EECS 22: Advanced C Programming

Lecture 16

Rainer Dömer

doemer@uci.edu

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

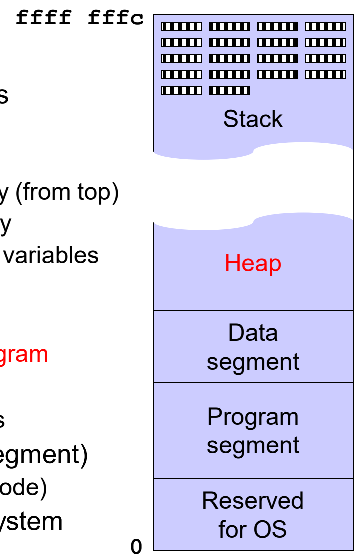
Lecture 16: Overview

- Data Structures
  - Review: Memory organization
  - Objects in memory
  - Pointers
- Dynamic Data Structures
  - Dynamic memory allocation
  - Example: Student records

EECS22: Advanced C Programming, Lecture 16 (c) 2017 R. Doemer 2

Review: Memory Organization

- Memory Segmentation
  - typical (virtual) memory layout on processor with 4-byte words and 4 GB of memory
  - Stack
    - grows and shrinks dynamically (from top)
    - contains function call hierarchy
    - stores stack frames with local variables
  - Heap
    - "free" storage
    - dynamic allocation by the program
  - Data segment
    - global (and **`static`**) variables
  - Program segment (aka. text segment)
    - program instructions (binary code)
  - Reserved area for operating system



EECS22: Advanced C Programming, Lecture 16 (c) 2017 R. Doemer 3

Objects in Memory

- Data in memory is organized as a set of objects
- Every object has ...
  - ... a *type* (e.g. **`int`**, **`double`**, **`char[5]`**)
    - type is known to the compiler at compile time
  - ... a *value* (e.g. **`42`**, **`3.1415`**, **`"text"`**)
    - value is used for computation of expressions
  - ... a *size* (number of bytes in the memory)
    - in C, the **`sizeof`** operator returns the size of a variable or type
  - ... a *location* (address in the memory)
    - in C, the "address-of" operator (**`&`**) returns the address of an object
- Variables ...
  - ... serve as identifiers for objects
  - ... are bound to objects
  - ... give objects a name

EECS22: Advanced C Programming, Lecture 16 (c) 2017 R. Doemer 4

Objects in Memory

- Example: Variable values, addresses, and sizes

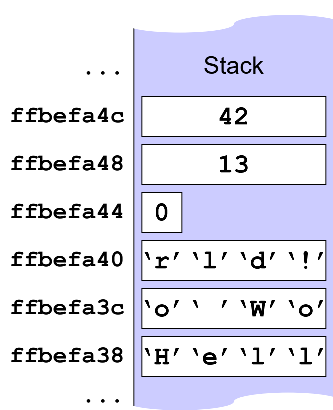
```
int x = 42;
int y = 13;
char s[] = "Hello World!";

printf("Value   of x    is %d.\n", x);
printf("Address of x    is %p.\n", &x);
printf("Size    of x    is %u.\n", sizeof(x));
printf("Value   of y    is %d.\n", y);
printf("Address of y    is %p.\n", &y);
printf("Size    of y    is %u.\n", sizeof(y));
printf("Value   of s    is %s.\n", s);
printf("Address of s    is %p.\n", &s);
printf("Size    of s    is %u.\n", sizeof(s));
printf("Value   of s[1] is %c.\n", s[1]);
printf("Address of s[1] is %p.\n", &s[1]);
printf("Size    of s[1] is %u.\n", sizeof(s[1]));
```


Objects in Memory

- Example: Variable values, addresses, and sizes

```
int x = 42;
int y = 13;
char s[] = "Hello World!";
...
```

```
Value   of x    is 42.
Address of x    is ffbefa4c.
Size    of x    is 4.
Value   of y    is 13.
Address of y    is ffbefa48.
Size    of y    is 4.
Value   of s    is Hello World!.
Address of s    is ffbefa38.
Size    of s    is 13.
Value   of s[1] is e.
Address of s[1] is ffbefa39.
Size    of s[1] is 1.
```



Objects in Memory

- Example: Size and alignment on Linux servers
- 32-bit architecture (2^{32}= 4 GB):

Type	Size	Alignment
`char`	1	1
`short`	2	2
`int`	4	4
`long`	4	4
`long long`	8	4
`float`	4	4
`double`	8	4
`long double`	12	4
`void*`	4	4

- 64-bit architecture (2^{64}= 16 EB) e.g. crystalcove.eecs.uci.edu:

Type	Size	Alignment
`char`	1	1
`short`	2	2
`int`	4	4
`long`	8	8
`long long`	8	8
`float`	4	4
`double`	8	8
`long double`	16	16
`void*`	8	8

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)

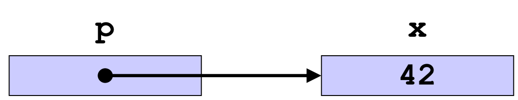
```
#include <stdio.h>

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

EECS22: Advanced C Programming, Lecture 16 (c) 2017 R. Doemer 9

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 */
```



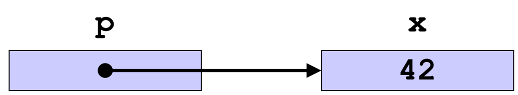
EECS22: Advanced C Programming, Lecture 16 (c) 2017 R. Doemer 10

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
```

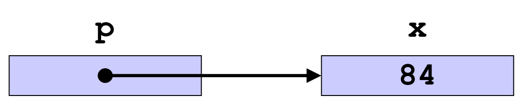


Pointers

- 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);
```

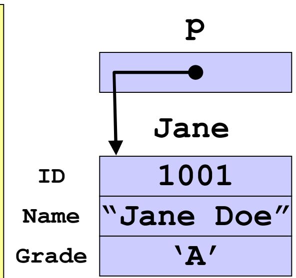
```
x is 42, content of p is 42
x is 84, content of p is 84
```



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
```

EECS22: Advanced C Programming, Lecture 16 (c) 2017 R. Doemer 13

Dynamic Data Structures

- Static Data Structures
  - E.g. arrays, structures
  - Size (and type) known at compile time
  - Compiler automatically allocates memory (linker, loader)
    - Data segment (global/static variables)
    - Stack (local/automatic variables)
- Dynamic Data Structures
  - E.g. lists, trees, graphs
  - Size (and type) not known until run time
  - Programmer manually allocates memory (as needed)
    - Heap (dynamic objects)
  - ➢ *Dynamic Memory Allocation!*
    - ➢ Program explicitly allocates and de-allocates memory
    - ➢ Program explicitly performs memory management functions

EECS22: Advanced C Programming, Lecture 16 (c) 2017 R. Doemer 14

Dynamic Data Structures

- Dynamic Memory Allocation

```
#include <stdlib.h>
void *malloc(size_t size);
```

  - Allocates **size** bytes of memory space on the heap
    - Allocated memory space is uninitialized
  - Returns a pointer to the memory (address of first byte)
    - Return type is **void***, meaning "pointer to unknown type"
    - Return value is **NULL** (**0**) if requested size could not be allocated

```
void free(void *p);
```

  - De-allocates the memory at address **p**
    - Argument **p** must be a pointer to space allocated by **malloc()**
  - Does nothing if **p** is **NULL**

➢ Advise:
  - Always check return value of **malloc()**!
  - Always use **malloc()** and **free()** in pairs!

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 */
```

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 */
...
```


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 */

...
```

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 16 (c) 2017 R. Doemer 19

Dynamic Memory Allocation

- Example Student Records: **Makefile**

```
# Makefile: Student Records

# macro definitions
CC = gcc
DEBUG = -g
#DEBUG = -O2
CFLAGS = -Wall -ansi -std=c99 $(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 16 (c) 2017 R. Doemer 20

Dynamic Memory Allocation

- Example Session

```
% vi Student.h
% vi Student.c
% vi Makefile
% make
gcc -Wall -ansi -std=c99 -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.
%
```