Arrays and Pointers (part 2)

CSE 2031

Fall 2012

Be extra careful with pointers!

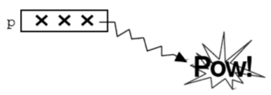
Common errors:

- 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 1

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



- Example 2

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

Null Pointer Dereferencing

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

Better code:

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

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).

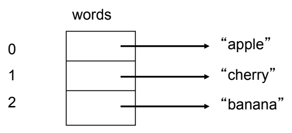
Inappropriate Use of Freed Memory

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

Arrays of Pointers (5.6)

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

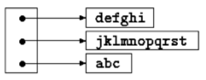
- `words` is an array of pointers to `char`.
- Each element of `words` (`words[0], words[1], words[2]`) is a pointer to `char`.



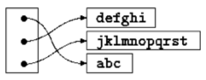
Arrays vs. Pointers

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

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



Previous example



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}};

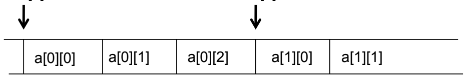
To access the elements:

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

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

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
```

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.

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];
}
```

Pointers vs. Multi-D Arrays (5.9)

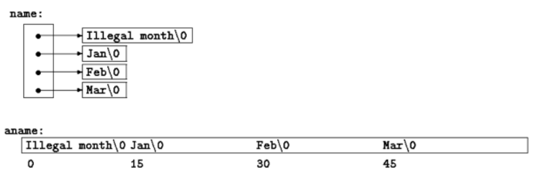
```
int a[10][20];
int *b[10];
```

- 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).

Advantage of Pointer Arrays

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

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



Command-Line Arguments (5.10)

- Up to now, we defines 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.

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
```

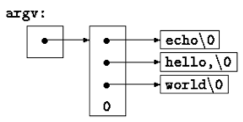
```
a.out hi by 3
Number of arg = 4
a.out
hi
by
3
```

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`

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`



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;
}
```

echo, 2nd Version

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

Next time ...

- Structures (Chapter 6)