

# 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

- **Pointer Operations**
  - Definition, initialization and assignment
  - Pointer dereferencing
  - Pointer arithmetic
    - Increment, decrement
  - Pointer comparison
- **Pointers and Arrays**
  - Equivalence!
  - Array layout in linear address space
- **Validating Dynamic Memory Usage**
  - `valgrind`, a memory error detector

## Pointer Operations

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

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

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

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

EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

3

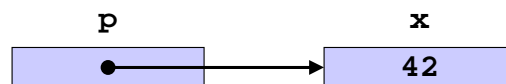
## Pointer Operations

- 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
```



EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

4

## Pointer Operations

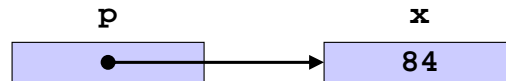
- Pointer Dereferencing
  - The unary \* operator dereferences a pointer to the value it points to (“*content-of*” operator)

```
#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
```



EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

5

## Pointer Operations

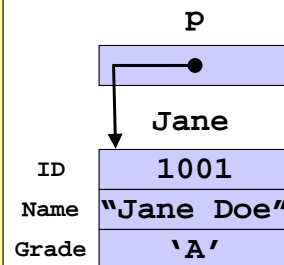
- Pointer Dereferencing
  - The -> operator dereferences a pointer to a structure to the named structure member (“*member-access*” operator)

```
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
```

EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

6

## Pointer Operations

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element
      - Boundaries apply! Pointing outside of `A[0]` to `A[N]` is undefined!

```
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,
```

## Pointer Operations

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element
      - Boundaries apply! Pointing outside of `A[0]` to `A[N]` is undefined!

```
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,
```

## Pointer Operations

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element
      - Boundaries apply! Pointing outside of `A[0]` to `A[N]` is undefined!

```
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,
```

## Pointer Operations

- Pointer Arithmetic
  - Pointers pointing into arrays may be ...
    - ... incremented to point to the next array element
    - ... decremented to point to the previous array element
      - Boundaries apply! Pointing outside of `A[0]` to `A[N]` is undefined!

```
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,
```

## Pointer Operations

- Pointer Comparison
  - Pointers may be compared for object identification or position
    - operators == and != are useful to determine *object identity*
    - operators <, <=, >=, and > are applicable *only to objects in the same array*

```
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!
```

## Pointer Operations

- Pointer Comparison
  - Pointers may be compared for object identification or position
    - operators == and != are useful to determine *object identity*
    - operators <, <=, >=, and > are applicable *only to objects in the same array*

```
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!
```

## Pointer Operations

- Pointer Comparison
  - Pointers may be compared for object identification or position
    - operators == and != are useful to determine object *identity*
    - operators <, <=, >=, and > are applicable *only to objects in the same array*

```
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 points to an element after p2!\n");
  }
if (p1 < p2)
  { printf("p1 points to an element before p2!\n");
  }
```

```
p1 points to an element before p2!
```

EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

13

## 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;
```

EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

14

## Pointers and Arrays

- Dynamic Arrays
  - Example 1: Fixed 1-dim. array
    - Fixed definition
    - Passed as fixed array
    - Fixed array access
    - Fixed 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: Fixed 1-dim. array
    - Fixed definition
    - Passed as fixed array **plus size**
      - Received as pointer and size!
      - Accessed via pointer with offset!

```
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 and size!
      - Accessed via pointer with offset!

```
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;
}
```

EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

17

## Pointers and Arrays

- Dynamic Arrays
  - Example 4:
    - Fixed 2-dim. array
      - Fixed definition
      - Passed as fixed array
      - Fixed array access
      - Fixed 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;
}
```

EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

18

## Pointers and Arrays

- Dynamic Arrays
  - Example 5:
    - Mixed 2-dim. array**
      - Fixed definition of dimension 1 (columns)
      - Dynamic allocation of dimension 2 (rows)
    - Passed as array with dynamic dimension 2 (number of rows) and sizes
    - Fixed 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;
}
```

EECS22: Advanced C Programming, Lecture 13

## 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!
    - An 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;
}
```

EECS22: Advanced C Programming, Lecture 13

## Dynamic Memory Allocation

- Typical Dynamic Memory Usage Errors
  - ⚡ Omitting `malloc()`: Access to unallocated memory
  - ⚡ Reading uninitialized memory
  - ⚡ Omitting `free()`: Memory *leak*
  - ⚡ Freeing memory too early, or multiple times
  - ⚡ ...
- Validating Dynamic Memory Usage
  - `valgrind`: A memory error detector (and more)
    - Instruments the program at (right before) run-time
    - Intercepts and checks calls to `malloc()` and `free()`
    - Intercepts and checks memory accesses
    - Reports any errors to the user (or a log file)
  - Use `valgrind` for testing and debugging!
    - There should be 0 errors and 0 bytes leaked!

EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

21

## Dynamic Memory Allocation

- Example Student Records: `student.h`

```

/* Student.h: header file for student records */

#ifndef STUDENT_H
#define STUDENT_H

#define SLEN 40

struct Student
{
    int ID;
    char Name[SLEN+1];
    char Grade;
};
typedef struct Student STUDENT;

/* allocate a new student record */
STUDENT *NewStudent(int ID, char *Name, char Grade);

/* delete a student record */
void DeleteStudent(STUDENT *s);

/* print a student record */
void PrintStudent(STUDENT *s);

#endif /* STUDENT_H */

```

EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

22

## Dynamic Memory Allocation

- Example Student Records: `student.c` (part 1/3)

```

/* Student.c: maintaining student records */
#include "Student.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>

/* allocate a new student record */
STUDENT *NewStudent(int ID, char *Name, char Grade)
{
    STUDENT *s;
    s = malloc(sizeof(STUDENT));
    if (!s)
    {
        perror("Out of memory! Aborting...");
        exit(10);
    } /* fi */
    s->ID = ID;
    strncpy(s->Name, Name, SLEN);
    s->Name[SLEN] = '\0';
    s->Grade = Grade;
    return s;
} /* end of NewStudent */
...

```

EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

23

## Dynamic Memory Allocation

- Example Student Records: `student.c` (part 2/3)

```

...

/* delete a student record */
void DeleteStudent(STUDENT *s)
{
    assert(s);
    free(s);
} /* end of DeleteStudent */

/* print a student record */
void PrintStudent(STUDENT *s)
{
    assert(s);
    printf("Student ID:    %d\n", s->ID);
    printf("Student Name:  %s\n", s->Name);
    printf("Student Grade: %c\n", s->Grade);
} /* end of PrintStudent */

...

```

EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

24

## Dynamic Memory Allocation

- Example Student Records: `student.c` (part 3/3)

```

...
/* test the student record functions */
int main(void)
{
    STUDENT *s1 = NULL, *s2 = NULL;
    printf("Creating 2 student records...\n");
    s1 = NewStudent(1001, "Jane Doe", 'A');
    s2 = NewStudent(1002, "John Doe", 'C');

    printf("Printing the student records...\n");
    PrintStudent(s1);
    PrintStudent(s2);

    printf("Deleting the student records...\n");
    DeleteStudent(s1);
    s1 = NULL;
    DeleteStudent(s2);
    s2 = NULL;

    printf("Done.\n");
    return 0;
} /* end of main */
/* EOF */

```

EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

25

## Dynamic Memory Allocation

- Example Student Records: `Makefile`

```

# Makefile: Student Records

# macro definitions
CC = gcc
DEBUG = -g
#DEBUG = -O2
CFLAGS = -Wall -ansi $(DEBUG) -c
LFLAGS = -Wall $(DEBUG)

# dummy targets
all: Student

clean:
    rm -f *.o
    rm -f Student

# compilation rules
Student.o: Student.c Student.h
    $(CC) $(CFLAGS) Student.c -o Student.o

Student: Student.o
    $(CC) $(LFLAGS) Student.o -o Student

# EOF

```

EECS22: Advanced C Programming, Lecture 13

(c) 2014 R. Doemer

26

## Dynamic Memory Allocation

- Example Session

```
% vi Student.h
% vi Student.c
% vi Makefile
% make
gcc -Wall -ansi -g -c Student.c -o Student.o
gcc -Wall -g Student.o -o Student
% Student
Creating 2 student records...
Printing the student records...
Student ID: 1001
Student Name: Jane Doe
Student Grade: A
Student ID: 1002
Student Name: John Doe
Student Grade: C
Deleting the student records...
Done.
%
```

## Dynamic Memory Allocation

- Example Session

```
% valgrind Student
==23638== Memcheck, a memory error detector
==23638== [...]
==23638== Command: Student
Creating 2 student records...
Printing the student records...
Student ID: 1001
Student Name: Jane Doe
Student Grade: A
Student ID: 1002
Student Name: John Doe
Student Grade: C
Deleting the student records...
Done.
==23638== HEAP SUMMARY:
==23638==      in use at exit: 0 bytes in 0 blocks
==23638== total heap usage: 2 allocs, 2 frees, 96 bytes allocated
==23638==
==23638== All heap blocks were freed -- no leaks are possible
==23638== ERROR SUMMARY: 0 errors from 0 contexts [...]
%
```