CSO-Recitation 05

CSCI-UA 0201-007

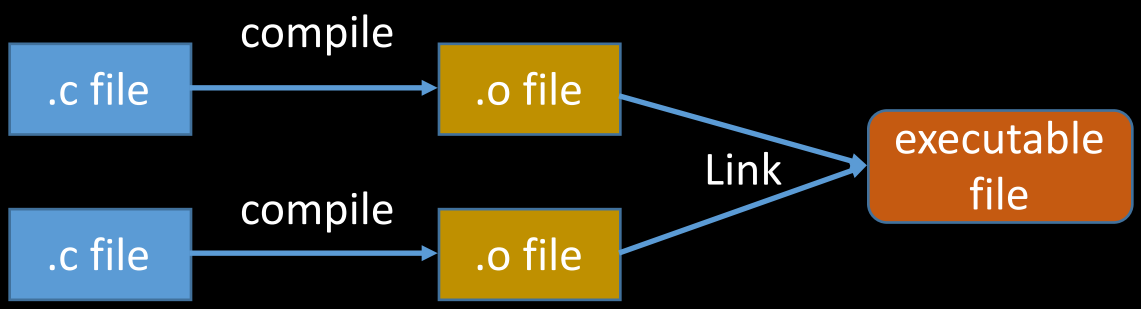
R05: Assessment 03 & Pointers & Arrays

Today’s Topics

- Assessment 03
- Pointers
- Arrays

Assessment 03

Q1 Make



compile

compile

.c file

.c file

.o file

.o file

Link

executable file

What does this make rule do?

```
prog: main.o util.o
	gcc main.o util.o -o prog
```

A. It compiles object files main.o and util.o and generates the object file prog

[✓ B.] It links object files main.o and util.o and generates the executable file prog

C. It compiles and links object files main.o and util.o and generates the executable file prog

D. It links object files main.c and util.c and generates the object file prog

Q2 C program organization

Which of the following statements are **true** about C program?

A. A header file (*.h) includes the implementation of functions to be used in other source files.

[✓ B.] A header file (*.h) includes the signature (aka declaration) of functions to be used in other source files.

C. Every source file (*.c) must contain a main function.

D. Each C binary executable file is compiled from exactly one file.

E. One can execute an object file, e.g. test.o by typing ./test.o

Q3 Floating point (smallest #)

What's the bit-pattern of the smallest positive single precision float point number?

A. 0x70000001

B. 0x80000001

[✓ C. 0x00000001]

D. 0x0007ffff

E. 0x7f800000

F. 0x7f7fffff

- Smallest positive FP:
  - denormalized encoding
  - 0000 0000 0xxx xx… xx

Q4 Floating point (largest #)

What's the bit-pattern of the largest positive single precision floating point number? (∞ does not count)

A. 0x70000001

B. 0x80000001

C. 0x00000001

D. 0x0007ffff

E. 0x7f800000

[✓ F. 0x7f7fffff]

- Largest positive FP:
  - normalized encoding
  - 0xxx xxxx xxxx xx… xx
- A: 0111 0000 000…01
- E: 0111 1111 1000 0..0 -> special value
- F: 0111 1111 0111 11..1

Q5 Floating point (precision)

What the highest and lowest precision for single precision floating points?

A. 2^{-149} and 2^{105}

B. 2^{-150} and 2^{104}

[✓ C. 2^{-149} and 2^{104}]

D. 2^{-150} and 2^{105}

E. 2^{-23} and 2^{23}

F. 2^{-126} and 2^{127}

G. 2^{-127} and 2^{127}

- highest precision:
  - smallest positive – 0
  - smallest positive:
    - 0000 0000 0000 0… 01
    - $E=1-127=-126$, $M=(0.F)_2 = 2^{-23}$
    - $FP=2^{-23}*2^{-126} = 2^{-149}$
  - 2^{-149}

- lowest precision:
  - largest positive – second largest positive
  - largest positive:
    - 0111 1111 0111 11..11
    - $E=exp-127=2^8-2-127 = 127$
    - $M=(1.F)_2=2^0+2^{-1}+2^{-2}+\ldots+2^{-23}=2-2^{-23}$
    - $FP=(2-2^{-23})*2^{127}$
  - second largest positive:
    - 0111 1111 0111 11..10
    - $E=exp-127=2^8-2-127 = 127$
    - $M=2^0+2^{-1}+2^{-2}+\ldots+2^{-22}=2-2^{-22}$
    - $FP=(2-2^{-22})*2^{127}$
- $(2-2^{-23})*2^{127} - (2-2^{-22})*2^{127} = (2^{-22}-2^{-23})*2^{127}$
- $=2^{-23}*2^{127}=2^{104}$

Q6 Left-shift

Which value is the closest to 1<<20

A. 1000

[✓ B. 1 million]

C. 1 billion

D. 2000

E. 2 million

F. 2 billion

- 1<<20
- $0..01\underbrace{00..000}_{}$
- $2^{20} = (2^{10})^2 = 1024^2 \approx (10^3)^2 = 10^6$

Q7 Bit-wise ops

Variable x is of type unsigned int. Which of the following statements returns the most significant byte of x?

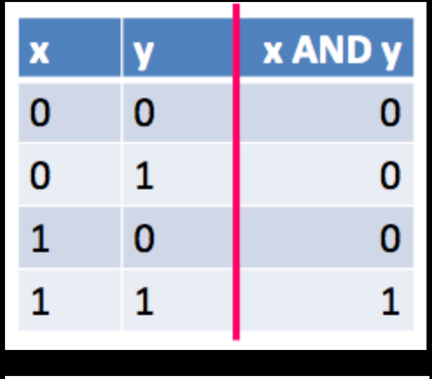
A. (char)x — least significant byte

[✓ B. (char)(x >> 24)]

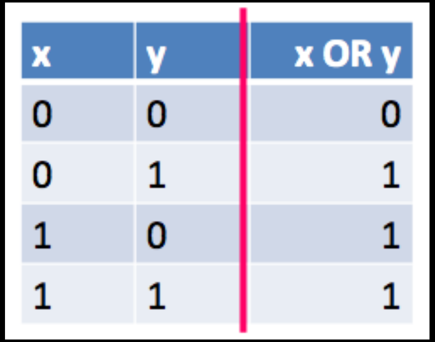
C. (char)(x | 0xff000000) — least significant byte

D. (char)(x & 0xff000000) — 0x00

E. None of the above



x	y	x AND y
0	0	0
0	1	0
1	0	0
1	1	1



x	y	x OR y
0	0	0
0	1	1
1	0	1
1	1	1

```
  0xXXXXXXXX
| 0xFF000000
------------
  0xFFXXXXXX
```

```
  0xXXXXXXXX
& 0xFF000000
------------
  0xXX000000
```

Q8 Floating point (find exp)

Given a 4-byte bit-pattern 0x72deadbe representing a single-precision floating point number, what's its corresponding 8-bit exponent field?

- 0xe5
- 11100101

- 0x72deadbe
- 0111 0010 1101 ….
- 1110 0101
- 0xe5

Q9 Floating point (clear exp)

& is often used to mask off bits: b&0 = 0
| can be used to turn some bits on: b|1=1

Suppose fi is an unsigned int whose bit pattern represents a single-precision floating point number, which of the following statements clears the exponent field of corresponding floating point number?

A. fi = fi & 0x100fffff

[✓ B. fi = fi & 0x807fffff]

C. fi = fi & 0x80ffffff

D. fi = fi & (0xff<<23) fi & 0111 1111 1000 00.. 00

E. fi = fi | (0xff<<23) fi | 0111 1111 1000 00.. 00

[✓ F. fi = fi & (~(0xff<<23))] fi & 1000 0000 0111 11.. 11

G. fi = fi & (~(1<<23)) fi & 1111 1111 0111 11.. 11

- clear the exponent field
- fi & mask
- mask = 1000 0000 0111 1..1
  - mask = 0x807fffff
  - mask = ~(0xff<<23)
    = ~0x7f800000
    = 0x807fffff

Q10 GDB

Inside GDB, a program may stop because of

A. a signal, e.g. one sent by users typing "Ctrl-C".

B. a breakpoint

C. step command

[✓ D. all of the mentioned]

Pointers

A variable that stores a memory address

What are pointers?

- They are variables that store addresses
  - Pointers can have different types, depending on what they point to
    - But they remain the same size – for us on a 64-bit system, 8 bytes (64 bits)

Type	Value	Address
int	an integer number	memory address
float	a floating point number	memory address
char	a character/byte	memory address
pointer	memory address	memory address

- If I want the value of a variable var -> var
- If I want the address of a variable var -> &var
- If var is a pointer, then I can get the value of the variable that var points to -> *var

What are pointers?

- They are variables that store addresses
  - Pointers can have different types, depending on what they point to
    - But they remain the same size – for us on a 64-bit system, 8 bytes (64 bits)
- Two primary operations
  - & - called “reference”
    - Gets the address of a variable / array element
    - You perform this to get the value for a pointer
  - * - called “de-reference”
    - Gets the value located at a memory address
    - You perform this on the pointer

How do you use pointers?

- Say you have a variable var
  - int var = 10;
- You can make a pointer called ptr using this code
  - int *ptr;
- ptr can be set to point to var with the reference operator
  - ptr = &var;
- The value of ptr is now the address of var, not its value
  - To get the value, de-reference:
    - *ptr //this equals to 10
    - *ptr = 5; // this sets var to 5

Pointer types

- Why do we need pointer types?
    - Without it, making mistakes like de-referencing a number by accident would be common
    - Without it, pointer arithmetic wouldn’t work
- What is pointer arithmetic?
    - If you have a pointer called ptr, the value of ptr+1 is based off the type of ptr
        - If ptr is a char*, then ptr+1 is the memory address of next char after ptr
        - If ptr is an int*, then ptr+1 is the memory address of next int after ptr
    - ptr+n means “start at ptr, and go forward as many bytes as *n* copies of what ptr points to take up”

Function arguments and pointers

- In C, arguments are passed by value
  - Means that when you call a function, the arguments are copied from the caller to the function's stack frame
  - This means that if a function modifies one of its arguments, it is not modified for whoever called the function
- If you want to pass a reference, you must use pointers
  - Then the function can modify the variable by dereferencing the pointer

Arrays

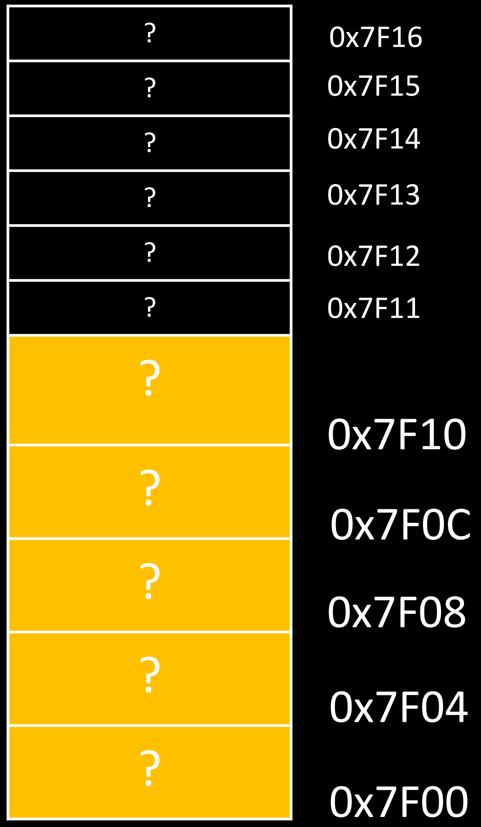
Contiguous, homogenous data

What are arrays?

- Basically, they are chunks of memory that hold a number of elements of the same data type
- This memory is contiguous, that is, the elements are all touching
- You can define an int array like this
  - int my_array[5];
  - This will make an array of 5 ints (20 bytes)
  - You can initialize the array as follows:
    - int my_array[5] = {1, 2, 3, 4, 5};
    - You can also set it to all zeroes using int my_array[5]={0};
- You can index with the [] operator
  - my_array[0] gets the first element of my_array
  - my_array[0] = 5 sets the first elelment of my_array to 5

Defining an array

- int arr[5];
- The value of an array is the address of its first element
  - The value of arr is 0x7F00
    - arr==&arr[0]
- Let a pointer points to the 1st element of this array
  - int *p = arr;
    - int *p = &arr[0];
- Array and pointer can be syntactically equivalent
  - *p == p[0], here also *p==arr[0]
  - *arr (==arr[0]) / *(arr+2) ==arr[2]



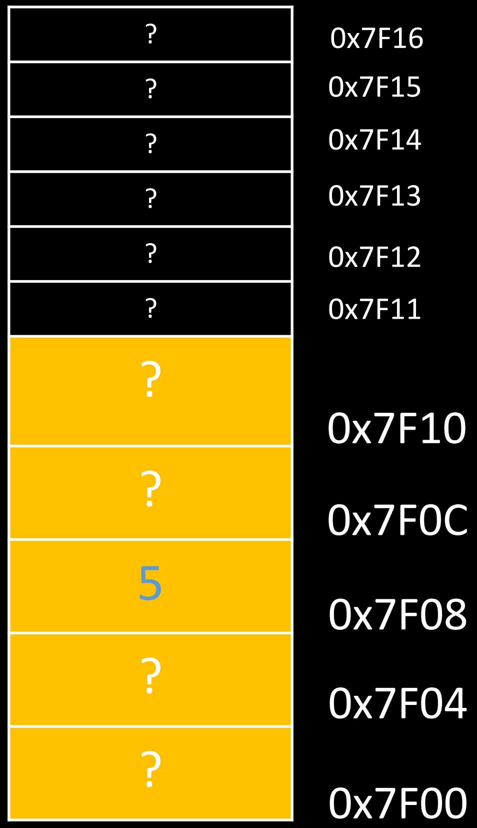
?	0x7F16
?	0x7F15
?	0x7F14
?	0x7F13
?	0x7F12
?	0x7F11
?	0x7F10
?	0x7F0C
?	0x7F08
?	0x7F04
?	0x7F00

Pointer and array

- One difference between an array name and a pointer
  - A pointer is a variable
    - p = arr; / p++; are legal
  - But an array name is not a variable..
    - cannot write things like arr++; / arr=p; (illegal)
- When an array name is passed to a function,
  - What it passed is the location of the initial element
  - So within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address

Indexing an array

- int arr[5];
- Arrays can be index like so
  - arr[2] = 5;
  - This will set the third element of arr to 5
  - This is the same as *(arr + 2) = 5;
    - Which is to say, this is done by taking the value of arr, 0x7F00, and adding 2 to it according to pointer arithmetic
    - The size of int is 4, so we are going 8 bytes passed arr, 8 + 0x7F00 = 0x7F08



?	0x7F16
?	0x7F15
?	0x7F14
?	0x7F13
?	0x7F12
?	0x7F11
?	0x7F10
?	0x7F0C
5	0x7F08
?	0x7F04
?	0x7F00

Arrays and functions

- Array names act as pointers to the array’s first element
- To use a function with an array, we use pointers
    - But we need to also pass the number of elements in this array to function

Pointers to pointers (Pointer arrays)

- Since pointers are variable themselves, they can be stored in arrays just as other variables can
  - char *a[2];
- Let a pointer points to the 1st element of this array (of pointers)
  - char **p = &a[0]; / char **p=a;
- An array of pointers
- Think about what can this do?