the <code>i</code> th motor

Accessing Members of a **struct** (continued)

- Let

`struct motor *p;`

- Then

`(*p).volts` — is the voltage of the **motor** pointed to by **p**

`(*p).phases` — is the number of phases of the **motor** pointed to by **p**

Why the parentheses?

Accessing Members of a **struct** (continued)

- Let

struct motor

- Then

(*p) .voltage — is the voltage of the **motor** pointed to by **p**

(*p) .phases — is the number of phases of the **motor** pointed to by **p**

Because '.' operator has higher precedence than unary '*'

Accessing Members of a **struct** (continued)

- Let

```
struct motor *p;
```

- Then

(*p) .volts — is the voltage of the **motor** pointed to by **p**

(*p) .phases — is the number of phases of the **motor** pointed to by **p**

Accessing Members of a **struct** (continued)

- The **(*p) .member** notation is a nuisance
 - Clumsy to type; need to match ()
 - Too many keystrokes
- This construct is so widely used that a special notation was invented, i.e.,
 - **p->member**, where **p** is a pointer to the structure
- Ubiquitous in *C* and *C++*

Previous Example Becomes ...

- Let

```
struct motor *p;
```

- Then

`p -> volts` — is the voltage of the `motor` pointed to by `p`

`p -> phases` — is the number of phases of the `motor` pointed to by `p`

Operations on `struct`

- Copy/assign

```
struct motor p, q;  
p = q;
```

- Get address

```
struct motor p;  
struct motor *s  
s = &p;
```

- Access members

```
p.volts;  
s -> amps;
```

Operations on `struct` (continued)

- Remember:—
 - Passing an argument by value is an instance of *copying* or *assignment*
 - Passing a return value from a function to the caller is an instance of *copying* or *assignment*

- E.g,:—

```
struct motor f(struct motor g)
{
    struct motor h = g;
    ...;
    return h;
}
```

Initialization of a `struct`

- Let `struct motor` {
 `float volts;`
 `float amps;`
 `int phases;`
 `float rpm;`
}; `//struct motor`
- Then
 `struct motor m = {208, 20, 3, 1800};`
initializes the `struct`

Why structs?

- Open-ended data structures
 - E.g., structures that may grow during processing
 - Avoids the need for `realloc()` and a lot of copying
- Self-referential data structures
 - Lists, trees, etc.

Example

```
struct item {  
    char *s;  
    struct item *next;  
}
```

Yes! This is legal!



- I.e., an `item` can point to another `item`
- ... which can point to another `item`
- ... which can point to yet another `item`
- ... etc.

Thereby forming a *list* of `items`

Typedef

- Definition:— a **typedef** is a way of *renaming* a type

- E.g.,

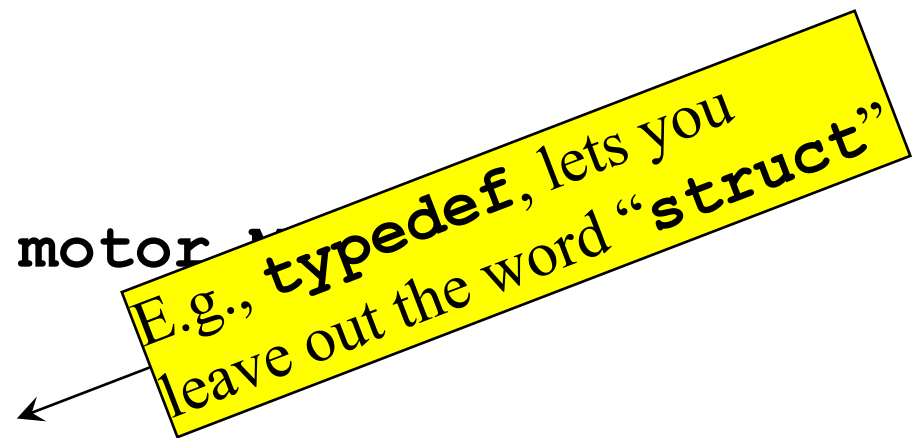
```
typedef struct motor {
```

```
    Motor m, n;
```

```
    Motor *p, r[25];
```

```
    Motor function(const Motor m; ...) ;
```

E.g., **typedef**, lets you
leave out the word “**struct**”

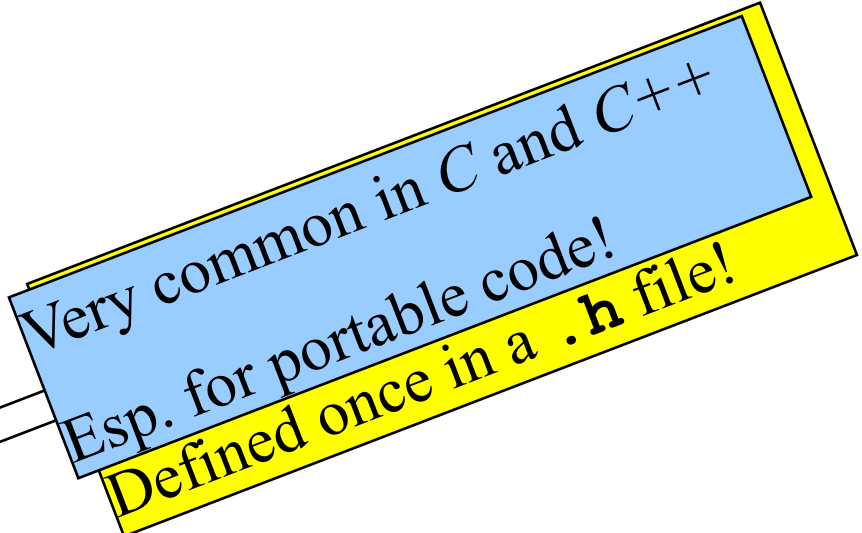


typedef (continued)

- **typedef** may be used to rename *any* type
 - Convenience in naming
 - Clarifies purpose of the type
 - Cleaner, more readable code
 - Portability across platforms
- E.g.,
 - `typedef char *String;`
- E.g.,
 - `typedef int size_t;`
 - `typedef long int32;`
 - `typedef long long int64;`

typedef (continued)

- **typedef** may be used to rename *any* type
 - Convenience in naming
 - Clarifies purpose of the type
 - Cleaner, more readable code
 - Portability across platforms
- E.g.,
 - `typedef char *String;`
- E.g.,
 - `typedef int size_t;`
 - `typedef long int32;`
 - `typedef long long int64;`



Very common in C and C++
Esp. for portable code!
Defined once in a **.h** file!

Unions

- A **union** is like a **struct**, but only one of its members is stored, not all
 - I.e., a single variable may hold different types at different times
 - Storage is enough to hold largest member
 - Members are overlaid on top of each other
- E.g.,

```
union {  
    int ival;  
    float fval;  
    char *sval;  
} u;
```

Unions (continued)

- It is *programmer's responsibility* to keep track of which type is stored in a **union** at any given time!

```
struct taggedItem {  
    enum {iType, fType, cType} tag;  
    union {  
        int ival;  
        float fval;  
        char *sval;  
    } u;  
};
```

Unions (continued)

- It is *programmer's responsibility* to keep track of which type is stored in a **union** at any given time!
- E.g., (p. 148)

```
struct taggedItem {  
    enum {iType, fType, cType} tag;  
    union {  
        int ival;  
        float fval;  
        char *sval;  
    } u;  
};
```

Members of **struct** are:–

```
enum tag;  
union u;
```

Value of **tag** says which
member of **u** to use

Unions (continued)

- **unions** are used much less frequently than **structs** — mostly
 - in the inner details of operating system
 - in device drivers
 - in embedded systems where you have to access registers defined by the hardware