C Programming Language:
C ADTs , 2d Dynamic Allocation

Math 230

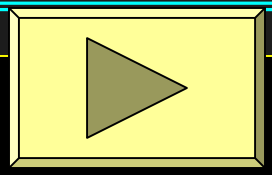
Assembly Language Programming
(Computer Organization)

Thursday Jan 31, 2008

Overview

- Row major format
- 1 and 2-d dynamic allocation
- struct and union

Strings: Dynamic Allocation



```
/* malloc example: string generator*/
#include <stdio.h>
#include <stdlib.h>

int main ()
{
  int i,n;
  char * buffer;

  printf ("How long do you want the string? ");
  scanf ("%d", &i);

  buffer = malloc (i+1);
  if (buffer==NULL) exit (1);

  for (n=0; n<i; n++)
    buffer[n]=rand()%26+'a';
  buffer[i]='\0';

  printf ("Random string: %s\n",buffer);
  free (buffer);

  return 0;
}
```

Dynamic Array Allocation, Addressing

- Recall that &a[0] == a => a pointer constant

```
int a[20];
```

```
int* a;
a = malloc(20 * sizeof(int));
```

- C will do pointer math. Note:

`(a+2)` ➔ `&a[0] + 2*sizeof(a[0])`

2-dimensional arrays

	j=0	j=1	j=2	j=3
i=0	x_0	x_1	x_2	x_3
i=1	x_4	x_5	x_6	x_7
i=2	x_8	x_9	x_{10}	x_{11}

- For static array declaration: 3 rows, 4 cols
- What's the equivalent *declaration* in pointer notation of `int x[3][4];` ?
- How do you address x[1][3] via pointers?

Row-Major Format I

- Interpret [3][4] as 3 groups of 4-tuples

	j=0	j=1	j=2	j=3
i=0	x_{00}	x_{01}	x_{02}	x_{03}
i=1	x_{10}	x_{11}	x_{12}	x_{13}
i=2	x_{20}	x_{21}	x_{22}	x_{23}

	j=0	j=1	j=2	j=3
i=0	x_0	x_1	x_2	x_3
i=1	x_4	x_5	x_6	x_7
i=2	x_8	x_9	x_{10}	x_{11}

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}
0				4				8			

Row-Major Format II

	j=0	j=0	j=0	j=0
i=0	x_0	x_1	x_2	x_3
i=1	x_4	x_5	x_6	x_7
i=2	x_8	x_9	x_{10}	x_{11}

x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9	x_{10}	x_{11}
0				4				8			

x_{00}	x_{01}	x_{02}	x_{03}	x_{10}	x_{11}	x_{12}	x_{13}	x_{20}	x_{21}	x_{22}	x_{23}
0				4				8			

- The first index selects the 4-tuple (the set)
  - x[1][2] ➔ start @ absolute position 4
  - move two places
    - moves us to sixth position

Row-Major Format III

- If the compiler knows it's working with groups of 4, it can determine the address of any element (based on indices, eg x[2][1], x[1][3], etc.)

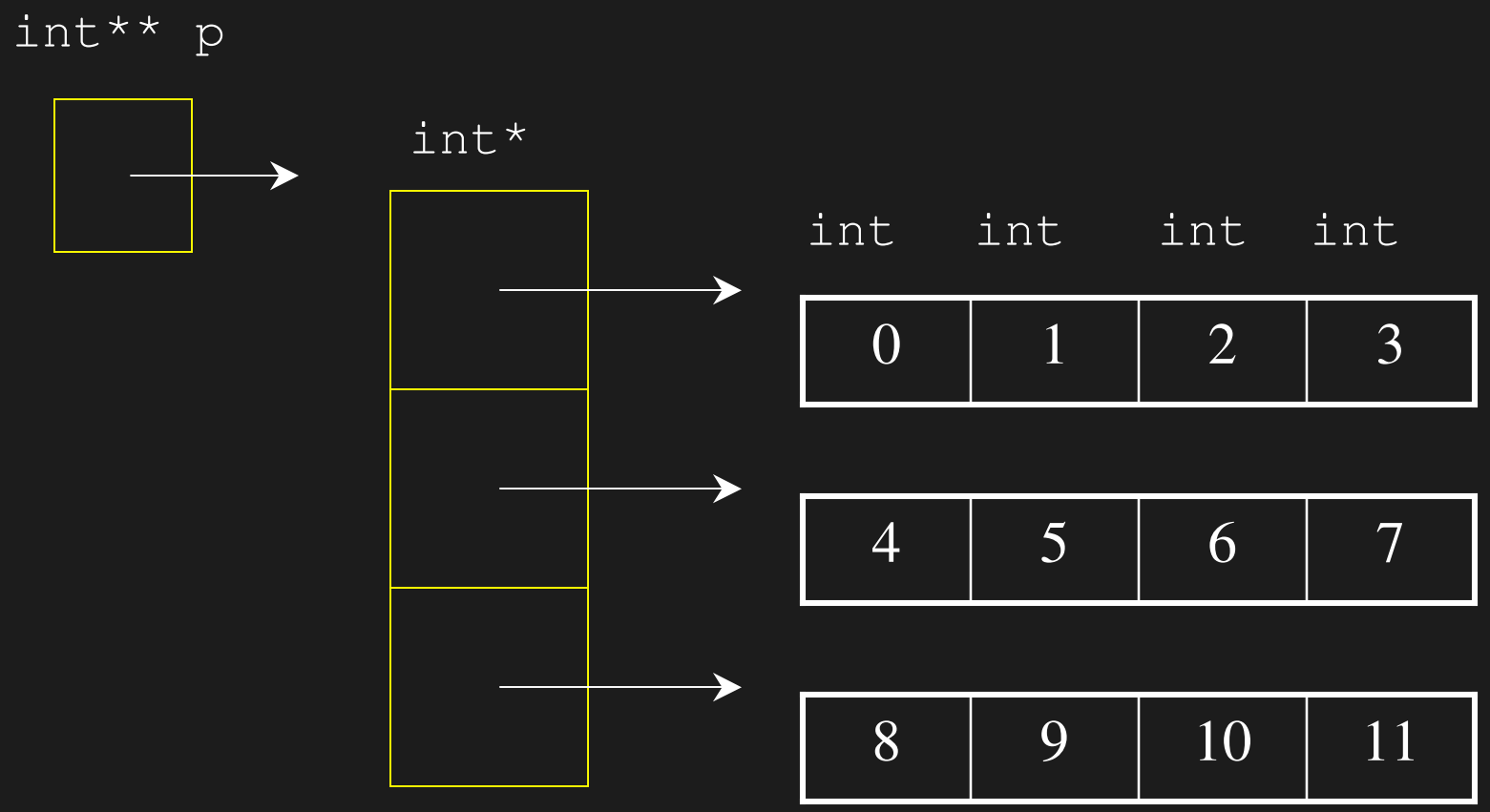
x[1][2] is located @ 1×4+2 bytes

from address of x[0][0]

Hence: x[i][j] = *(x + i×n + j)

where n is the row size (i.e., number of columns)

2d - Dynamic Allocation I




```
int** p;  // vs int p[3][4];
```

2d - Dynamic Allocation II

```
int** p;  // vs int p[3][4];
p = malloc( 3*sizeof(int*));
if( p = NULL) error();
for(i=0; i<3; i++)
  p[i] = malloc(4*sizeof(int));
```

To dereference:

```
p[i][j]  = *(p[i] + j)
         = *( *(p+i) + j)
```

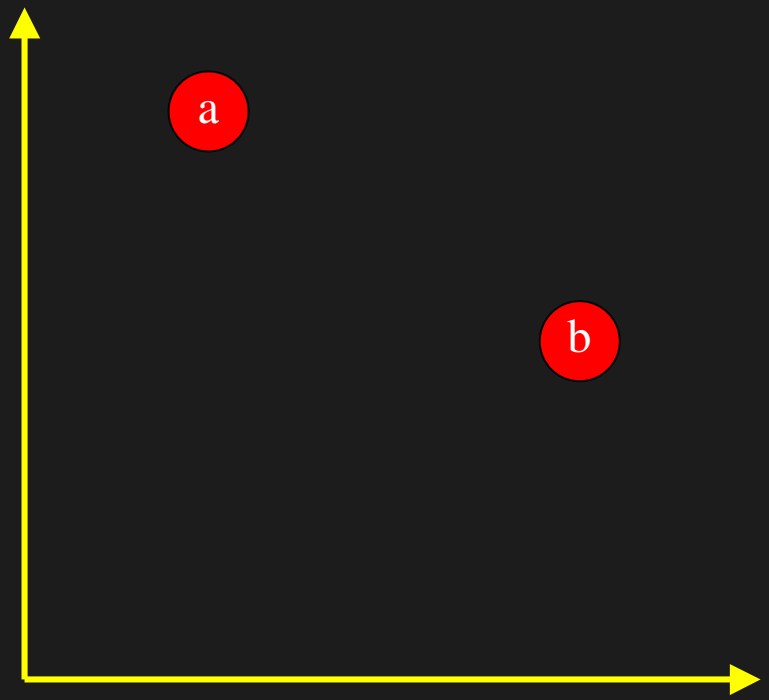
more info: http://www.csc.liv.ac.uk/~frans/COMP101/AdditionalStuff/multiDarrays.html

Administrivia

- Lab 2 posted and due next Thursday 2/7, b/f 11am for sign-off
- Lab 3 available and due Thursday 2/14, 11am
- Project 1 posted and due Feb 24, Sun, b/f midnight for submission

Structures: Building ADTs

- Points
  - We can use a new Abstract Data Type (ADT) to define a point in a plane, and the operations we perform on that point.




```
struct ptData { /* 1 */
    double x;
    double y;
};
struct ptData  a, b, c[10];
```

Structures

- The following are all equivalent statements:

```
struct ptData { /* 1 */
    double x;
    double y;
};
struct ptData  a, b, c[10];
```

```
struct ptData {/* 2 */
    double x;
    double y;
};
typedef struct ptData Point;
Point   a, b, c[10];
```

```
typedef struct {/* 3 */
    double x;
    double y;
} Point;
Point   a, b, c[10];
```

Structures

- We can use the statement
  `Point a, b;`
  to declare 2 variables of type `Point`
- We can refer to individual members of a structure by name, eg

```
a.x = 1.0 ;
a.y = 1.0 ;

b.x = 4.0 ;
b.y = 5.0 ;
```

- This allows set `a` to represent the Point (1, 1) and `b` to represent (4, 5).

Structures

- Initialization

```
typedef struct {/* 3a */
    double x;
    double y;
} Point;
Point   a = {1.0, 2.0};
Point   b = {0.0, 1.1};
Point   c, d[10];
```

Unions:

- are memory that contain a variety of objects over time
- can be used to hold data of type character, integer, double precision, or other C data types
- can only contain one of these types at any given time
- its members share space, thus conserving memory storage
- its members are accessed in the same manner as structures

Union declarations are the same as struct declarations

```
union u_tag{
        char cval;
        int ival;
        double dval;
} u;
int x = 3;
char y = 'f';
double pi = 3.1459;

u.ival = x;
u.cval = 'f';
u.dval = pi;
```

Valid Union Operations:

- Assignment to union of same type: =
- Taking the address: &
- Accessing union members: .
- Accessing members using pointers: ->