EECS 22: Advanced C Programming

Lecture 13

Rainer Dömer

doemer@uci.edu

The Henry Samueli School of Engineering
Electrical Engineering and Computer Science
University of California, Irvine

Lecture 13: Overview

- Pointers
  - Definition, initialization and assignment
  - Pointer dereferencing
  - Pointer arithmetic
    - Increment, decrement
  - Pointer comparison
  - Pointers and Arrays
    - Equivalence!
    - Array layout in linear address space

Pointers

- *Pointers* are variables whose values are *addresses*
  - The "address-of" operator (**&**) returns a pointer!
- Pointer Definition
  - The unary ***** operator indicates a pointer type in a definition

```
int x = 42;   /* regular integer variable */
int *p;       /* pointer to an integer */
```

- Pointer initialization or assignment
  - A pointer may be set to the "address-of" another variable

```
p = &x;       /* p points to x */
```

  - A pointer may be set to **0** (points to no object)

```
p = 0;        /* p points to no object */
```

  - A pointer may be set to **NULL** (points to "NULL" object)

```
#include <stdio.h>   /* defines NULL as 0 */
p = NULL;     /* p points to no object */
```


Pointers

- Pointer Dereferencing
  - The unary ***** operator dereferences a pointer to the value it points to ("content-of" operator)

```
#include <stdio.h>

int  x = 42;   /* regular integer variable */
int *p = NULL; /* pointer to an integer */
```

Pointers

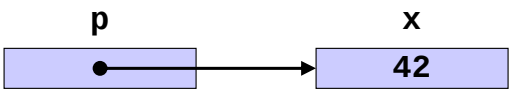
- Pointer Dereferencing
  - The unary * operator dereferences a pointer to the value it points to ("content-of" operator)

```
#include <stdio.h>

int  x = 42;   /* regular integer variable */
int *p = NULL; /* pointer to an integer */

p = &x;        /* make p point to x */
printf("x is %d, content of p is %d\n", x, *p);
```

```
x is 42, content of p is 42
```



Pointers

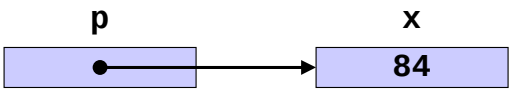
- Pointer Dereferencing
  - The unary * operator dereferences a pointer to the value it points to ("content-of" operator)

```
#include <stdio.h>

int  x = 42;   /* regular integer variable */
int *p = NULL; /* pointer to an integer */

p = &x;        /* make p point to x */
printf("x is %d, content of p is %d\n", x, *p);
*p = 2 * *p;   /* multiply content of p by 2 */
printf("x is %d, content of p is %d\n", x, *p);
```

```
x is 42, content of p is 42
x is 84, content of p is 84
```



Pointers

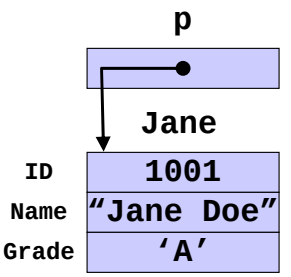
- Pointer Dereferencing
  - The **->** operator dereferences a pointer to a structure to the content of a structure member

```
struct Student
{  int  ID;
   char Name[40];
   char Grade;
};

struct Student Jane =
{1001, "Jane Doe", 'A'};

struct Student *p = &Jane;

void PrintStudent(void)
{
   printf("ID:    %d\n", p->ID);
   printf("Name:  %s\n", p->Name);
   printf("Grade: %c\n", p->Grade);
}
```




```
ID:    1001
Name:  Jane Doe
Grade: A
```


Pointers

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element

```
int x[5] = {10,20,30,40,50}; /* array of 5 integers */
int *p;                      /* pointer to integer */

p = &x[1];                   /* point p to x[1] */
printf("%d, ", *p);          /* print content of p */
```

```
20,
```

Pointers

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element

```
int x[5] = {10,20,30,40,50}; /* array of 5 integers */
int *p;                      /* pointer to integer */

p = &x[1];                   /* point p to x[1] */
printf("%d, ", *p);          /* print content of p */
p++;                         /* increment p by 1 */
printf("%d, ", *p);          /* print content of p */
```

```
20, 30,
```


Pointers

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element

```
int x[5] = {10,20,30,40,50}; /* array of 5 integers */
int *p;                      /* pointer to integer */

p = &x[1];                   /* point p to x[1] */
printf("%d, ", *p);          /* print content of p */
p++;                         /* increment p by 1 */
printf("%d, ", *p);          /* print content of p */
p--;                         /* decrement p by 1 */
printf("%d, ", *p);          /* print content of p */
```

```
20, 30, 20,
```

Pointers

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element

```
int x[5] = {10,20,30,40,50}; /* array of 5 integers */
int *p;                      /* pointer to integer */

p = &x[1];                   /* point p to x[1] */
printf("%d, ", *p);          /* print content of p */
p++;                         /* increment p by 1 */
printf("%d, ", *p);          /* print content of p */
p--;                         /* decrement p by 1 */
printf("%d, ", *p);          /* print content of p */
p += 2;                      /* increment p by 2 */
printf("%d, ", *p);          /* print content of p */
```

```
20, 30, 20, 40,
```


Pointers

- Pointer Comparison
  - Pointers may be compared for equality
    - operators **==** and **!=** are useful to determine *identity*
    - operators **<**, **<=**, **>=**, and **>** are *not* applicable

```
int x[5] = {10,20,10,20,10}; /* array of 5 integers */
int *p1, *p2;                /* pointers to integer */

p1 = &x[1]; p2 = &x[3];      /* point to x[1], x[3] */

if (p1 == p2)
  { printf("p1 and p2 are identical!\n");
   }
if (*p1 == *p2)
  { printf("Contents of p1 and p2 are the same!\n");
   }
```

```
Contents of p1 and p2 are the same!
```

Pointers

- Pointer Comparison
  - Pointers may be compared for equality
    - operators **==** and **!=** are useful to determine *identity*
    - operators **<**, **<=**, **>=**, and **>** are *not* applicable

```
int x[5] = {10,20,10,20,10}; /* array of 5 integers */
int *p1, *p2;                /* pointers to integer */

p1 = &x[1]; p2 = &x[3];      /* point to x[1], x[3] */
p1 += 2;                     /* increment p1 by 2 */
if (p1 == p2)
  { printf("p1 and p2 are identical!\n");
   }
if (*p1 == *p2)
  { printf("Contents of p1 and p2 are the same!\n");
   }
```

```
p1 and p2 are identical!
Contents of p1 and p2 are the same!
```


Pointers and Arrays

- In C, *Pointers and Arrays are equivalent!*
  - A pointer represents an address in memory
  - An array is represented by the address of its first element in memory
- Passing Arrays and Pointers to Functions
  - Arrays are passed *by reference*
  - Pointers are references and passed as such
- Array Access is equivalent to Pointer Dereferencing
  - Example:

```
int A[10];
...
A[0] = 42;
...
A[5] = 17;
```

```
int A[10], *p = &A[0];
...
*p = 42;
...
*(p+5) = 17;
```

Pointers and Arrays

- Dynamic Arrays
  - Example 1: Static 1-dim. array
    - Static definition
    - Passed as static array
    - Static array access
    - ➢ Static size everywhere!

```
int Sum(int A[100])
{
  int i, sum = 0;
  for(i=0; i<100; i++)
  { sum += A[i];
  }
  return sum;
}

int main(void)
{
  int d[100], s;
  ...
  s = Sum(d);
  ...
  return 0;
}
```


Pointers and Arrays

- Dynamic Arrays
  - Example 2: Static 1-dim. array
    - Static definition
    - Passed as static array plus size
    - ➢ Received as pointer!
    - ➢ Accessed via pointer!

```
int Sum(int *p, int m)
{
  int i, sum = 0;
  for(i=0; i<m; i++)
  { sum += *(p + i);
  }
  return sum;
}

int main(void)
{
  int d[100], s;
  ...
  s = Sum(d, 100);
  ...
  return 0;
}
```

Pointers and Arrays

- Dynamic Arrays
  - Example 3: Dynamic 1-dim. array
    - ➢ Dynamic allocation
    - ➢ Passed as pointer plus size
    - ➢ Received as pointer!
    - ➢ Accessed via pointer!

```
int Sum(int *p, int m)
{
  int i, sum = 0;
  for(i=0; i<m; i++)
  { sum += *(p + i);
  }
  return sum;
}

int main(void)
{
  int *d, s;
  d = malloc(sizeof(int)*100);
  if (!d)
     { exit(10); }
  ...
  s = Sum(d, 100);
  free(d);
  ...
  return 0;
}
```


Pointers and Arrays

- Dynamic Arrays
  - Example 4: Static 2-dim. array
    - Static definition
    - Passed as static array
    - Static array access
    - ➢ Static sizes everywhere!

```
int Sum(int A[5][20])
{
  int i, j, sum = 0;
  for(i=0; i<5; i++)
    for(j=0; j<20; j++)
    { sum += A[i][j];
    }
  return sum;
}

int main(void)
{
  int d[5][20], s;
  ...
  s = Sum(d);
  ...
  return 0;
}
```

Pointers and Arrays

- Dynamic Arrays
  - Example 5: Mixed 2-dim. array
    - Static definition of dimension 1 (columns)
    - Dynamic allocation of dimension 2 (rows)
    - ➢ Passed as array with dynamic dimension 2 (number of rows) and sizes
    - ➢ Static array access
    - ➢ Multi-dimensional arrays are arrays of arrays…

```
int Sum(int A[][20], int m,int n)
{
  int i, j, sum = 0;
  for(i=0; i<m; i++)
    for(j=0; j<n; j++)
    { sum += A[i][j];
    }
  return sum;
}

int main(void)
{
  int (*d)[20], s;
  d = malloc(sizeof(int[20])*5);
  if (!d)
     { exit(10); }
  ...
  s = Sum(d, 5, 20);
  free(d);
  ...
  return 0;
}
```


Pointers and Arrays

- Dynamic Arrays
  - Example 6: Dynamic 2-dim. array
    - ➢ Dynamic allocation of all dimensions
    - ➢ Passed as pointer
    - ➢ Received as pointer!
    - ➢ Accessed via pointer!
  - ➢ Any array…
    - ➢ Of any dimension
    - ➢ Of any size
  - ➢ …can be mapped into linear address space!

```
int Sum(int *p, int m, int n)
{
  int i, j, sum = 0;
  for(i=0; i<m; i++)
    for(j=0; j<n; j++)
    { sum += *(p + i*n + j);
    }
  return sum;
}

int main(void)
{
  int *d, s;
  d = malloc(sizeof(int)*5*20);
  if (!d)
     { exit(10); }
  ...
  s = Sum(d, 5, 20);
  free(d);
  ...
  return 0;
}
```