Arrays and Pointers (part 2)

EECS 2031

Be extra careful with pointers!

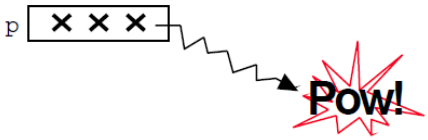
Common errors:

- Overruns and underruns
  - Occurs when you reference a memory beyond what you allocated.
- Uninitialized pointers
- Null pointers de-referencing
- Memory leaks
- Inappropriate use of freed memory
- Inappropriately freed memory

Uninitialized Pointers

- Example:

```
int *p;
*p = 20;
```



3

Null Pointer Dereferencing

```
main( ) {
 int *x;
 x = ( int * )malloc( sizeof( int ) );
 *x = 20;  /* What could go wrong? */
}
```

Better code:

```
x = ( int * ) malloc( sizeof( int ) );
if ( !x ) {
  printf( "Insufficient memory!\n" );
  exit( 1 );
}
*x = 20;
```

4

Memory Leaks

```
int *x;
x = (int *)
     malloc( 20 );
x = (int *)
     malloc( 30 );
```

- The first memory block is lost for ever.
- May cause problems (exhaust memory).

```
int *x;
x = (int *)
     malloc( 20 );
… …
free( x );
x = (int *)
     malloc( 30 );
```

5

Inappropriate Use of Freed Memory

```
char *x;
x = (char *) malloc( 50 );
free( x );
x[0] = 'A';      /* Does work on some systems though */
```

6

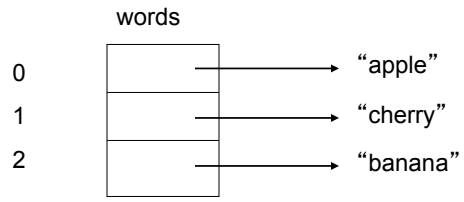
Arrays of Pointers (5.6)

```
char *words[] = { "apple", "cherry", "banana" };
```

- **words** is an array of pointers to **char**.

  (char *)(words[])

- Each element of **words** (**words[0], words[1], words[2]**) is a pointer to **char**.



7

Arrays vs. Pointers

What is the difference between the previous example and the following?

```
char words[][10] = { "apple",
                     "cherry",
                     "banana"};
```

8

Arrays of Pointers: Example

```
char *words[] = { “apple”,
                  “cherry”,
                  “prune” };
                      +1
char **p;
p = words;
printf("%c\n", **p);
printf("%c\n",*(*(p+1)+2));
printf("%c\n",*(*(p+2)+2)+1);
```

9

Arrays vs. Pointers: Example

```
main()
{
char mots[][10] = { "banana", "peach", "honeydew" };
printf( "%s %s %s\n", mots[0], mots[1], mots[2] );
mots[1][0] = 'P';
printf( "%s %s %s\n", mots[0], mots[1], mots[2] );

char *words[] = { "apple", "cherry", "prune" };
printf( "%s %s %s\n", words[0], words[1], words[2] );
/* Test 1:
words[1][0] = 'C';  */
/* Test 2:
char **p;
p = words;
**p = 'A';  */
printf( "%s %s %s\n", words[0], words[1], words[2] );
}
```

10

Pointers to Whole Arrays

```
char a[10] = "banana";
char *p1;
char (*p2)[10];

p1 = a;  /* p1 points to first element*/
p2 = &a; /* p2 points to the whole array */
printf("%c %c", *p1, **p2);

// What's the difference between p1+1 and p2+1 ?
// What's char *p3[100] ?
```


Pointers to Pointers (5.6)

- Pointers can point to integers, floats, chars, and other pointers.

```
int **j;
int *i;
int k = 10;
i = &k;
j = &i;
printf("%d   %d   %d\n", j, i, k);
printf("%d   %d   %d\n", j, *j, **j);
printf("%x   %x   %x\n", j, *j, **j);
```

Output on some system:

-1073744352 -1073744356 10
-1073744352 -1073744356 10
bffff620 bffff61c a

Multi-dimensional Arrays (5.7)

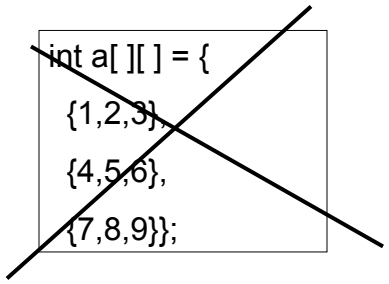
```
int a[3][3];
```

```
int a[3][3] = {
  {1,2,3},
  {4,5,6},
  {7,8,9}};
```

```
int a[ ][3] = {
  {1,2,3},
  {4,5,6},
  {7,8,9}};
```

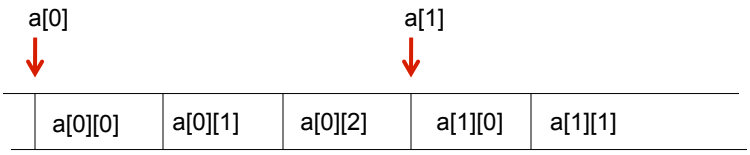
To access the elements:

```
if ( a[2][0] == 7 )
   printf ( ... );
for ( i=0, j=0; ... ; i++, j++ )
   a[i][j] = i+j;
```



Multi-dimensional Arrays (cont.)

- Multi-dimensional arrays are arrays of arrays.
- For the previous example, a[0] is a pointer to the first row.
- Lay out in memory



Multi-dimensional Arrays: Example

```
#include <stdio.h>

int main() {
 float *pf;
 float m[][3]={{0.1, 0.2, 0.3},
               {0.4, 0.5, 0.6},
               {0.7, 0.8, 0.9}};
 printf("%d  \n", sizeof(m));
 pf = m[1];
 printf("%f   %f   %f  \n",*pf, *(pf+1), *(pf+2));
}
```

```
 36
 0.4000   0.5000  0.6000
```

15

Multi-D Arrays in Function Declarations

```
int a[2][13];  // to be passed to function f

f( int daytab[2][13] ) { ... }
or
f( int daytab[ ][13] ) { ... }
or
f( int (*daytab)[13] ) { ... }
```

Note: Only to the first dimension (subscript) of an array is free; all the others have to be specified.

16

Initialization of Pointer Arrays (5.8)

```
/* month_name: return name of n-th month */
char *month_name( int n )
{
   static char *name[] = {
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
   };
   return (n < 1 || n > 12) ? name[0] : name[n];
}
```

17

Pointers vs. Multi-D Arrays (5.9)

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

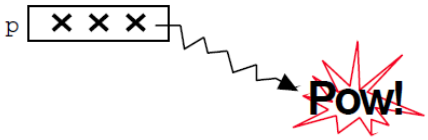
- a: 200 int-size locations have been set aside.
- b: only 10 pointers are allocated and not initialized; initialization must be done explicitly.
  - Assuming each element of b points to an array of 20 elements, total size = 200 integers + 10 pointers.
- Advantage of b: the rows of the array may be of different lengths (saving space).

18

Uninitialized Pointers (cont.)

- Example 1

```
int *p;
*p = 20;
```



- Example 2

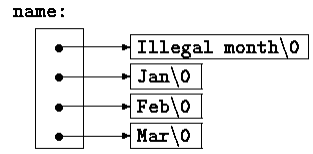
```
main() {
 char *x[10];
 strcpy( x[1],"Hello" );
}
```

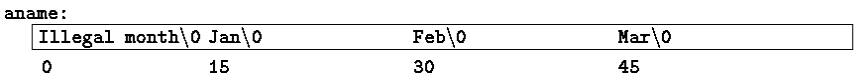
19

Advantage of Pointer Arrays

```
char *name[ ] = { "Illegal month", "Jan", "Feb", "Mar" };

char aname[ ][15] = {"Illegal month", "Jan", "Feb", "Mar" };
```





20

Command-Line Arguments (5.10)

- Up to now, we define **`main`** as **`main().`**
- Usually it is defined as

  `main( int argc, char *argv[] )`
- **`argc`** is the number of arguments.
- **`argv`** is a pointer to the array containing the arguments.
- **`argv[0]`** is a pointer to a string with the program name. So **`argc`** is at least 1.
- **`argv[argc]`** is a NULL pointer.

21

Command-Line Arguments (cont.)

```
main( int argc, char *argv[] ) {
int i;
printf( "Number of arg = %d\n", argc );
for( i = 0; i < argc; i++ )
        printf( "%s\n", argv[i] );
}
```

a.out
Number of arg = 1
a.out

a.out hi by 3
Number of arg = 4
a.out
hi
by
3

22

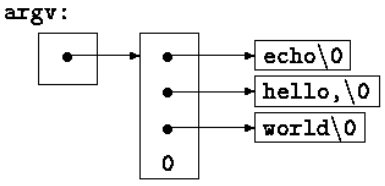
Example

- Write a program name echo (echo.c) which echoes its command-line arguments on a single line, separated by blanks.
- Command: `echo hello, world`
- Output: `hello, world`

23

Example: Diagram

- Write a program name echo (echo.c) which echoes its command-line arguments on a single line, separated by blanks.
- Command: `echo hello, world`
- Output: `hello, world`



24

echo, 1st Version

```
main( int argc, char *argv[] )
{
  int i;
  for (i = 1; i < argc; i++)
   printf("%s%s", argv[i], (i < argc-1) ? " " : "");
  printf("\n");
  return 0;
}
```

25

echo, 2nd Version

```
main( int argc, char *argv[] )
{
  while (--argc > 0)
    printf("%s%s", *++argv, (argc > 1) ? " " : "");
  printf("\n");
  return 0;
}
```

26

Complicated Declarations (5.12)

```
char **argv             argv: pointer to pointer to char
int (*daytab)[13]       daytab: pointer to array[13] of int
int *daytab[13]         daytab: array[13] of pointer to int
int *comp()             comp: function returning pointer to
                           int
```

27

Next lectures ...

- File I/O (section7.5)
- Functions (chapter 4)
- Introduction to UNIX (posted notes)

28