Introduction to C, C++, and Unix/Linux

CS 60

Lecture 9: Arrays and Structs

Today

→ Arrays and Structs

- Reading for Monday: K&R ch. 1-6 & 7.1-7.4

Arrays

For a local variable array, this can be an expression e.g., **`int x[big ? 100 : 20]`**

```
int x[100]
```

creates an array of 100 integers, indexed from 0 to 99

```
x[0], x[1], ..., x[99]
```

Array values must be initialized:

```
for (i=0; i<100; i++)
    x[i] = 0;
```

Not restricted to a constant subscript value (**`i`** is an expression, not a constant)

Arrays (cont.)

- An array can be initialized when declared:

```
int x[] = {0, 0, 0, 0, 0};
```

or

```
int x[5] = {0, 0, 0, 0, 0};
```

Compiler warning if the list is longer than N (5) elements

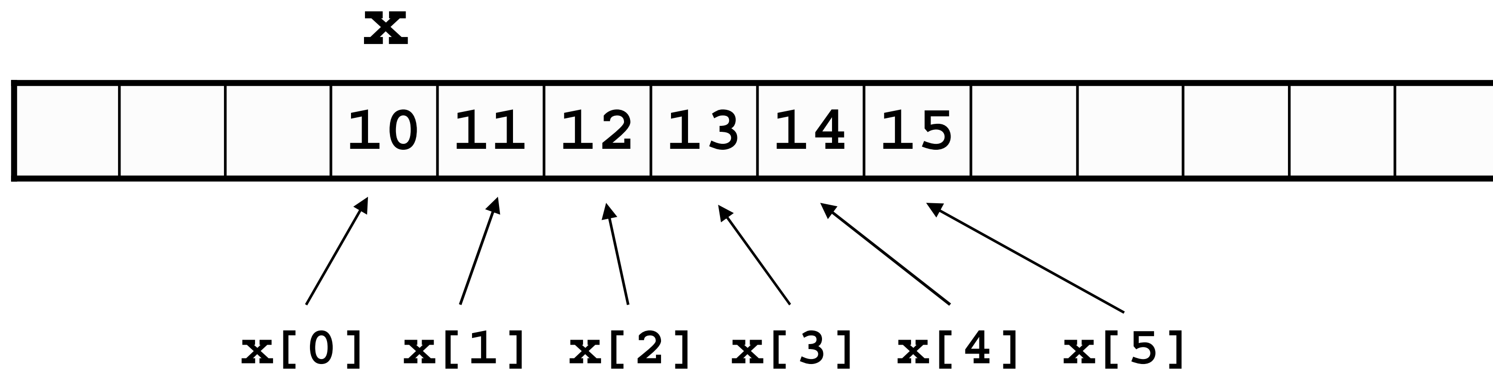
No problem if the list is shorter than N (5) elements

– Only the elements in the list will be initialized

Arrays (cont.)

- Array values are stored sequentially in memory

```
int x[] = {10, 11, 12, 13, 14, 15};
```

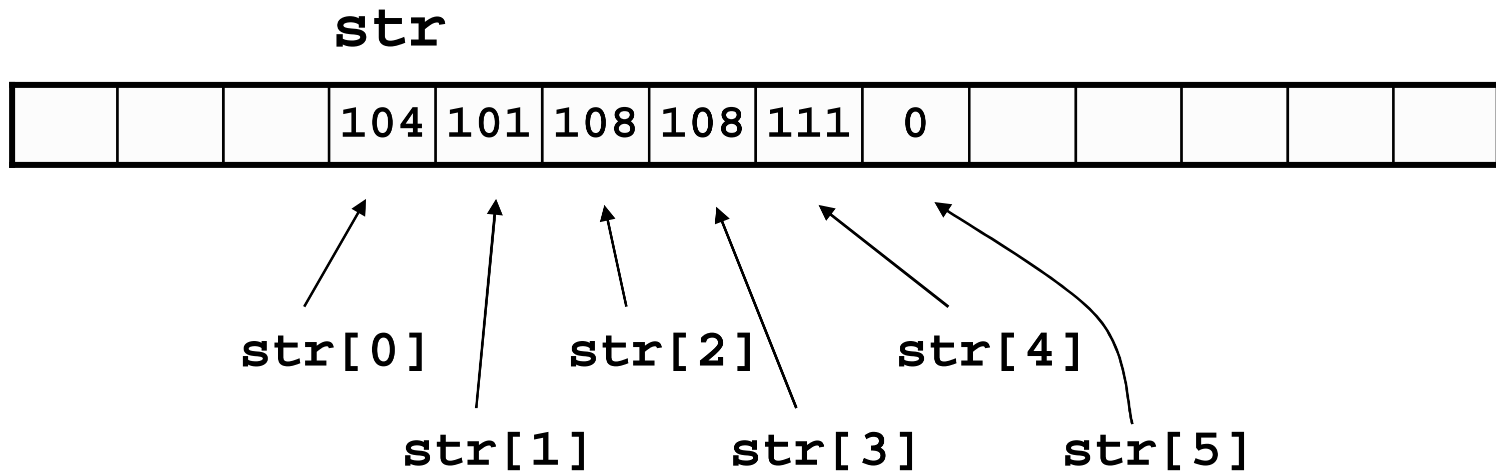


Initializing string arrays

```
char str[] =
   {'h', 'e', 'l', 'l', 'o', '\0'};
```

is equivalent to:

```
char str[] = "hello";
```



```
char str[3] = “YES”;
```

← What’s wrong with this?

Should be: **`char str[4] = “YES”;`**

Or better: **`char str[] = “YES”;`**

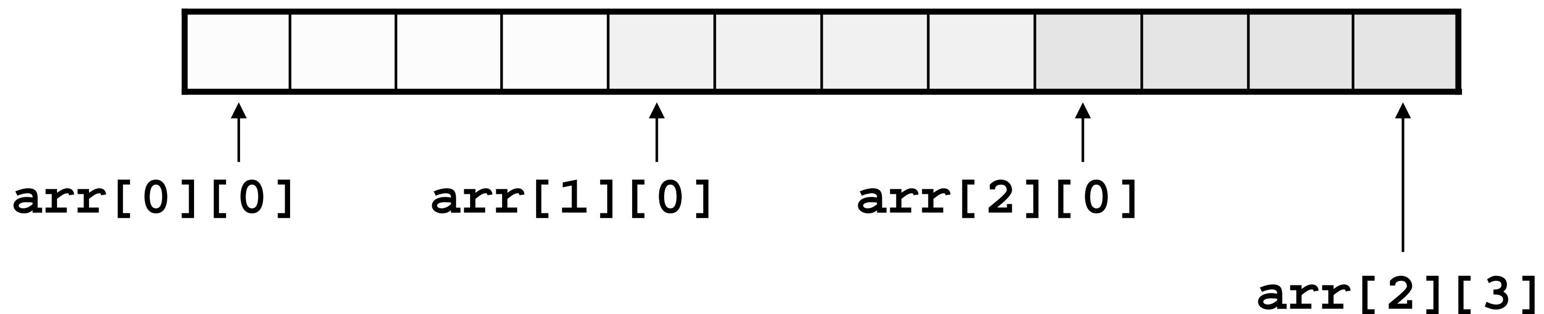
Multidimensional arrays

- Arrays can be two (or more) dimensional
  - Though usually, pointers will be more efficient

```
int arr[3][4]
```

Three groups of four integers

```
for (i=0; i<3; i++)
  for (j=0; j<4; j++)
    arr[i][j] = 0;
```



Initializing multidimensional arrays

```
char str[][6] = {"hello", "hi    ",
                 "howdy", "grtng" };
```

Cannot do this:

```
char str[][] = {"hello", "hi    ",
        ↑        "howdy", "grtng" };
```

3

"Three groups of two groups of two groups of three integers"

```
int arr[ ][2][2][3] = {
    {{{1, 2, 3}, {4, 5, 6}},
     {{7, 8, 9}, {10, 11, 12}}},
    {{{1, 2, 3}, {4, 5, 6}},
     {{7, 8, 9}, {10, 11, 12}}},
    {{{1, 2, 3}, {4, 5, 6}},
     {{7, 8, 9}, {10, 11, 12}}}
};
```

Also the order in which the array is laid out in memory

Question

- If you wanted to go through memory sequentially, which would you do:

```
char im[n][n]

for (i=0 i<n i++)
  for (j=0 j<n j++)
    arr[i][j] = x++;
```

```
char im[n][n]

for (i=0 i<n i++)
  for (j=0 j<n j++)
    arr[j][i] = x++;
```

arr[i][j] and **arr[i][j+1]** are adjacent in memory
arr[i][j] and **arr[i+1][j]** are not

Common mistake: **arr[3,4]** or **arr(3,4)** !!

Must be **arr[3][4]**

This may compile just fine – why?

```
int arr[3][4]

for (i=0 i<3 i++)
  for (j=0 j<4 j++)
    printf("%d\n", arr[i,j]);
```

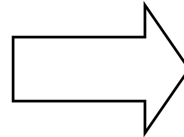
Evaluates to **arr[j]**

Which is a 1D array
But prints garbage.

Structs

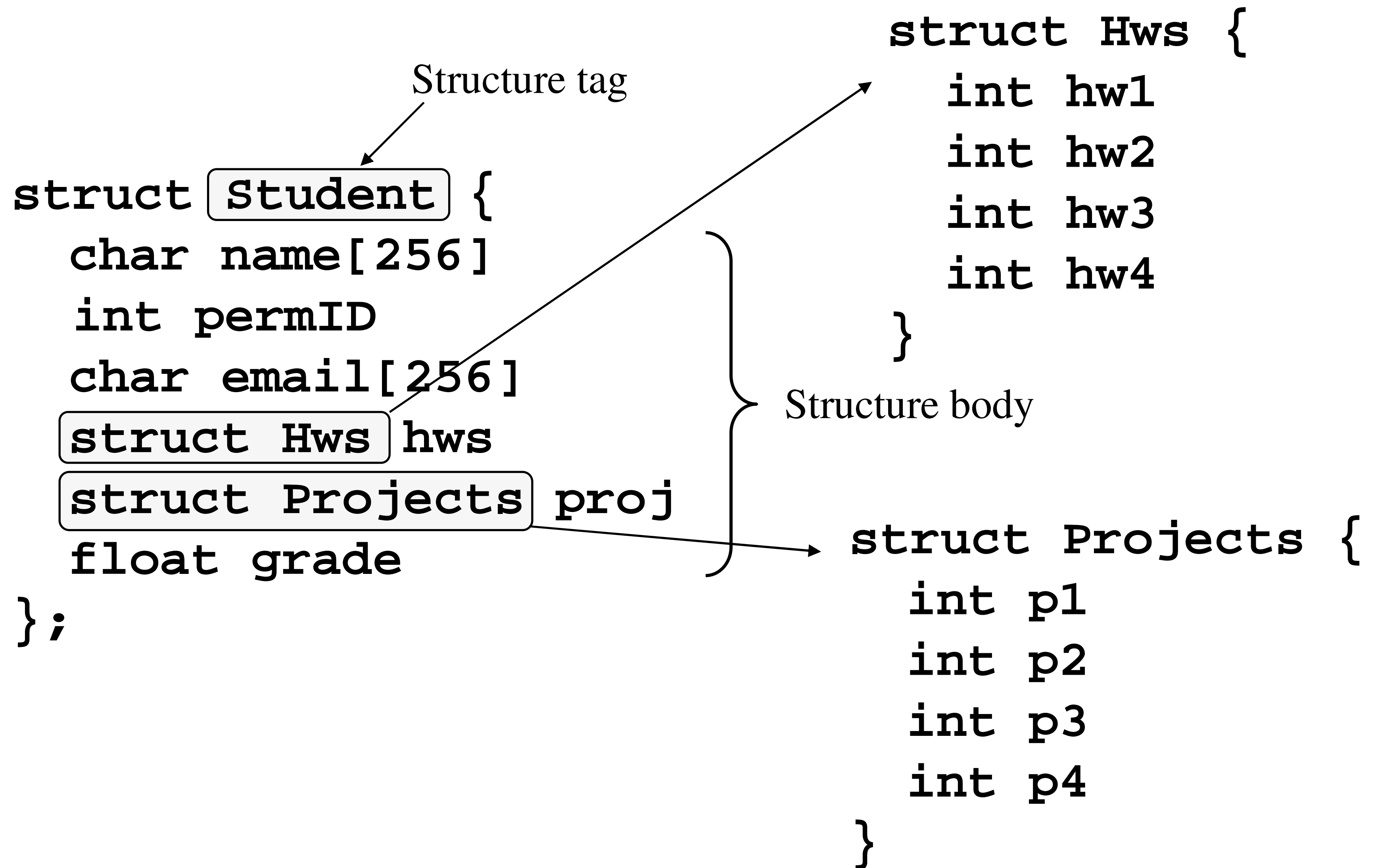
- A structure is a collection of variables, possibly of different types, grouped together under a single name for convenience
  - Memory is allocated as a block, with each item in sequence

Student info for CS60:
- Name
- Perm#
- Email address
- HW grades
- Project grades
- Final grade



```
student.name
student.permID
student.email
student.hws.hw1
student.hws.hw2
student.proj.p1
student.grade
```

struct declaration



`struct` declaration

`sizeof(struct Student)` returns 552

```
struct Student {
  char name[256]              256 bytes
  int permID                  4 bytes
  char email[256]             256 bytes
  struct Hws hws              16 bytes
  struct Projects proj        16 bytes
  float grade                 4 bytes
};                            ---------
                              552 bytes
```

`struct` declaration

```
struct Student {
  int permID
  char email[256]
  struct Hws hws
  struct Projects proj
  float grade
};
```

```
struct Student s1;
struct Student s2;
struct Student s[90];

s1.permID = 12345;
strcpy(s1.email,
  "joe@cs.ucsb.edu");
s2 = s1;

s[12].hws.hw1 = 24;
s[12].hws.hw2 = 25;

for (i=0 i<90 i++)
  s[i].grade = 100;
```

```
struct Point {
  int x
  int y
};
```

```
struct Point p1;
struct Point p2;

p1.x = 0;
p1.y = 0;
p2.x = 320;
p2.y = 240;
DrawRect(p1, p2);
```

…or…

```
struct Point {
  int x
  int y
} p1, p2;

p1.x = p1.y = 0;
```

…or…

```
struct {
  int x
  int y
} p1, p2;

p1.x = p1.y = 0;
```

Initializing and passing structures

```
struct Point p1 = {0, 0};
struct Point p2 = {320, 240};

DrawRect(p1, p2);
```

```
struct Point Doubles(struct Point in)
{
   struct point temp;
   temp.x = 2 * in.x;
   temp.y = 2 * in.y;
   return(temp);
}
```

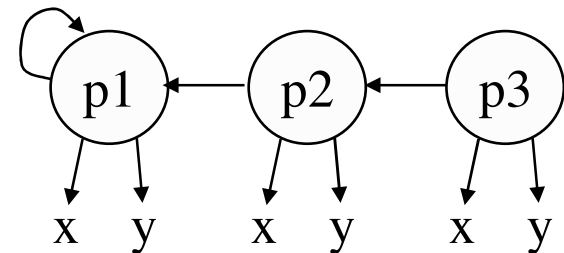
Self-referring structures

- Structures may have elements that are pointers to the same structure

```
struct lPoint {
  int x
  int y
  struct lPoint *prev
} p1, p2, p3;

p3.prev = &p2;
p2.prev = &p1;
p1.prev = &p1;
```

For example, to build a linked list



```
typedef type synonym
```

Typedef

- It’s inconvenient (and error prone) to always use “**`struct Point`**” as the type specifier
- The **`typedef`** keyword provides a mechanism to creates a synonym (an alias) for an existing type

```
typedef int Size;                 Size length, width;
typedef unsigned int UINT;        UINT count;
typedef unsigned char Byte;       Byte value;
typedef char *String;             String m = “Hi”;

typedef char* String;
```

typedef and **struct**

- Use **typedef** along with **struct** :

```
typedef struct point {
   int x
   int y
} Point;
```

```
typedef struct {
   int x
   int y
} Point;
```

```
Point a, b;
```

```
typedef struct Point {
   int x
   int y
} Point;
```

```
struct point a, b;
```

```
struct Point a, b;
```

Review:

```
typedef type synonym
```

```
typedef struct {...} synonym
```

```
typedef unsigned char Byte;
```

```
typedef char *String;
```

```
typedef struct Point {
   int x
   int y
} Point;
```

`typedef` can help make code portable:

`portable.h`

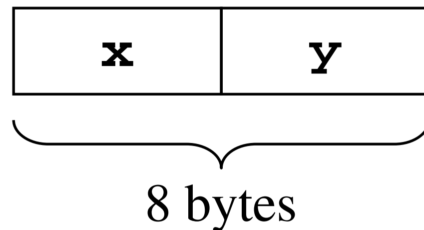
```
typedef short int SINT;
typedef unsigned short int USINT;
typedef int INT;
typedef unsigned int UINT;
typedef long int LINT;
typedef unsigned long int ULINT;
```

1. Always include **`portable.h`** and use **`SINT`**, **`USINT`**, **`INT`**, etc. in your code.
2. When changing machines, edit **`portable.h`** to make sure that **`SINT`** and **`USINT`** always define 16-bit values, **`INT`** and **`UINT`** always define 32-bit values, and **`LINT`** and **`ULINT`** always define 64-bit values.

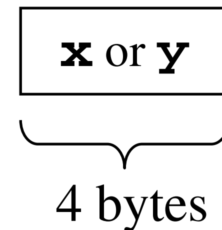
Unions

- Like typedefs, unions are data structures that can hold multiple elements
  - But rather than reserving memory for these elements sequentially, they all describe the same, overlapping, segment of memory

```
struct Point {
  int x;
  int y;
};
```




```
union Point {
  int x;
  int y;
};
```



```
union Point {
  int x;
  int y;
} pt;

pt.x = 100;
printf("y=%d", pt.y);
→ 100
```

```
union Point {
  unsigned int x;
  short int y[2];
  unsigned char s[4];
} var;

var.x = 0xffffffff;
var.y[0] = 0;
for (i=0 i<4 i++)
 printf("%d,", var.s[i]);

→ 0,0,255,255
```

```c
union Pixel32 {
  unsigned int val;
  struct {
     unsigned char r;
     unsigned char g;
     unsigned char b;
     unsigned char a;
  } rgba;
  float fval;
}

typedef union Pixel32 Pixel32;
```

```c
Pixel32 p;

p.val = 192;

p.rgba.r = 150;
p.rgba.g = 94;
p.rgba.b = 220;

p.fval = 2.3F;
```