CS 33

Introduction to C
Part 4

Boolean Algebra

- **Developed by George Boole in 19th Century**
  - **algebraic representation of logic**
    - » **encode "true" as 1 and "false" as 0**

And

- A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

- A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Not

- ~A = 1 when A=0

~	
0	1
1	0

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

Supplied by CMU.

General Boolean Algebras

- **Operate on bit vectors**
  - operations applied bitwise

```
  01101001     01101001     01101001
& 01010101   | 01010101   ^ 01010101   ~ 01010101
  --------     --------     --------     --------
  01000001     01111101     00111100     10101010
```

- **All of the properties of boolean algebra apply**

Supplied by CMU.

Example: Representing & Manipulating Sets

- **Representation**
  - width-w bit vector represents subsets of {0, …, w–1}
  - $a_j = 1$ iff $j \in A$

01101001 *76543210*	{ 0, 3, 5, 6 }
01010101 *76543210*	{ 0, 2, 4, 6 }

- **Operations**

&	intersection	01000001	{ 0, 6 }
|	union	01111101	{ 0, 2, 3, 4, 5, 6 }
^	symmetric difference	00111100	{ 2, 3, 4, 5 }
~	complement	10101010	{ 1, 3, 5, 7 }

Supplied by CMU.

Bit-Level Operations in C

- Operations &, |, ~, ^ available in C
  - apply to any “integral” data type
    - » `long, int, short, char`
  - view arguments as bit vectors
  - arguments applied bit-wise
- Examples (char datatype)

`~0x41` → `0xBE`

  `~01000001`$_2$ → `10111110`$_2$

`~0x00` → `0xFF`

  `~00000000`$_2$ → `11111111`$_2$

`0x69 & 0x55` → `0x41`

  `01101001`$_2$ & `01010101`$_2$ → `01000001`$_2$

`0x69 | 0x55` → `0x7D`

  `01101001`$_2$ | `01010101`$_2$ → `01111101`$_2$

Supplied by CMU.

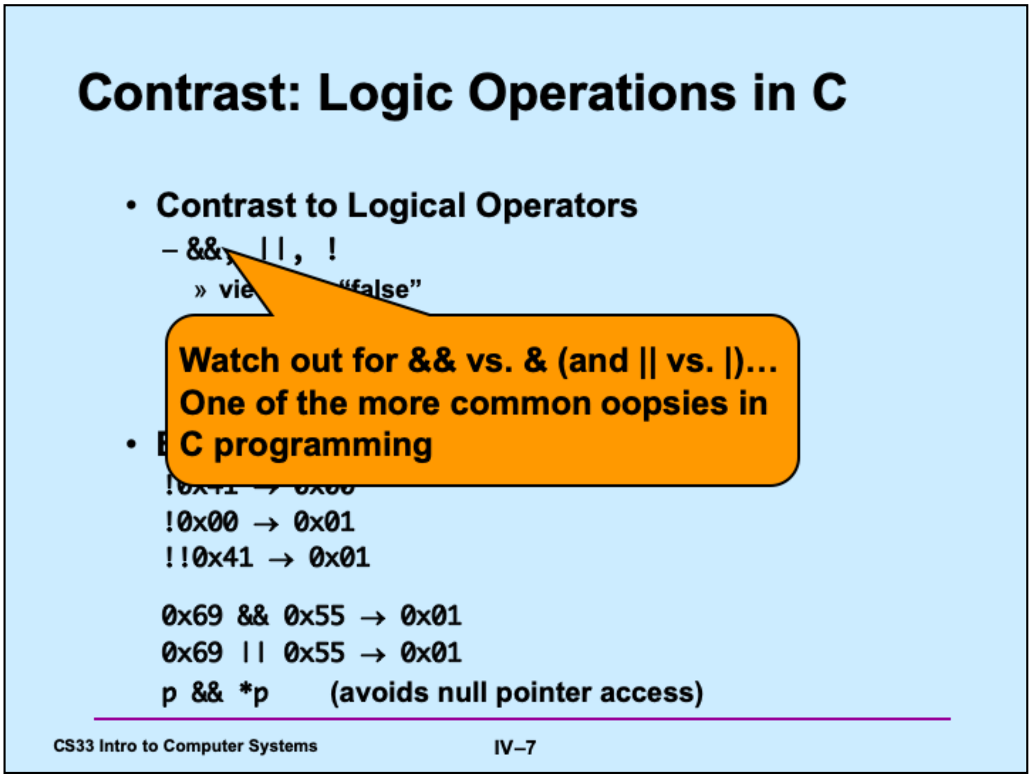
Contrast: Logic Operations in C

- **Contrast to Logical Operators**
  - `&&, ||, !`
    - » view 0 as “false”
    - » anything nonzero as “true”
    - » always return 0 or 1
    - » early termination/short-circuited execution
- **Examples (char datatype)**

`!0x41` → `0x00`
`!0x00` → `0x01`
`!!0x41` → `0x01`

`0x69 && 0x55` → `0x01`
`0x69 || 0x55` → `0x01`
`p && *p` (avoids null pointer access)

Supplied by CMU.



Supplied by CMU.

Quiz 1

- Which of the following would determine whether the next-to-the-rightmost bit of Y (declared as a char) is 1? (I.e., the expression evaluates to true if and only if that bit of Y is 1.)
  a) Y & 0x02
  b) !((~Y) & 0x02)
  c) both of the above
  d) none of the above

Recall that a char is an 8-bit integer.

Shift Operations

- **Left Shift: x << y**
  - shift bit-vector x left y positions
    - throw away extra bits on left
    - » fill with 0's on right
- **Right Shift: x >> y**
  - shift bit-vector x right y positions
    - » throw away extra bits on right
  - logical shift
    - » fill with 0's on left
  - arithmetic shift
    - » replicate most significant bit on left
- **Undefined Behavior**
  - shift amount < 0 or ≥ word size

Argument x	01100010
<< 3	00010*000*
Log. >> 2	*00*011000
Arith. >> 2	*00*011000

Argument x	10100010
<< 3	00010*000*
Log. >> 2	*00*101000
Arith. >> 2	*11*101000

Supplied by CMU.

Why we need both logical and arithmetic shifts should be clear by the end of an upcoming lecture. If one is applying a right shift to an *int*, it will be an arithmetic right shift. For unsigned *int*s, right shifts are logical right shifts. Why this is so will be explained in the upcoming lecture (it has to do with the representation of negative numbers).

Global Variables

The scope is global;
m can be used
by all functions

```
#define NUM_ROWS 3
#define NUM_COLS 4
int m[NUM_ROWS][NUM_COLS];

int main() {
   int row, col;
   for(row=0; row<NUM_ROWS; row++)
     for(col=0; col<NUM_COLS; col++)
        m[row][col] = row*NUM_COLS+col;
   return 0;
}
```

Global Variables

```
#define NUM_ROWS 3
#define NUM_COLS 4
int m[NUM_ROWS][NUM_COLS];

int main() {
   int row, col;
   printf("%u\n", m);
   printf("%u\n", &row);
   return 0;
}
```

```
$ ./a.out
8384
3221224352
```


Note that the reference to “m” gives the address of the array in memory.

The point of the slide is that global variables are in a different area of memory than are local variables.

Global Variables are Initialized!

```
#define NUM_ROWS 3
#define NUM_COLS 4
int m[NUM_ROWS][NUM_COLS];

int main() {
   printf("%d\n", m[0][0]);
   return 0;
}
```

```
$ ./a.out
0
```


If you don't explicitly initialize a global variable, its initial value is guaranteed to be zero.

Scope

```
int a;    // global variable

int main() {
   int a;    // local variable
   a = 0;
   proc();
   printf("a = %d\n", a); // what's printed?
   return 0;
}

int proc() {
   a = 1;
   return a;
}
```

```
$ ./a.out
0
```


Here we have two declarations for *a* – one as a global variable and one as a local variable. References to *a* in *main* are to the local variable, but elsewhere references are to the global variable.

Scope (continued)

```
int a;   // global variable

int main() {
   a = 2;
   proc(1);
   return 0;
}

int proc(int a) {
   printf("a = %d\n", a); // what's printed?
   return a;
}
```

```
$ ./a.out
1
```


Here *a* is declared as a parameter to *proc*, thus references to *a* in *proc* are to the parameter and not to the global variable.

Scope (still continued)

```c
int a;   // global variable

int main() {
   a = 2;
   proc(1);
   return 0;
}

int proc(int a) {
   int a;
   printf("a = %d\n", a); // what's printed?
   return a;
}
```

```
$ gcc prog.c
prog.c:12:8: error: redefinition of 'a'
   int a;
       ^
```


Syntax error: one can't have a local variable in a scope in which a parameter is declared with the same name.

Scope (more ...)

```
int a;    // global variable

int proc() {
   {
      // the brackets define a new scope
      int a;
      a = 6;
   }
   printf("a = %d\n", a); // what's printed?
   return 0;
}
```

```
$ ./a.out
0
```

Quiz 2

```
int a;

int proc(int b) {
   {int b=4;}
   a = b;
   return a+2;
}

int main() {
   {int a = proc(6);}
   printf("a = %d\n", a);
   return 0;
}
```

- **What's printed?**
  - a) 0
  - b) 4
  - c) 6
  - d) 8
  - e) nothing; there's a syntax error

Scope and For Loops (1)

```
int A[100];
for (int i=0; i<100; i++) {
  // i is defined in this scope
  A[i] = i;
}
```


It's often convenient to declare a for loop's index variable in the for loop, as shown in the slide.

Scope and For Loops (2)

```
int A[100];
initializeA(A);
for (int i=0; i<100; i++) {
  // i is defined in this scope
  if (A[i] < 0)
    break;
}
if (i != 100)
  printf("A[%d] is negative\n", i);
```

syntax error: reference to *i* is out of scope.

But be careful – the scope of such an index variable does not extend outside of the for loop.

Lifetime

```
int count;

int main() {
    func();
    ...
    func(); // what's printed by func?
    return 0;
}

int func() {
    int a;
    if (count == 0) a = 1;
    count = count + 1;
    printf("%d\n", a);
    return 0;
}
```

```
% ./a.out
1
-38762173
```


Even though *a* is given a value the first time *func* is called, on *func*'s second invocation *a* is not given a value and thus the result that's printed is "undefined". This is because the lifetime of *a* is just for the length of time its scope is active, which is from when the execution of *func* starts to when *func* returns. The *a* in the next invocation of *func* is different from the previous *a*.

Lifetime (continued)

```
int main() {
   func(1); // what's printed by func?
   return 0;
}
int a;
int func(int x) {
   if (x == 1) {
      a = 1;
      func(2);
      printf("%d\n", a);
   } else
      a = 2;
   return 0;
}
```

```
% ./a.out
2
```


In this case, *a* is global and thus the value set for it in one invocation of *func* is still there for the next invocation – the lifetime of *a* is that of the program itself.

Lifetime (still continued)

```
int main() {
   func(1); // what's printed by func?
   return 0;
}

int func(int x) {
   int a;
   if (x == 1) {
      a = 1;
      func(2);
      printf("a = %d\n", a);
   } else
      a = 2;
   return 0;
}
```

```
% ./a.out
1
```


Here *a* is local again. *func* is called (recursively) from within itself: the recursive invocation of *func* modifies a different *a* than is used in the first invocation. Thus the value printed is 2.

Lifetime (more ...)

```
int main() {
   int *a;
   a = func();
   printf("%d\n", *a); // what's printed?
   return 0;
}

int *func() {
   int x;
   x = 1;
   return &x;
}
```

```
% ./a.out
23095689
```


When a function returns, its local variables become out of scope and no longer active – the lifetime of local variables is from the instant the function is called to when it returns. Thus a pointer to a local variable refers to an undefined value if the variable is of a function invocation that is no longer active.

Lifetime (and still more ...)

```
int main() {
   int *a;
   a = func(1);
   printf("%d\n", *a); // what's printed?
   return 0;
}

int *func(int x) {
   return &x;
}
```

```
% ./a.out
98378932
```

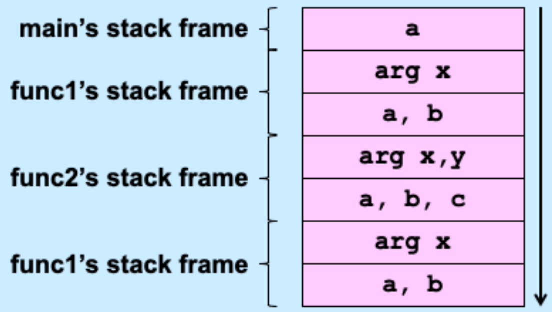

Similarly, a the lifetime of function arguments is the same as the lifetime of the function.

Rules

- Global variables exist for the duration of program's lifetime
- Local variables and arguments exist for the duration of the execution of the function
  - from call to return
  - each execution of a function results in a new instance of its arguments and local variables

Implementation: Stacks

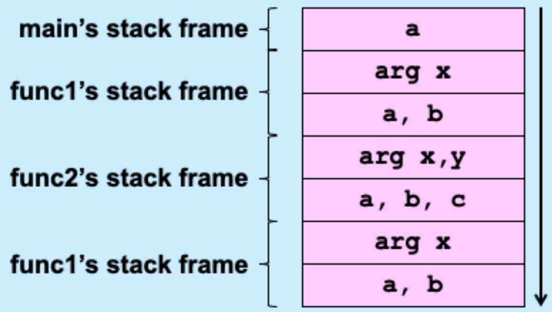
```
int main() {
   int a;
   func1(0);
   ...
}
int func1(int x) {
   int a,b;
   if (x==0) func2(a,2);
   ...
}
int func2(int x, int y) {
   int a,b,c;
   func1(1);
   ...
}
```



Function calling in C (and in most other languages) is implemented on stacks. Associated with an invocation of a function is a stack frame, which contains, among other things, its arguments and local variables. When a function is called, a stack frame for it is pushed onto the stack. When it returns, its stack frame is popped off the stack.

Implementation: Stacks

```
int main() {
   int a;
   func1(0);
   ...
}
int func1(int x) {
   int a,b;
   if (x==0) func2(a,2);
   ...
}
int func2(int x, int y) {
   int a,b,c;
   func1(1);
   ...
}
```



Quiz 3

```
void proc(int a) {
   int b=1;
   if (a == 1) {
      proc(2);
      printf("%d\n", b);
   } else {
      b = a*(b++)*b;
   }
}
int main() {
   proc(1);
   return 0;
}
```

- **What's printed?**
  - **a) 0**
  - **b) 1**
  - **c) 2**
  - **d) 4**

Static Local Variables

```
int *sub1() {
  int var = 1;
  …
  return &var;
  /* amazingly illegal */
}
```

```
int *sub2() {
  static int var = 1;
  …
  return &var;
  /* (amazingly) legal */
}
```

- Scope
  - like local variables
- Lifetime
  - like global variables

Static local variables have the same scope as other local variables, but their values are retained across calls to the procedures they are declared in. Like global variables, uninitialized static local variables are implicitly initialized to zero.

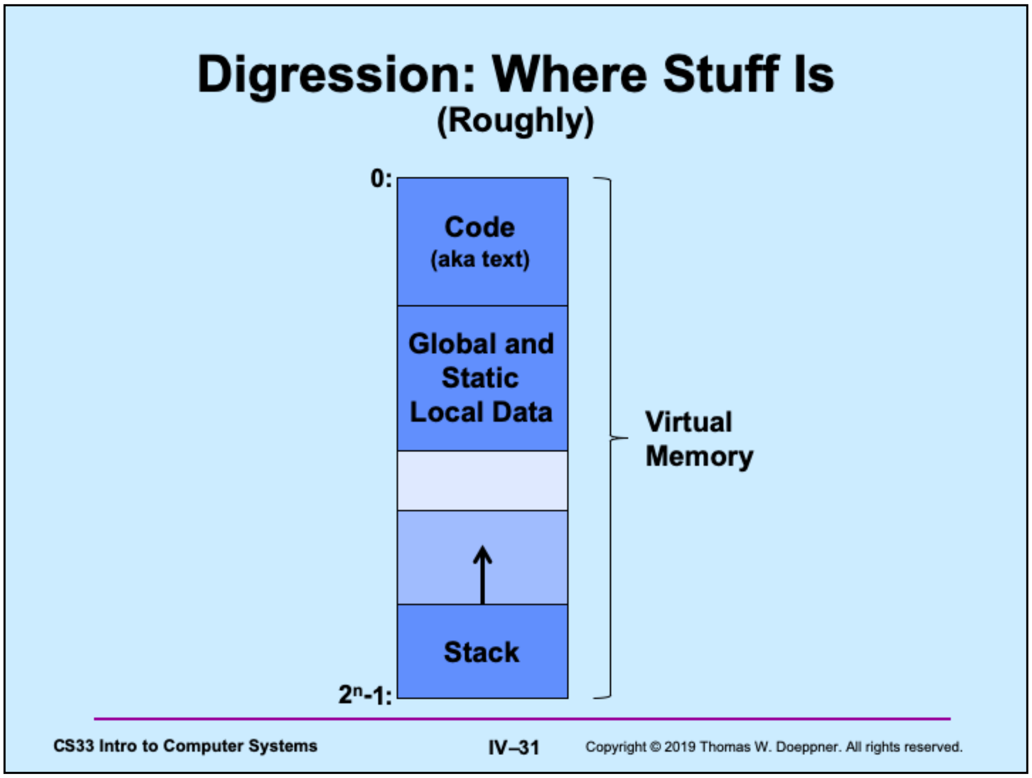
Quiz 4

```
int sub() {
  static int svar = 1;
  int lvar = 1;
  svar += lvar;
  lvar++;
  return svar;
}

int main() {
  sub();
  printf("%d\n", sub());
  return 0;
}
```

What is printed?

a) 2
b) 3
c) 4
d) 5



Let's step back and revisit our concept of virtual memory. All of a program, both code and data, resides in virtual memory. We begin to explore how all of this is organized. This is neither a complete nor a totally accurate picture, but serves to explain what we've seen so far. Executable code (also known, historically, as text) resides at the lower-addressed regions of virtual memory. After it comes a region of memory that contains global and static local data. At the high-addressed end of the address space is memory reserved for the stack. The stack itself starts at the high end of this region and grows (in response to function calls, etc.). If the end of the stack reaches the end of the region of memory reserved for it, a segmentation fault occurs and the program terminates.

This is clearly very rough. As we learn more about how computer systems work, we'll fill in more and more of the details.

scanf: Reading Data

```
int main() {
   int i, j;
   scanf("%d %d", &i, &j);
   printf("%d, %d", i, j);
}
```

```
$ ./a.out
   3        12
3, 12
```

Two parts

- **formatting instructions**
  - **whitespace in format string matches any amount of white space in input**
    - » **whitespace is space, tab, newline ('\n')**
- **arguments: must be addresses**
  - **why?**

The function *scanf* is called to read input, doing essentially the reverse of what *printf* does. Its first argument is a format string, like that of *printf*. Its subsequent arguments are pointers to locations where the input should be copied (after format conversion as specified in the format string). Note that we must have pointers for these arguments, not simple values, since arguments are passed by value. (Make sure you understand why this is important!)

The format conversion done is the reverse of what *printf* does. For example, *printf*, given the *%d* format code, converts the machine representation of an integer into its string representation in decimal notation. *scanf* with the same format code takes the string representation of a number in decimal notation and converts it to the machine representation of an integer.

#define (again)

```
#define CtoF(cel) (9.0*cel)/5.0 + 32.0
```

Simple textual substitution:

```
float tempc = 20.0;
float tempf = CtoF(tempc);
// same as tempf = (9.0*tempc)/5.0 + 32.0;
```


Careful ...

```c
#define CtoF(cel) (9.0*cel)/5.0 + 32.0
```

```c
float tempc = 20.0;
float tempf = CtoF(tempc+10);
// same as tempf = (9.0*tempc+10)/5.0 + 32.0;
```

```c
#define CtoF(cel) (9.0*(cel))/5.0 + 32.0
```

```c
float tempc = 20.0;
float tempf = CtoF(tempc+10);
// same as tempf = (9.0*(tempc+10))/5.0 + 32.0;
```


Be careful with how arguments are used! Note the use of parentheses in the second version.

Structures

```
struct ComplexNumber {
     float real;
     float imag;
};

struct ComplexNumber x;
x.real = 1.4;
x.imag = 3.65e-10;
```

Pointers to Structures

```
struct ComplexNumber {
     float real;
     float imag;
};

struct ComplexNumber x, *y;
x.real = 1.4;
x.imag = 3.65e-10;
y = &x;
y->real = 2.6523;
y->imag = 1.428e20;
```


Note that when we refer to members of a structure via a pointer, we use the "->" notation rather than the "." notation.

structs and Functions

```
struct ComplexNumber ComplexAdd(
          struct ComplexNumber a1,
          struct ComplexNumber a2) {
     struct ComplexNumber result;
     result.real = a1.real + a2.real;
     result.imag = a1.imag + a2.imag;
     return result;
}
```

Would This Work?

```
struct ComplexNumber *ComplexAdd(
            struct ComplexNumber *a1,
            struct ComplexNumber *a2) {
      struct ComplexNumber result;
      result.real = a1->real + a2->real;
      result.imag = a1->imag + a2->imag;
      return &result;
}
```


This doesn't work, since it returns a pointer to result that would not be in scope once the procedure has returned. Thus the returned pointer would point to an area of memory with undefined contents.

How About This?

```
void ComplexAdd(
        struct ComplexNumber *a1,
        struct ComplexNumber *a2,
        struct ComplexNumber *result) {
  result->real = a1->real + a2->real;
  result->imag = a1->imag + a2->imag;
  return;
}
```


This works fine: the caller provides the location to hold the result.

Using It …

```
struct ComplexNumber j1 = {3.6, 2.125};
struct ComplexNumber j2 = {4.32, 3.1416};
struct ComplexNumber sum;

ComplexAdd(&j1, &j2, &sum);
```

Arrays of *structs*

```
struct ComplexNumber j[10];
j[0].real = 8.127649;
j[0].imag = 1.76e18;
```

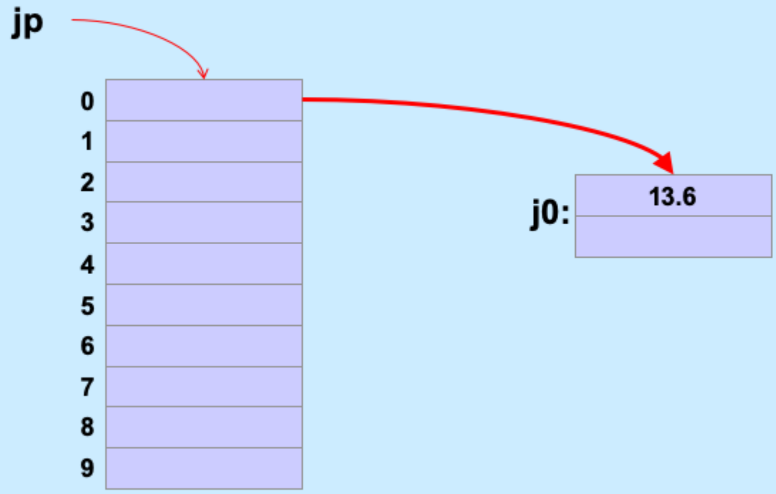
Arrays, Pointers, and *structs*

```
/* What's this? */
struct ComplexNumber *jp[10];
```

```
struct ComplexNumber j0;
jp[0] = &j0;
jp[0]->real = 13.6;
```


Subscripting (i.e., the "[]" operator) has a higher precedence than the "*" operator. Thus jp is an array of pointers to *struct ComplexNumbers*.

Memory View



Quiz 5

```
struct list_elem {
   int val;
   struct list_elem *next;
} a, b;

int main() {
   a->val = 1;
   a->next = &b;
   b->val = 2;
   printf("%d\n", a->next->val);
   return 0;
}
```

- **What happens?**
  a) syntax error
  b) seg fault
  c) prints something and terminates

Quiz 6

```
struct list_elem {
   int val;
   struct list_elem *next;
} a, b;

int main() {
   a.val = 1;
   a.next = &b;
   b.val = 2;
   printf("%d\n", a.next.val);
   return 0;
}
```

- What happens?
  a) syntax error
  b) seg fault
  c) prints something and terminates

Quiz 7

```
struct list_elem {
   int val;
   struct list_elem *next;
} a, b;

int main() {
   a.val = 1;
   b.val = 2;
   printf("%d\n", a.next->val);
   return 0;
}
```

- **What happens?**
  - a) syntax error
  - b) seg fault
  - c) prints something and terminates

Quiz 8

```
struct list_elem {
   int val;
   struct list_elem *next;
} a, b;

int main() {
   a.val = 1;
   a.next = &b;
   b.val = 2;
   printf("%d\n", a.next->val);
   return 0;
}
```

- **What happens?**
  a) syntax error
  b) seg fault
  c) prints something and terminates

