CSO-Recitation 07

CSCI-UA 0201-007

R07: Assessment 05 & Assembly & lab2

Today’s Topics

- Assessment 05
- Assembly
- More about lab2
  - More debugging
  - Some valuable questions asked
- Some exercises

Assessment 05

Q1 ASCII

Suppose char c stores some ASCII character. What could be its value interpreted as a signed 1-byte integer?

A. any integer in the range [-128,127]

B. any integer in the range [0, 255]

C. any integer in the range [0, 127]

D. any integer in the range [-1, 255]

- ASCII characters:
- use one byte (with MSB=0) to represent each character

- if it is interpreted as a signed 1-byte int:
  - smallest: 00000000 -> 0
  - largest: 011111111 -> 127

Q2 String

```
1: char c = 'a';
2: int x = strlen(&c);
```

What's the value of x after the above two lines of code?

A. Compilation error at line 1
B. Compilation error at line 2
C. x = 0
D. x = 1
E. x = 2
F. x = 3
G. x's value is undefined (i.e. could be any int value).

- What is C's solution to determine string length?
  - Programmers are expected to store a NULL character at the end of the string (by convention)
  - Count the #char until '\0'

Q3 String

```
1: char c = '\0';
2: int x = strlen(&c);
```

What's the value of x after the above two lines of code?

A. Compilation error at line 1
B. Compilation error at line 2
C. x = 0
D. x = 1
E. x = 2
F. x = 3
G. x's value is undefined (i.e. could be any int value).

- What is C's solution to determine string length?
  - Programmers are expected to store a NULL character at the end of the string (by convention)
  - Count the #char until '\0'

Q4 String

Exercise:
- what if `int x = strlen(&a);` ?
- what if ` int a = 0x01414243;`?

```
1: int a = 0x00414243;
2: int x = strlen((char *)&a);
```

What's the value of x after the above two lines of code?

A. Compilation error at line 1
B. Compilation error at line 2
C. x = 0
D. x = 1
E. x = 2
F. x = 3
G. x's value is undefined (i.e. could be any int value).

- What is C's solution to determine string length?
  - Programmers are expected to store a NULL character at the end of the string (by convention)
  - Count the #char until '\0'
- (char *)&a -> casting to char *

Q5 Hex symbol to Int

Function hex_symbol_to_int converts a hex symbol (in ASCII character) to its corresponding integer value from 0 to 15. For example, hex_symbol_to_int('1') should return (1-byte) integer with value 1; hex_symbol_to_int('b') should return (1-byte) integer with value 11.

```
char hex_symbol_to_int(char h)
{
   char r = -1;
   if (h >= '0' && h <= '9') {
L1:
      r = ???
   } else if (h >= 'a' && h <= 'f') {
L2:
      r = ???
   } else if (h >= 'A' && h <= 'F') {
L3:
      r = ???
   }
   return r;
}
```

What should be the **right hand side** of the assignment statement at Label L1/L2/L3 in function hex_symbol_to_int?

Q5 Hex symbol to Int

What should be the **right hand side** of the assignment statement at Label L1/L2/L3 in function hex_symbol_to_int?

```
char hex_symbol_to_int(char h)
{
   char r = -1;
   if (h >= '0' && h <= '9') {
L1:
      r = ???
   } else if (h >= 'a' && h <= 'f') {
L2:
      r = ???
   } else if (h >= 'A' && h <= 'F') {
L3:
      r = ???
   }
   return r;
}
```

- L1:
  - h-'0';
  - h-48; // r=h-'0'; // r=h-48;
  - wrong: h-47; // r-’0’; // r-48; // lose the statement terminator (;)
- L2:
  - h-'a'+10; // h-'a'+('9'-'0')+1;
  - h-87; // h-'W';
  - wrong: h-86; // r-’0’; // r-87; // lose the statement terminator (;)
- L3:
  - h-'A'+10; // h-’A'+('9'-'0')+1;
  - h-55; // h-'7';

Q6 Hex string to int

Function hex_string_to_int converts a 8-character hex string to its corresponding 4-byte int value (We assume the hex string does not contain the hex notation prefix "0x").

```
int hex_string_to_int(char *s)
{
   assert(strlen(s)==8);
   int result = 0;
   while (*s) {
      char v = hex_symbol_to_int(*s);
L0:
      //to be completed by you
   }
   return result;
}
```

Suppose int x = hex_string_to_int("ffffffff"), what should be the value of x if hex_string_to_int is implemented correctly?

A. 2^{32}-1
B. 2^{31}-1
C. -1
D. 0
E. -2^{31}

Q6 Hex string to int

Function hex_string_to_int converts a 8-character hex string to its corresponding 4-byte int value (We assume the hex string does not contain the hex notation prefix "0x").

```c
int hex_string_to_int(char *s)
{
    assert(strlen(s)==8);
    int result = 0;
    while (*s) {
        char v = hex_symbol_to_int(*s);
L0:
        //to be completed by you
    }
    return result;
}
```

Completing the loop body at Label L0 (Code may require more than 1 line):

```c
result = (result << 4) + v; // result=(result<<4) | (v&0xF);
s++;
```

v & 0xF -> mask off the left-4-bits of v

result << 4 first

(result << 4) | (v & 0xF) -> turn some bits of result on

Q6 Hex string to int

Function hex_string_to_int converts a 8-character hex string to its corresponding 4-byte int value (We assume the hex string does not contain the hex notation prefix "0x").

```c
int hex_string_to_int(char *s)
{
    assert(strlen(s)==8);
    int result = 0;
    while (*s) {
        char v = hex_symbol_to_int(*s);
L0:
        //to be completed by you
    }
    return result;
}
```

Completing the loop body at Label L0 (Code may require more than 1 line):

```c
result = 16 * result + v;
s++;
```

- s= “123abc”
- *s==‘1’
- v -> hex_symbol_to_int(‘1’) = 1
- result=16*0+1=1
- *s -> ‘2’ (s++;)
- v -> 2
- result=1*16+2
- …

Assembly

C is for people

Why Assembly

- In the real world, computers don’t “understand” code
- They only “understand” a set of instructions
- To run code
  - 1. The CPU fetches an instruction from the memory at the PC(program counter)
  - 2. The CPU decodes that instruction
  - 3. If needed, the CPU fetches data from memory
  - 4. The CPU performs computations
  - 5. If needed, the CPU writes data to memory
  - 6. The CPU increments the PC to the next instruction

Why Assembly

- Computers don’t “understand” assembly either, but assembly maps much more closely to machine instructions than C code
- Assembly code involves instruction “mnemonics”
  - For x86_64, These are things like addq, movq, imul

X86 general purpose registers

- Accessing memory is very, very slow compared to the rest of what a CPU can do
- Registers are fast temporary storage
- X86-64 ISA: 16 8-byte general purpose registers
- Originally there were 8, all 16-bits large
    - %ax, %bx, %cx, %dx, %si, %di, %bp, %sp
    - These have 32-bit counterparts – add an e, eg %eax, %esp
    - These also have 64-bit counterparts – add an r, eg %rax, %rsp
- With 64 bits came 8 more registers, %r8 to %r15
    - These have 32-bit counterparts - add a d, eg %r8d
    - These have 16-bit counterparts – add a w, eg %r8w
- All registers also allow you to access their lowest 8 bits
- %ax, %bx, %cx, and %dx, allow you to access their upper 8 bits

Important Instructions

Instruction	What it does
mov src, dest	dest = src
add src, dest	dest = dest + src
sub src, dest	dest = dest - src
imul src, dest	dest = dest * src
inc dest	dest = dest + 1

More about lab2

Debugging & Some valuable questions

More on debugging

1. Program received signal SIGSEGV, Segmentation fault.
   - GDB will tell you where your code segfaulted
   - GDB can tell you what values are what
     - why your code segfaulted

Debugging a crash

- *run* your program
- Use *bt* to see the call stack
  - You can also use *where* to see where you were last running
- Use *frame* to go to where your code was last running
- Use *list* to see the code that ran
- Check the locals (*info locals*) and args (*info args*) to see if they are bad

More on debugging

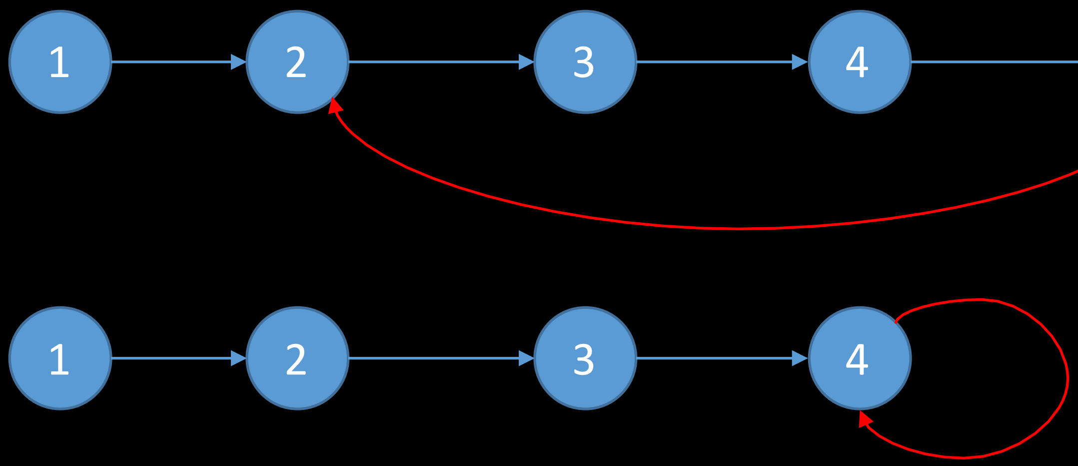
1. Program received signal SIGSEGV, Segmentation fault.
   - GDB will tell you where your code segfaulted
   - GDB can tell you what values are what
     - why your code segfaulted
2. Program get stuck
   - infinite loop

Debugging an infinite loop

- Just *run* it inside gdb and hit control-c (signal)
- *list* the code
  - This is so you can see the loop condition
- *step* over the code
- Check (*print*) the values involved in the loop condition
  - Are they changing the right way? Are the variables changing at all?

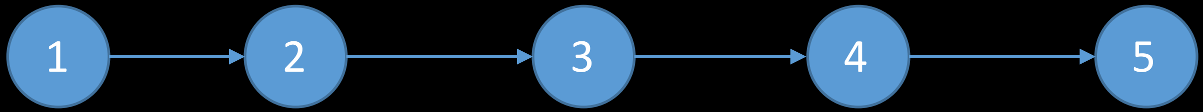
Loop in a linked list

- When I insert a node into the linked list, what will cause a loop?

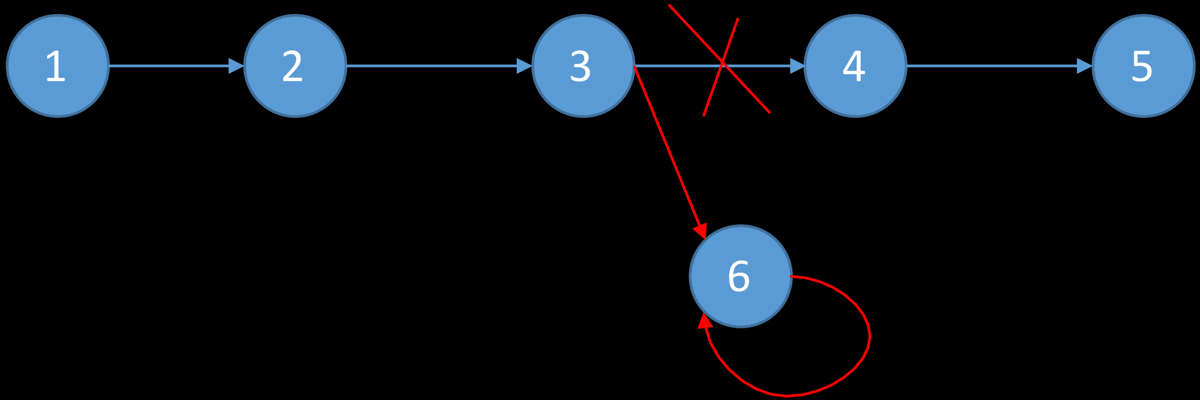


Loop in a linked list

- When I insert a node into the linked list, what will cause a loop?

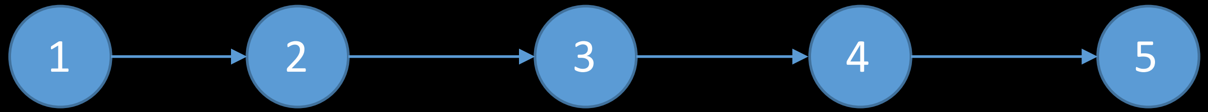


- I need to insert n6 between n3 and n4:
  - suppose our *head* now points to n3
  - head -> next = n6
  - n6 -> tuple.key = keys
  - n6 -> tuple.value= value
  - n6 -> next = head->next

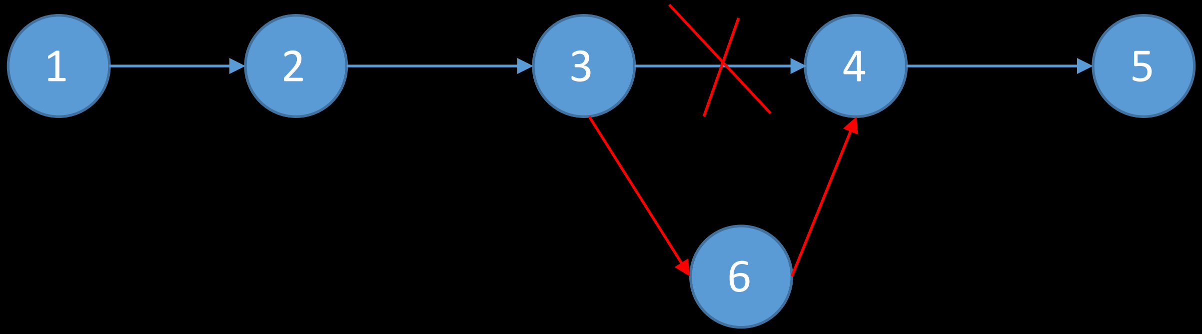


Loop in a linked list

- When I insert a node into the linked list, what will cause a loop?



- I need to insert n6 between n3 and n4:
  - suppose our *head* now points to n3
  - n6 -> next = head->next
  - n6 -> tuple.key = keys
  - n6 -> tuple.value= value
  - head -> next = n6



More on debugging

1. Program received signal SIGSEGV, Segmentation fault.
    - GDB will tell you where your code segfaulted
    - GDB can tell you what values are what
        - why your code segfaulted
2. Program get stuck
    - infinite loop
3. GDB can print structs!
    - p *head will print the fields of the struct pointed to by *head*
4. GDB can interpret numbers however you tell it to!
    - Use the x command to view the data at a memory address
        - x buf means print the value at *buf*
    - x/10b means print 10 (10) bytes (b) – can be used to “x/8b $rdi”
    - use p command with /x to print number in hex notation: p /x val

Function pointer

- In lab2, we invoke the function by function pointer *accum*
- Like normal data pointers (int *, char *, ..), we can have pointers to functions
  - A function pointer points to code, not data. Typically a function pointer stores the start of executable code
  - A function’s name can also be used to get functions’ address
  - In general, function pointer refer to functions of any signature. return type does not necessarily have to be void
  - Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function
    - why this useful?

Function pointer

- Like normal data pointers, a function pointer can be passed as an argument and can also be returned from a function
- We can invoke different functions into one function by using a function pointer
  - as long as the different functions using the same parameters and have the same return types
  - In C, we can use function pointers to avoid code redundancy

Testing

- When you fail in one test case, it does not mean you can only have bugs in this function implementation
  - Even if you have passed the previous test cases..
- No one test can help you test all possible bugs in your code
- Led to an interesting research topic:
  - Proof of Program Correctness