6.S096: Introduction to C/C++

Frank Li, Tom Lieber, Kyle Murray

Lecture 4: Data Structures and Debugging!

January 17, 2012

Today…

- Memory Leaks and Valgrind Tool
- Structs and Unions
- Opaque Types
- Enum and Typedef
- GDB Debugging Tool

Today…

- **Memory Leaks and Valgrind Tool**
- Structs and Unions
- Opaque Types
- Enum and Typedef
- GDB Debugging Tool

Memory Issues (Live Demos!)

- Illegal memory accesses (segmentation faults)
- Stack overflow
  - Infinite recursions
- Double frees
- Dangerous use of uninitialized variables (see Lec 2)
- Memory leaks

Memory Leaks

- Since memory is user managed, there can be mistakes.
- Memory leaks = allocated memory that's not freed.
- Why it matters?

Valgrind

- A memory profiling tool. Helps you find memory leaks.
- Compile with debug mode (using -g flag).

  gcc -g hello.c -o hello

- Includes tools such as memcheck, cachegrind, callgrind, etc…
- Command line:
  - valgrind --tool=*tool_name* ./program_name
  - Example: valgrind --tool=memcheck ./hello_world

Valgrind No Leaks Demo

- $ valgrind --tool=memcheck ./hello_world

  …

  =18515== malloc/free: in use at exit: 0 bytes in 0 blocks.

  ==18515== malloc/free: 1 allocs, 1 frees, 10 bytes allocated.

  ==18515== For a detailed leak analysis, rerun with: --leak-check=yes

Valgrind Leak Demo

```
#include <stdlib.h>
int main()
 {
    char *x = (char*) malloc(100);  //Mem Leak!
    return 0;
}
```

Valgrind No Leaks Demo

- $ valgrind --tool=memcheck ./mem_leak

...

```
==12196== HEAP SUMMARY:
==12196==     in use at exit: 100 bytes in 1 blocks
==12196==   total heap usage: 1 allocs, 0 frees, 100 bytes allocated
==12196==
==12196== LEAK SUMMARY:
==12196==    definitely lost: 100 bytes in 1 blocks
==12196==    indirectly lost: 0 bytes in 0 blocks
==12196==      possibly lost: 0 bytes in 0 blocks
==12196==    still reachable: 0 bytes in 0 blocks
==12196==         suppressed: 0 bytes in 0 blocks
```

More Valgrind Functionalities

- Can also detect invalid pointer use
  - char *arr = malloc(10);

    arr[10] = ‘a’
- Using uninitialized variables
  - int x;

    if(x==0){…}
- Double/invalid frees
- Valgrind doesn’t check bounds on statically allocated arrays though!

Today…

- Memory Leaks and Valgrind Tool
- **Structs and Unions**
- Opaque Types
- Enum and Typedef
- GDB Debugging Tool

User Defined Data Types

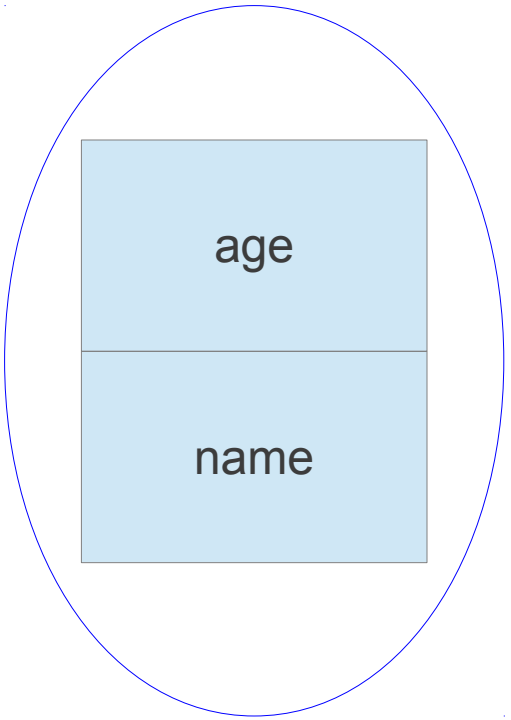
- C has no objects, but has data structures that can help fill in roles. Only contains data.

- Structures and unions are like data objects
  - Contains groups of basic data types
  - Can be used like any normal type
  - Not true objects b/c they can't contain functions

Structures

- Declaring structs

```
struct person{
    int age;
    char name[50];
};
struct person joe;
```



A person!

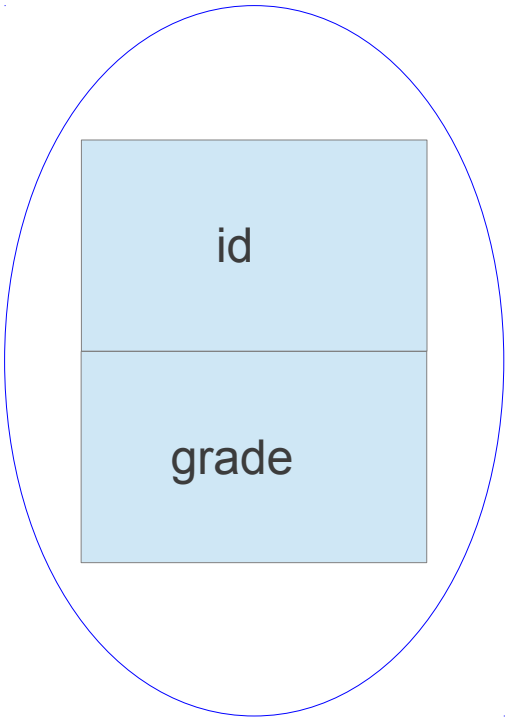
- Accessing members: . operator
  - instance_name.struct_member
  - joe.age

Structures

- Initializing structs
  - struct student {
    int id;
    char grade;
    };
    struct student frank= {1, 'A'};

Or

    struct student frank;
    frank.id = 1;
    frank.grade = 'A';



Hmm…is that all I am?
A number and a letter?

:(

Structures

- Struct pointers:
  - struct student * frank= malloc(sizeof(struct student));
- Accessing members of struct pointers: ->
  - frank->grade = 'A' // Hooray (for me)!
- Can have structs within structs. Just use more . or -> operators to access inner structs.

Struct Memory

- Struct size != sum of member sizes
- All members must “align” with largest member size
- Each member has own alignment requirements

  Ex: char = 1-byte aligned.

  short = 2-byte aligned.

  int = 4-byte aligned. ←Refer to documentation

Struct Memory

- Blackboard Example:

```
struct x{
    char a; //1-byte, must be 1-byte aligned
    short b; //2-bytes, must be 2-byte aligned
    int c;  // Biggest member (4-bytes). X must be 4-byte
            // aligned
    char d;
}
```

Unions

- Can only access one member at a time. Union stores all data in same chunk of memory.

```
union data{
    int x; char y;
};
union data mydata;
mydata.y = 'a';
mydata.x = 1;
printf("%d\n", mydata.x) // Will print out 1
printf("%c\n", mydata,y) // Will print out JUNK!
```

Common Use of Unions

- Union within struct

```
struct student{
    union grade{
        int percentage_grade; //ex: 99%
        char letter_grade; // 'B'
    };
    int grade_format; // I set to 0 if grade is int, 1 if grade is
    char
};
struct student frank;
frank.grade.percentage_grade = 90;
frank.grade_format = 0;
```

Today…

- Memory Leaks and Valgrind Tool
- Structs and Unions
- **Opaque Types**
- Enum and Typedef
- GDB Debugging Tool

Opaque Types

- Type exposed via pointers where definition can still change. Ex: Can change struct person definition in test.c without recompiling my_file.c

test.h:

```
struct person;
```

test.c:

```
struct person{
    … //def here
};
```

my_file.c:

```
#include “test.h”
int person_function(struct person * ptr){
    …
}
```

Today…

- Memory Leaks and Valgrind Tool
- Structs and Unions
- Opaque Types
- **Enum and Typedef**
- GDB Debugging Tool

Typedef

- Typedef assigns alternate name to existing type.
  - typedef existing_type alternate_name

    typedef int seconds_t;

    seconds_t x = 3;

    typedef *struct person* person;

    person frank; // instead of struct person frank
- Good for shorthand and code readability
- Opaque types

    typedef *struct person** person;

    int func(person me){ …}

Enum

- Define own variable type with a set of int values

```
enum time_t { morning, noon, afternoon, night};
enum time_t class = morning;
if (class == afternoon) { … }
```

- What actually happens is enum values are mapped to increasing sequence of int:
  - morning = 0, noon = 1, afternoon = 2, night = 3
  - You can explicitly assign int values:

```
enum time_t { morning, noon=2, afternoon, night};
```

morning = 0, noon = 2, afternoon = 3, night = 4

Why use Enum

- Like a #define variable but is actually a C element (has a type, obeys scoping rules)

```
#define NORTH 0
#define EAST 1
#define SOUTH 2
#define WEST 3

int direction = SOUTH;
//Compiler sees:
//int direction = 2
```

```
enum dir_t{
    NORTH,
    EAST,
    SOUTH,
    WEST
};

typedef enum dir_t dir_t;
dir_t direction = SOUTH;
```

Today…

- Memory Leaks and Valgrind Tool
- Structs and Unions
- Opaque Types
- Enum and Typedef
- **GDB Debugging Tool**

Debugger (Your LifeSaver)

- Compile with –g flag (debug mode)

  gcc -g hello.c -o hello

- Command-line debugger: gdb ./prog_name

  gdb ./hello

GDB Example

```
C:\Documents and Settings\U0030494\Desktop>gcc -g helloworld.c -o helloworld.exe


C:\Documents and Settings\U0030494\Desktop>helloworld
Hello World!
C:\Documents and Settings\U0030494\Desktop>gdb helloworld
GNU gdb (GDB) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "mingw32".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from C:\Documents and Settings\U0030494\Desktop/helloworld.exe..
.done.
(gdb) r
Starting program: C:\Documents and Settings\U0030494\Desktop/helloworld.exe
[New Thread 5848.0x12a0]
Hello World!
Program exited with code 014.
(gdb) q

C:\Documents and Settings\U0030494\Desktop>_
```

Debugger (Live Demo!)

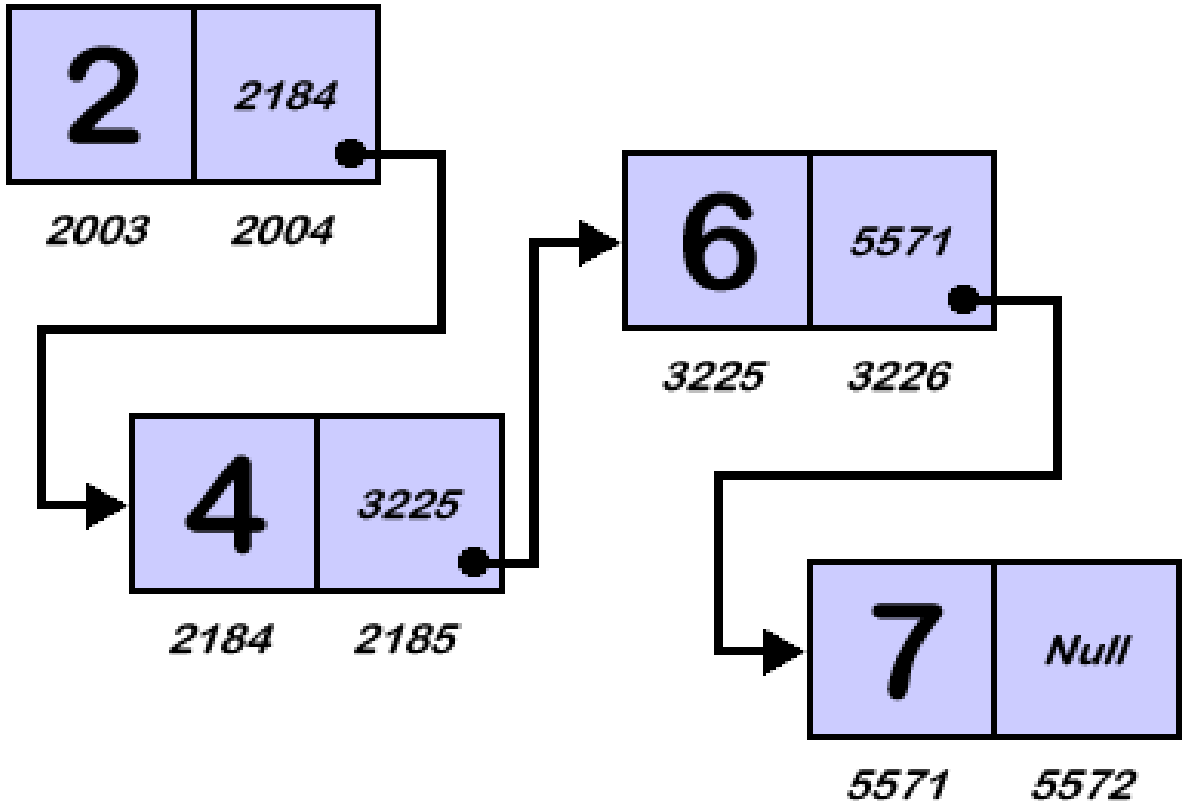
Useful commands:

- q/quit: exit GDB
- r/run: run program
- b/break (filename:)linenumber: create a breakpoint at the specified line.
- s/step: execute next line, or step into a function
- c/continue: continue execution
- p variable: print the current value of the variable
- bt: print the stack trace (very useful!)

Summary

- Use Valgrind and GDB to find errors/memory bugs
- C structs and unions are like data “objects”
- Opaque types allow flexibility in struct/union usage
- Enum and Typedef saves you time!

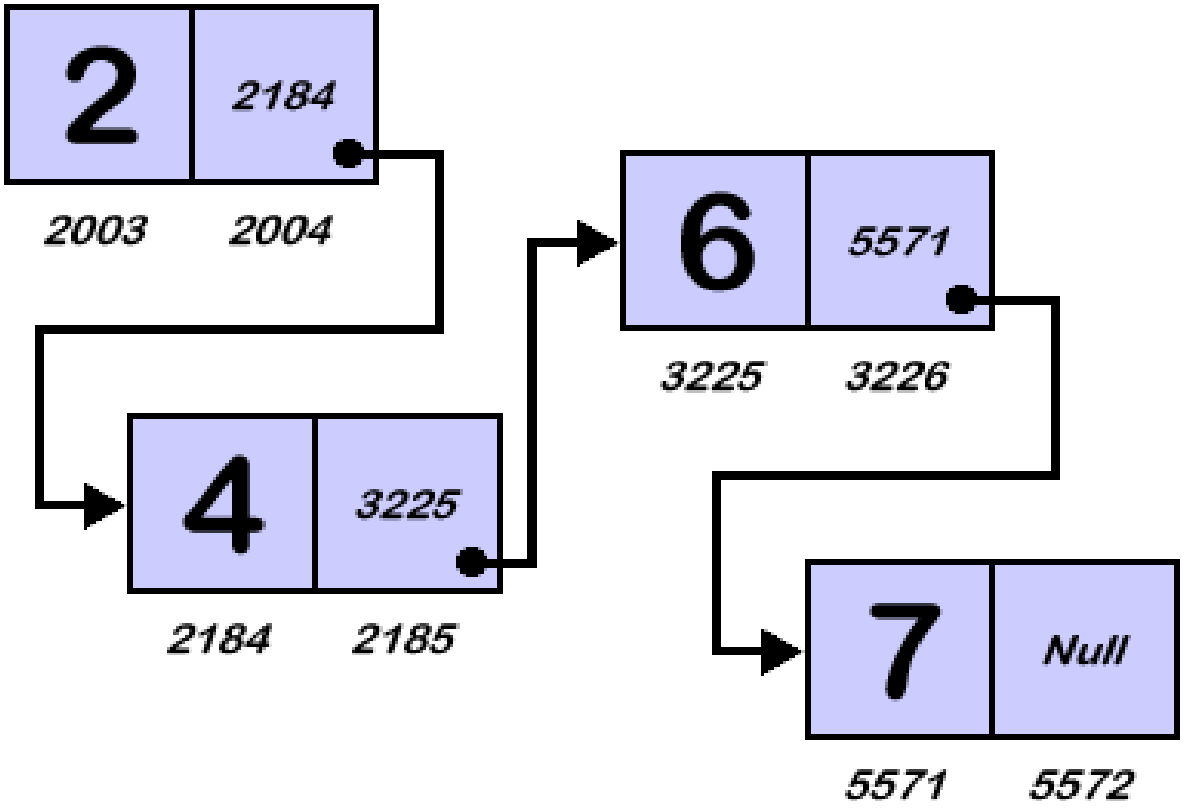
Advanced Data Type Example



Courtesy of Osman Balci, Virginia Tech Center for Innovation in Learning. Used with permission.

Image from: http://courses.cs.vt.edu/csonline/DataStructures/Lessons/OrderedListImplementationView/Lesson.html

The Linked List!

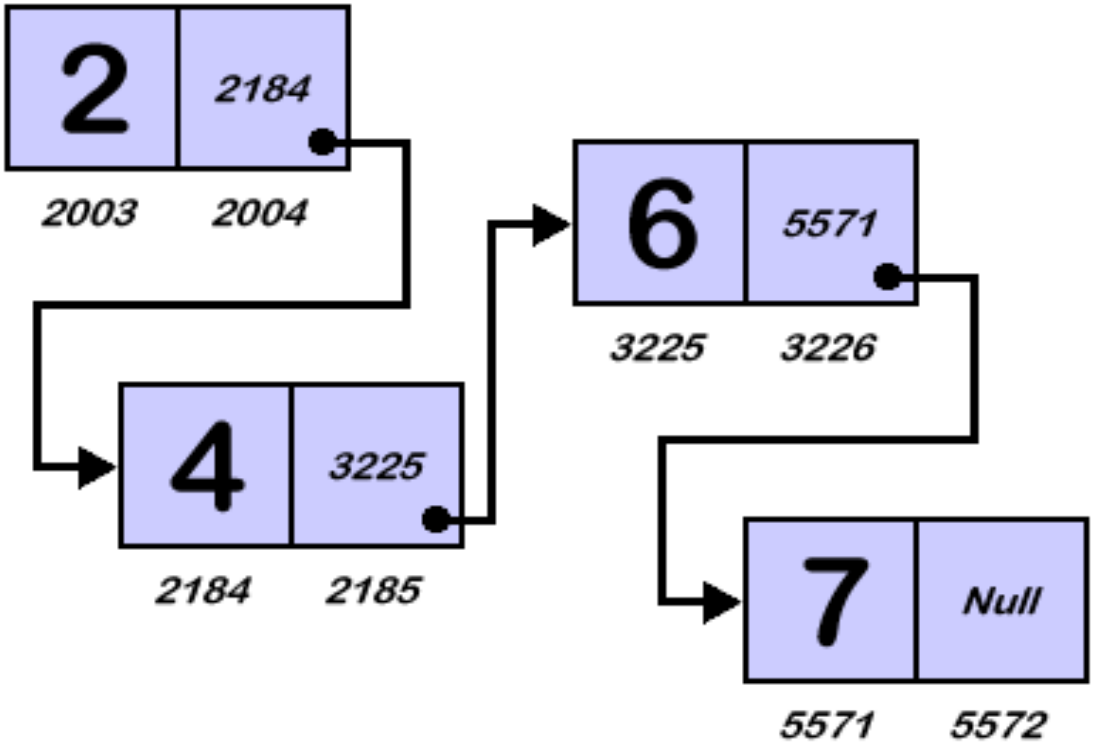


Courtesy of Osman Balci, Virginia Tech Center for Innovation in Learning. Used with permission.

Image from: http://courses.cs.vt.edu/csonline/DataStructures/Lessons/OrderedListImplementationView/Lesson.html

Singly Linked Lists

- Singly linked list is a sequential list of nodes where each node contains a value and a link/pointer to the next node.



Courtesy of Osman Balci, Virginia Tech Center for Innovation in Learning. Used with permission.

Image from: http://courses.cs.vt.edu/csonline/DataStructures/Lessons/OrderedListImplementationView/Lesson.html

Singly Linked Lists

- Nodes:

```
struct node{
    int data;
    struct node* next;
};
typedef struct node node;
node* head = NULL;  //Points to beginning of list. Set
                    // to null initially.
```

Singly Linked Lists

- Adding new data to list

```
node* add_data(int data){
    node* new_node = (node*) malloc(sizeof(node));
    new_node->data = data;
    new_node->next = head;
    head = new_node;
    return new_node;
}
```

Singly Linked Lists

- Searching through list

```
node * find_data(int data){
    node* current;
    for( current = head; current->next!=NULL; current= current->next){
        if(current->data == data) return current;
    }
    return NULL;
}
```

Singly Linked Lists

- Removing a certain data value

```
void rm_data(int data){
    //Special case if the head has the data
    if( head->data == data ) {
        node* tmp = head;
        head = head->next;
        free(tmp);
        return;
    }
    ...//Code continues on next slide
}
```

Singly Linked Lists

- Removing a certain data value

```
void rm_data(int data){
    ...//Code from previous slide
    node* prev, *current;
    for(current = head; current->next!=NULL; current= current->next){
       if(current->data == data){
          prev->next = current->next;
          free(current);
          return ;
       }
       prev = current;
    }
}
```

6.S096 Introduction to C and C++

IAP 2013