EECS 2031

Software Tools

Module 7 – Arrays, Structs, Pointers

Arrays

- An ordered list of data of the same type
- Each item in an array is called an **element**
- Loops commonly used for manipulation
- Programmers set array sizes explicitly

Declaring Arrays

- Syntax

```
type name[size];
```

- Examples

```
int bigArray[10];
double a[3];
char grade[10], oneGrade;
```

Accessing elements

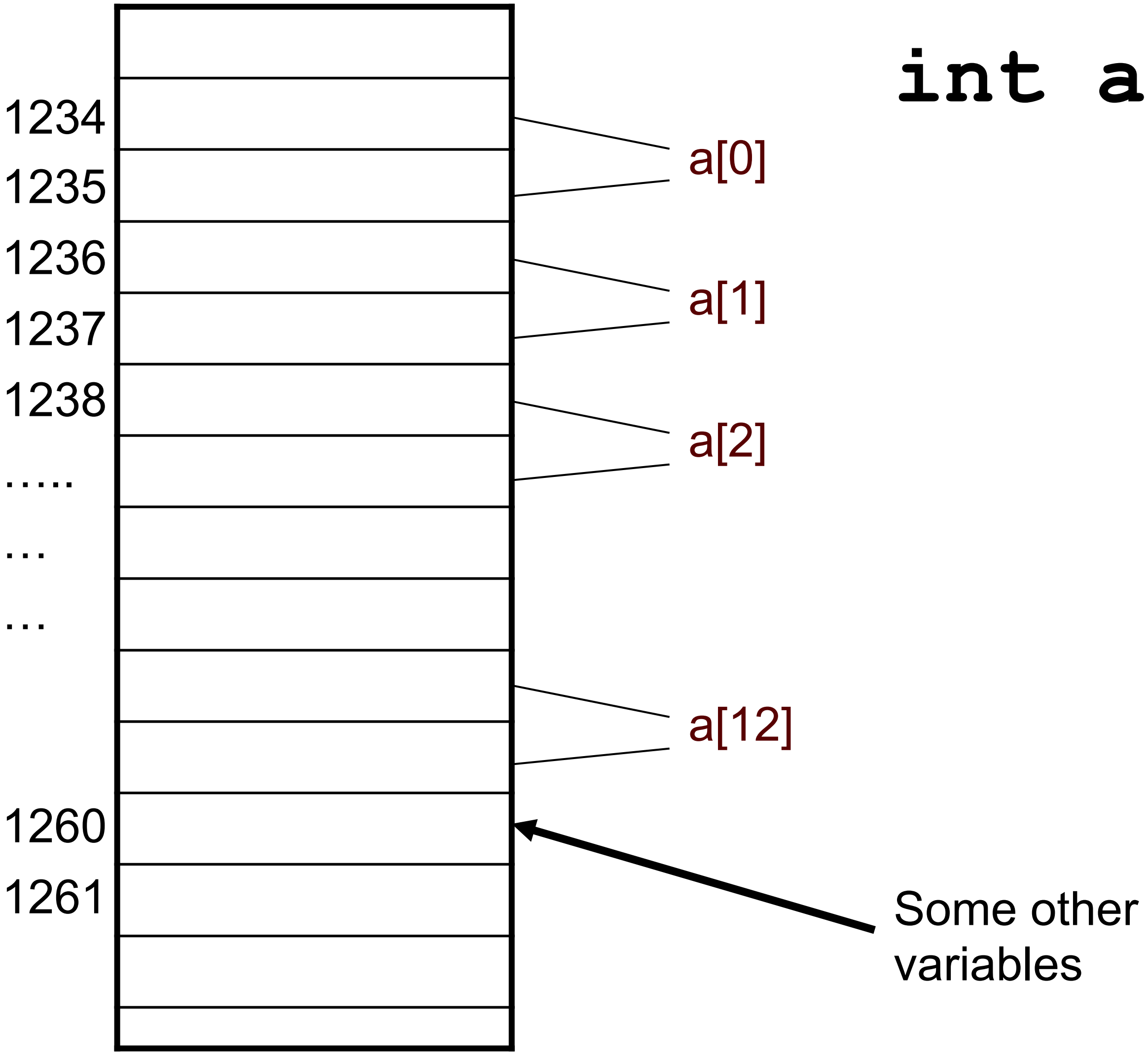
- The following array declaration allocates memory for 5 integers

```
int score[5];
```

- Elements are accessed using the bracket notation
- `score[0]` is the first element
- `score[4]` is the last element
- The number in the brackets is called the **index** of the element

Arrays Stored in Memory

```
int a[13];
```



Array Initialization

- Array elements are not initialized automatically
- Initialization can be done at declaration time

```
int a[5] = {1, 2, 3, 4, 5};
```

- Declares array `a` and initializes first element to `1`, second to `2` etc.

Array Initialization

```
int b[5] = {11,22};
```

- Declares array **b**, initializes first two elements, and all remaining elements are set to zero

```
int c[ ] = {1,2,8,9,5};
```

- Declares array **c**, sets its length to 5, and initializes all elements

Common error: Index out of range

- C does not check array boundaries
- It is the responsibility of the programmer not to access array elements that do not exist
- Behaviour is **undefined** when an out-of-range index is used

Common error: Index out of range

- Possible results of index out of range
  - Runtime error
  - Strange values for other variables
  - Nothing goes wrong at all
- Behaviour may be different when run in a different environment
- Very hard to debug
- See **`array.c`**

Multi-dimensional Arrays

- **`int a[4][5];`**
- Defines an array of 4 rows and 5 columns
- In memory, all elements are laid out in row-major order, i.e. first will be the elements of the first row, and then the elements of the second row etc.
- **`a[1][3]`** is the element in the second row and fourth column

Multi-Array Initialization

```
int a[2][3] = {
    {22, 44, 66}, // Row 0
    {97, 98, 99}  // Row 1
};
```

- As can be seen above, multi-dimensional arrays are really arrays of arrays
- `a[0]` is an array of 3 elements

Arrays of strings

```
char names[5][30];
```

- Declares an array of 5 strings
- Each string can be as long as 29 characters (remember that the array must include space for the `\0` character)
- `names[1]` refers to the second string

Structures

- Sometimes data is related
  - Time expressed in hours and minutes
  - Coordinates of a point
- It would be better to group such related data, similarly to what classes do in Java
- In C, we can define **structures** that encapsulate related data

Structures

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

- The above declares a new type called `struct point`
- We can declare variables of this type: `struct point origin;`

Structure Members

- `x` and `y` are called the data members of the structure
- They can be accessed using dot notation

```
origin.x = 0;
origin.y = 0;
```

`typedef`

- `struct point` is an awkward name for a type
- The `typedef` keyword allows the definition of new types

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

`Point` is now a valid type

Nested Structures

```
typedef struct rect {
  Point pt1;
  Point pt2;
} Rectangle;
Rectangle screen;
screen.pt1.x = 0;
screen.pt2.y = 400;
```

Structures and Functions

- Structures are very helpful when it comes to functions
- They allow a function to return multiple pieces of data
- A single parameter can also contain multiple pieces of data

Structures and Functions

```
Point makepoint(int x, int y) {
  Point temp;
  temp.x = x;
  temp.y = y;
  return temp;
}
```

Using Structures

- Structures cannot be assigned

```
Point pt1, pt2;
pt1.x = 0;
pt1.y = 0;
pt2 = pt1;   /* Error! */
```

- Must write a function to copy a structure

Initializing Structures

```
typedef struct {
   float width;
   float height;
} Dimensions;
Dimensions sofa = {2.0, 3.0};
```

Structures and Arrays

- Declaring arrays whose elements are structures is helpful in many situations

```
Point points[100];

points[3].x = 34;
```

- We’ll return to arrays and structures once we discuss pointers

Pointers

- A pointer is a variable whose value is a memory address
- The memory address ***pointed to*** by the pointer typically contains actual data, such as integers or structures
- The following declares an integer, and a pointer to an integer:

```
int i;

int *p;
```

Pointers and addresses

- To connect a pointer to the data, use the reference operator **&**

```
int i = 23;
int *p;
p = &i;
```

- The general form is:

```
pointer = &data;
```

- The type of **data** and the type **pointer** points to must match

Dereferencing a pointer

- To access the data a pointer points to, use the dereference operator `*`

```
int i = 23;
int *p = &i;
int j = *p;
```

- Unfortunately, the type declaration of a pointer, and its dereferencing look the same (`*p`) but they are quite different

Dereferencing a pointer

- If **`p`** is a pointer to an integer, then **`*p`** can be used anywhere an **`int`** variable can be used
- See **`alias.c`**
- Pointers make debugging much harder!

Pointer assignment

- Consider the following program snippet

```
int i = 8, j = 9;
int *p1, *p2;
p1 = &i;
p2 = &j;
```

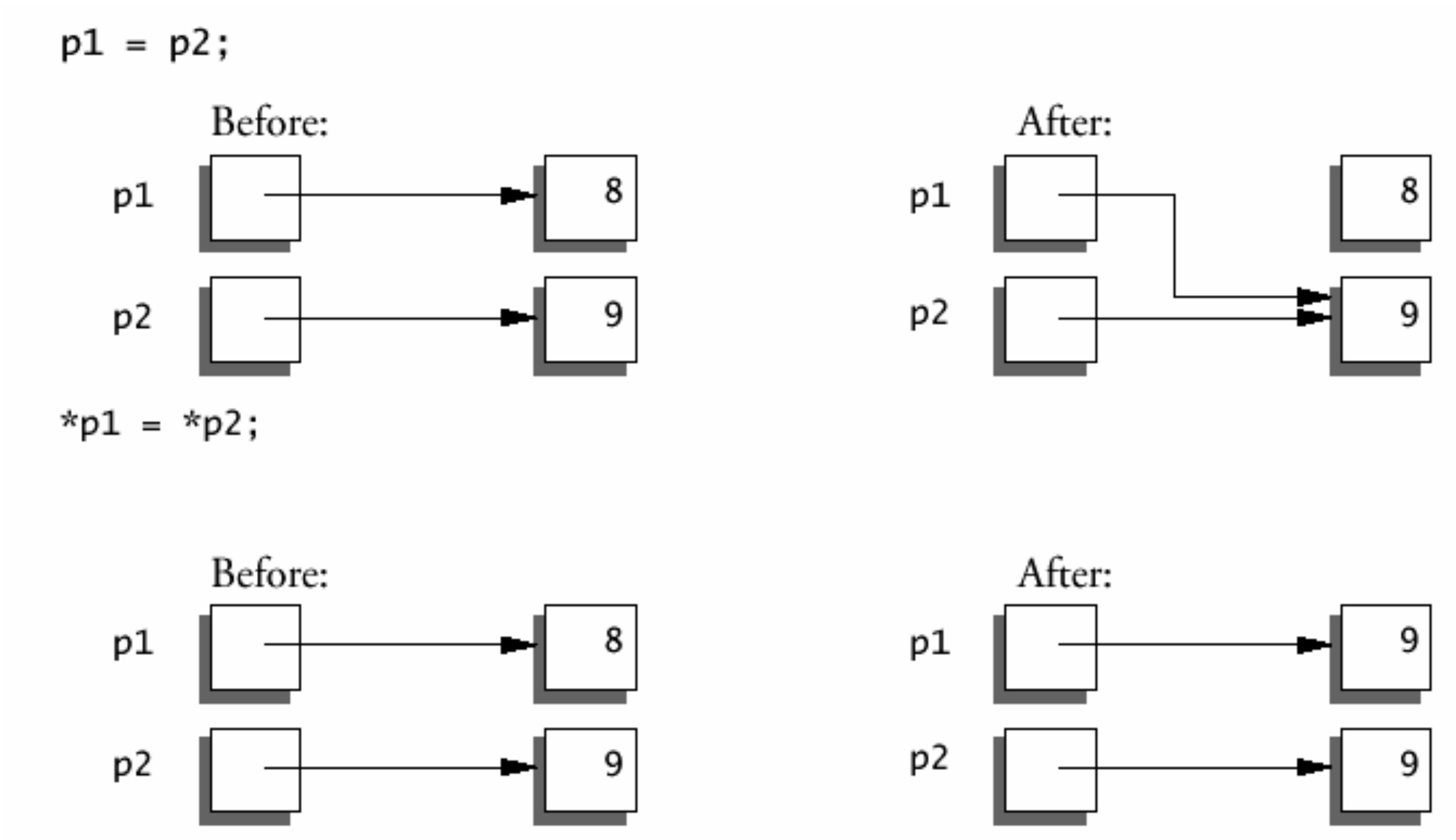
- What is the effect of

```
p1 = p2;
```

- What about

```
*p1 = *p2;
```

Pointer Assignment



```
p1 = p2;
```

```
*p1 = *p2;
```

Pointers and Function Arguments

- Suppose we want to write a function that swaps the values of two integers a and b
- Because of the way arguments are passed to functions in C, it is a bit tricky to do
- See **`swapWrong.c`** and **`swap.c`**

Pointers and Function Arguments

- C passes arguments to functions by value
- This means that a copy of the variable is given to the function
- The function can only change the local copy of the variable
- What if we want to change a variable in the calling function?
  - Pass a pointer to the variable

Pointers and Function Arguments

- This is why `scanf` expects a pointer to the data we want to read
    - It can then access the data, and update it
- See Section 6.8 in the textbook
- If we want a function to modify a structure, we also need to pass a pointer to the structure

Pointers and Arrays

- The identifier of an array is equivalent to the address of its first element

```
int numbers[20];
int *p;
p = numbers;
```

- **`p`** now points to the first element of the array
- In other words, **`numbers`** is the same as **`&numbers[0]`**

Pointers and Arrays

- The identifier of an array behaves like a pointer but cannot be assigned to

```
int numbers[20];
int *p;
numbers = p; // Invalid
```

- **`p`** can be assigned to point to any int, but **`numbers`** will always point to the same address
- Think of **`numbers`** as a constant

Pointer Arithmetic

```
int numbers[20], *p;
p = numbers;
int x = *p;
```

- **`x`** is equal to the first element of the array

```
int y = *(p+1);
```

- **`y`** is equal to the second element of the array

```
p++;
```

- **`p`** points to the second element of the array

Pointer Arithmetic

```
int i = 9;
int *p = &i;
```

- The value of **`p`** is the memory address of **`i`**, e.g. 1234
- Adding one to **`p`** will increase its value by **`sizeof(int)`**
- After **`p++;`** the value of p will be 1238 (assuming **`sizeof(int)`** is 4)

Pointers and Arrays

```
int a[10];

int *pa;
```

- All expressions below are valid

```
a[i]     ⇔    *(a+i)

&a[i]    ⇔    a+i

pa[i]    ⇔    *(pa+i)
```

Computing String Length

```
int strlen(char *s)
{
    int n;
    for (n = 0; *s != '\0’; s++ )
        n++;
    return n;
}
```

Computing String Length

Following are valid examples of using the `strlen` function in the previous slide

```
char array[20] = “hello world”;

char *ptr = array;

strlen("hello world”);

strlen(array);

strlen(ptr);
```

Pointer Arithmetic

- Given pointers **`p`** and **`q`** of the same type and integer **`n`**, the following pointer operations are legal:
- **`p+n`**, **`p-n`**
  - **`n`** is scaled according to the size of the objects **`p`** points to. If **`p`** points to an integer of 4 bytes, **`p+n`** advances by 4*n bytes
- Continued on next slide...

Pointer Arithmetic

- `p-q` (assuming `p>q`)
  - Returns an `int`: the difference between the two addresses divided by `sizeof(type)`
  - `p+q` is illegal!
- `q = p;  p = q + 100;`
  - `p` and `q` must point to the same types
  - Casting is possible but should be avoided

Pointer Arithmetic

- More things you can do with pointers

```
if ( p == q )
if ( p != q + n )
p = NULL;
if ( p == NULL )
```

Counting String Length v2

```
int strlen(char *s)
{
  char *p = s;
  while (*p != '\0')
    p++;
  return p - s;
}
```

Important point about strings

```
char amessage[] = “hello”;
char *pmessage  = “hello”;
```

- **`amessage`** will always refer to the same memory address
- **`pmessage`** may later be modified to point elsewhere

Pointers to Structures

```
Point origin = {0, 0};
Point *pp;
pp = &origin;
printf("%d\n", (*pp).x);
```

- The parentheses in `(*pp).x` are necessary
- `*pp.x` would imply `pp` is a structure

Pointers to Structures

- `(*pp).x` can be written as `pp->x`

  ```
  printf("%d\n", pp->x);
  ```

- If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure

- See `pointstruct.c`

Why pointers?

- Pointers can be confusing and a source of hard to resolve bugs, but they are also quite powerful
- They allow sharing of data
- They allow dynamic memory management (see next module)
- See **`strcpy.c`** for more examples