COMP201 Fall 2020 Final Exam
Duration: **120 minutes**

Student ID: |___|___|___|___|___|___|___|___|___|

Lab Section: |___|___|

First Name(s): _____ **Last Name:** _____

---

*Do **not** turn this page until you are told to do so.*
In the meantime, please read the instructions below.

Good Luck!

This exam contains **8 multi-part questions** and you have **120 minutes** to earn 100 marks.

- When the exam begins, please write your student ID, name and lecture section on top of this page, and sign the academic integrity code given below.
- Check that the exam booklet contains a total of **19 pages**, including this one.
- This exam is an **open book and notes exam**.
- Show all work, as partial credit will be given since you will be graded not only on the correctness and efficiency of your answers, but also on your clarity that you express it. Be neat.

# 1:	_____/ 12
# 2:	_____/ 14
# 3:	_____/ 12
# 4:	_____/ 14
# 5:	_____/ 15
# 6:	_____/ 13
# 7:	_____/ 10
# 8:	_____/ 10
TOTAL:	_____/ 100

I hereby declare that I have completed this exam individually, without support from anyone else.

I hereby accept that only the below listed sources are approved to be used during this open-source exam:
(i) Coursebook,
(ii) All material that is made available to students via Blackboard for this course, and
(iii) Notes taken by me during lectures.

I have not used, accessed or taken any unpermitted information from any other source. Hence, all effort belongs to me.

Signature: _____

Question 1A. Calling Functions in Assembly [12 POINTS]

In the following, you are provided some assembly generated by compiling a C program using gcc compiler.

```
0000000000400540 <mystery>:
  400540:       41 54                   push   %r12
  400542:       31 c0                   xor    %eax,%eax
  400544:       49 89 fc                mov    %rdi,%r12
  400547:       55                      push   %rbp
  400548:       89 d5                   mov    %edx,%ebp
  40054a:       53                      push   %rbx
  40054b:       89 f3                   mov    %esi,%ebx
  40054d:       03 5f 04                add    0x4(%rdi),%ebx
  400550:       bf 10 06 40 00          mov    $0x400610,%edi
  400555:       89 de                   mov    %ebx,%esi
  400557:       e8 b4 fe ff ff          callq  400410 <printf@plt>
  40055c:       41 03 6c 24 08          add    0x8(%r12),%ebp
  400561:       bf 10 06 40 00          mov    $0x400610,%edi
  400566:       31 c0                   xor    %eax,%eax
  400568:       89 ee                   mov    %ebp,%esi
  40056a:       e8 a1 fe ff ff          callq  400410 <printf@plt>
  40056f:       8d 04 2b                lea    (%rbx,%rbp,1),%eax
  400572:       5b                      pop    %rbx
  400573:       5d                      pop    %rbp
  400574:       41 5c                   pop    %r12
  400576:       c3                      retq
```

(a) [3 POINTS] How many parameters does this function take?

3. It uses `%rdi, %esi, %edx`

(b) [3 POINTS] Does this function return any value?

Yes. It uses %eax.

(c) [3 POINTS] What is the purpose of the `push` instruction at the first line?

A: allocate space for local variables
B: save parameters
C: save caller-saved registers
D: save callee-saved registers

(d) [3 POINTS] What is the type of the variable the register `%rdi` stores?

Array or Pointer

Question 1B. Calling Functions in Assembly [12 POINTS]

In the following, you are provided some assembly generated by compiling a C program using gcc compiler.

```
0000000000400540 <mystery>:
  400540:       41 54                   push   %r12
  400542:       31 c0                   xor    %eax,%eax
  400544:       49 89 fc                mov    %rdi,%r12
  400547:       55                      push   %rbp
  400548:       89 f5                   mov    %esi,%ebp
  40054a:       53                      push   %rbx
  40054b:       8b 5f 04                mov    0x4(%rdi),%ebx
  40054e:       bf 10 06 40 00          mov    $0x400610,%edi
  400553:       01 f3                   add    %esi,%ebx
  400555:       89 de                   mov    %ebx,%esi
  400557:       e8 b4 fe ff ff          callq  400410 <printf@plt>
  40055c:       41 03 6c 24 08          add    0x8(%r12),%ebp
  400561:       bf 10 06 40 00          mov    $0x400610,%edi
  400566:       31 c0                   xor    %eax,%eax
  400568:       0f af eb                imul   %ebx,%ebp
  40056b:       89 ee                   mov    %ebp,%esi
  40056d:       e8 9e fe ff ff          callq  400410 <printf@plt>
  400572:       8d 04 2b                lea    (%rbx,%rbp,1),%eax
  400575:       5b                      pop    %rbx
  400576:       5d                      pop    %rbp
  400577:       41 5c                   pop    %r12
  400579:       c3                      retq
```

(a) [3 POINTS] How many parameters does this function take?

2. It uses `%rdi, %esi`

(b) [3 POINTS] Does this function return any value?

Yes. It uses `%eax`.

(c) [3 POINTS] What is the purpose of the `push` instruction at the first line?

A: allocate space for local variables
B: save parameters
C: save caller-saved registers
D: save callee-saved registers

(d) [3 POINTS] What is the type of the variable the register `%rdi` stores?

Array or Pointer

Question 2. x86-64 Procedures and Code Optimization [14 POINTS]

In the following, you are given some code fragments written in C and the corresponding assembly codes generated by the GCC compiler (some parts of the assembly code may be intentionally omitted). In these questions, each code fragment contains a variable, constant, or a function named `mando` (short for Mandalorian). As an officer of the New Republic, your task is to locate `mando` at a certain point marked with `***`. In particular, you are asked to provide either an assembly expression or constant (like a memory address) that holds the value of `mando`. Below is an example question and the corresponding answer to make what is expected more clear.

Where is Mando?

```
int identity(int mando) {
    return mando;
}
```

```
00000000004004ed <identity>:
  4004ed:       55                      push   %rbp
  4004ee:       48 89 e5                mov    %rsp,%rbp
  4004f1:       89 7d fc                mov    %edi,-0x4(%rbp)
  4004f4:       8b 45 fc                mov    -0x4(%rbp),%eax
           ***
  4004f7:       5d                      pop    %rbp
  4004f8:       c3                      retq
```

Answer: Any of the expressions `%edi`, `-0x4(%rbp)`, `%eax`, and `%rax` are valid answers. They all hold the value of `mando` at the marked point. Note that, however, if the asterisks came before the *first* instruction (before `4004ed`), `%edi` would be the only correct answer.

(a) [2 POINTS] Where is Mando?

```
int myfunc1(int a, int b, int mando) {
    if (a > b)
        return mando;
    else
        return 4 * mando;
}
```

```
0000000000400505 <myfunc1>:
  400505:       55                      push   %rbp
  400506:       48 89 e5                mov    %rsp,%rbp
  400509:       89 7d fc                mov    %edi,-0x4(%rbp)
  40050c:       89 75 f8                mov    %esi,-0x8(%rbp)
  40050f:       89 55 f4                mov    %edx,-0xc(%rbp)
           ***
  400512:       8b 45 fc                mov    -0x4(%rbp),%eax
  400515:       3b 45 f8                cmp    -0x8(%rbp),%eax
  400518:       7e 05                   jle    40051f <myfunc1+0x1a>
  40051a:       8b 45 f4                mov    -0xc(%rbp),%eax
  40051d:       eb 06                   jmp    400525 <myfunc1+0x20>
  40051f:       8b 45 f4                mov    -0xc(%rbp),%eax
  400522:       c1 e0 02                shl    $0x2,%eax
  400525:       5d                      pop    %rbp
  400526:       c3                      retq
```

Answer: `%edx or -0xc(%rbp)`

(b) [2 POINTS] Where is Mando?

```
int int_array_sum(int* a, int len) {
    int mando = 0;
    for (int i = 0; i < len; i++)
        mando += a[i];
    return mando;
}
```

```
0000000000400527 <int_array_sum>:
  400527:       55                      push   %rbp
  400528:       48 89 e5                mov    %rsp,%rbp
  40052b:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  40052f:       89 75 e4                mov    %esi,-0x1c(%rbp)
  400532:       c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)
  400539:       c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%rbp)
  400540:       eb 1d                   jmp    40055f <int_array_sum+0x38>
  400542:       8b 45 f8                mov    -0x8(%rbp),%eax
  400545:       48 98                   cltq
  400547:       48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  40054e:       00
  40054f:       48 8b 45 e8             mov    -0x18(%rbp),%rax
          ***
  400553:       48 01 d0                add    %rdx,%rax
  400556:       8b 00                   mov    (%rax),%eax
  400558:       01 45 fc                add    %eax,-0x4(%rbp)
  40055b:       83 45 f8 01             addl   $0x1,-0x8(%rbp)
  40055f:       8b 45 f8                mov    -0x8(%rbp),%eax
  400562:       3b 45 e4                cmp    -0x1c(%rbp),%eax
  400565:       7c db                   jl     400542 <int_array_sum+0x1b>
  400567:       8b 45 fc                mov    -0x4(%rbp),%eax
  40056a:       5d                      pop    %rbp
  40056b:       c3                      retq
```

Answer: -0x4(%rbp)

(c) [2 POINTS] Where is Mando?

```
int int_array_shift(int* a, int b, int mando, int len) {
    for (int i = 0; i < len; i++)
        a[i] += b * (mando + 2);
    return mando;
}
```

```
000000000040056c <int_array_shift>:
INSTRUCTIONS OMITTED
  4005b4:       83 c2 02                add    $0x2,%edx
  4005b7:       0f af 55 e4             imul   -0x1c(%rbp),%edx
  4005bb:       01 ca                   add    %ecx,%edx
  4005bd:       89 10                   mov    %edx,(%rax)
          ***
  4005bf:       83 45 fc 01             addl   $0x1,-0x4(%rbp)
  4005c3:       8b 45 fc                mov    -0x4(%rbp),%eax
  4005c6:       3b 45 dc                cmp    -0x24(%rbp),%eax
  4005c9:       7c bb                   jl     400586 <int_array_shift+0x1a>
```

```
  4005cb:       8b 45 e0                mov    -0x20(%rbp),%eax
  4005ce:       5d                      pop    %rbp
  4005cf:       c3                      retq
```

Answer: -0x20(%rbp)

(d) [2 POINTS] Where is Mando?

```
int array2d_get(int** arr2d, int row, int col) {
    int* mando = arr2d[row];
    return mando[col];
}
```

```
00000000004005d0 <array2d_get>:
  4005d0:       55                      push   %rbp
  4005d1:       48 89 e5                mov    %rsp,%rbp
  4005d4:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
  4005d8:       89 75 e4                mov    %esi,-0x1c(%rbp)
  4005db:       89 55 e0                mov    %edx,-0x20(%rbp)
  4005de:       8b 45 e4                mov    -0x1c(%rbp),%eax
  4005e1:       48 98                   cltq
  4005e3:       48 8d 14 c5 00 00 00    lea    0x0(,%rax,8),%rdx
  4005ea:       00
  4005eb:       48 8b 45 e8             mov    -0x18(%rbp),%rax
  4005ef:       48 01 d0                add    %rdx,%rax
  4005f2:       48 8b 00                mov    (%rax),%rax
  4005f5:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  4005f9:       8b 45 e0                mov    -0x20(%rbp),%eax
          ***
  4005fc:       48 98                   cltq
  4005fe:       48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx
  400605:       00
  400606:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  40060a:       48 01 d0                add    %rdx,%rax
  40060d:       8b 00                   mov    (%rax),%eax
  40060f:       5d                      pop    %rbp
  400610:       c3                      retq
```

Answer: %rax, %eax or -0x8(%rbp)

(e) [2 POINTS] Where is Mando?

```
int myfunc2(int x) {
    if (x > 4) {
        return mando(63 * x);
    } else {
        return mando(124 * x);
    }
}
```

```
0000000000400611 <myfunc2>:
  400611:       55                      push   %rbp
  400612:       48 89 e5                mov    %rsp,%rbp
  400615:       48 83 ec 08             sub    $0x8,%rsp
  400619:       89 7d fc                mov    %edi,-0x4(%rbp)
  40061c:       83 7d fc 04             cmpl   $0x4,-0x4(%rbp)
  400620:       7e 13                   jle    400635 <myfunc2+0x24>
  400622:       8b 55 fc                mov    -0x4(%rbp),%edx
  400625:       89 d0                   mov    %edx,%eax
          ***
  400627:       c1 e0 06                shl    $0x6,%eax
  40062a:       29 d0                   sub    %edx,%eax
  40062c:       89 c7                   mov    %eax,%edi
  40062e:       e8 c6 fe ff ff          callq  4004f9
  400633:       eb 16                   jmp    40064b <myfunc2+0x3a>
  400635:       8b 45 fc                mov    -0x4(%rbp),%eax
  400638:       c1 e0 02                shl    $0x2,%eax
  40063b:       89 c2                   mov    %eax,%edx
  40063d:       c1 e2 05                shl    $0x5,%edx
  400640:       29 c2                   sub    %eax,%edx
  400642:       89 d0                   mov    %edx,%eax
  400644:       89 c7                   mov    %eax,%edi
  400646:       e8 ae fe ff ff          callq  4004f9
  40064b:       c9                      leaveq
  40064c:       c3                      retq
```

Answer: 4004f9

In the remaining, you are given an assembly code which is compiled from one of the above functions using a different compiler or a different optimization level. Your task is to determine what the given assembly code corresponds to.

(f) [2 POINTS] What's `mando`? Circle one.

```
0000000000400540 <mando>:
  400540:       85 c9                   test   %ecx,%ecx
  400542:       89 d0                   mov    %edx,%eax
  400544:       7e 1d                   jle    400563 <mando+0x23>
  400546:       8d 52 02                lea    0x2(%rdx),%edx
  400549:       0f af f2                imul   %edx,%esi
  40054c:       8d 51 ff                lea    -0x1(%rcx),%edx
  40054f:       48 8d 4c 97 04          lea    0x4(%rdi,%rdx,4),%rcx
  400554:       0f 1f 40 00             nopl   0x0(%rax)
  400558:       01 37                   add    %esi,(%rdi)
  40055a:       48 83 c7 04             add    $0x4,%rdi
  40055e:       48 39 cf                cmp    %rcx,%rdi
  400561:       75 f5                   jne    400558 <mando+0x18>
  400563:       f3 c3                   repz retq
```

myfunc1 int_array_sum int_array_shift array2d_get myfunc2

(g) [2 POINTS] What's `mando`? Circle one.

```
0000000000400580 <mando>:
  400580:       83 ff 04                cmp    $0x4,%edi
  400583:       7f 0b                   jg     400590 <mando+0x10>
  400585:       89 f8                   mov    %edi,%eax
  400587:       ba 7c 00 00 00          mov    $0x7c,%edx
  40058c:       0f af c2                imul   %edx,%eax
  40058f:       c3                      retq
  400590:       89 f8                   mov    %edi,%eax
  400592:       c1 e0 06                shl    $0x6,%eax
  400595:       29 f8                   sub    %edi,%eax
  400597:       c3                      retq
```

`myfunc1` `int_array_sum` `int_array_shift` `array2d_get` `myfunc2`

Question 3A. Data and Stack Frames [12 POINTS]

(a) [4 POINTS] Consider the following C program, in which `H` and `J` are constants expressed with `#define` directives, and the corresponding assembly code generated by the gcc compiler. By inspecting the assembly code, try to find the values of `H` and `J`.

```
int arr1[H][J];
int arr2[J][H];

void update_array(int x, int y) {
    arr2[y][x] = 3*arr1[x][y];
}
```

```
00000000004004f0 <update_array>:
  4004f0:       48 63 f6                movslq %esi,%rsi
  4004f3:       48 63 ff                movslq %edi,%rdi
  4004f6:       48 8d 04 76             lea    (%rsi,%rsi,2),%rax
  4004fa:       48 8d 14 87             lea    (%rdi,%rax,4),%rdx
  4004fe:       48 8d 04 7f             lea    (%rdi,%rdi,2),%rax
  400502:       48 8d 04 46             lea    (%rsi,%rax,2),%rax
  400506:       8b 04 85 60 10 60 00    mov    arr1(,%rax,4),%eax
  40050d:       8d 04 40                lea    (%rax,%rax,2),%eax
  400510:       89 04 95 80 11 60 00    mov    %eax,arr2(,%rdx,4)
  400517:       c3                      retq
```

`H = 12`

`J = 6`

(b) [4 POINTS] Determine the offset of each field and the total size (in bytes) of the structure given below, considering the alignment requirements of a 64-bit machine.

Structure	d1	c1	f1	c2	s1	p1	s2	Total

`struct mystruct {`
`  int d1;`
`  char c1;`
`  float f1;`
`  char c2;`
`  short s1;`
`  int *p1;`
`  short s2;`
`}`
	0	**4**	**8**	**12**	**14**	**16**	**24**	**32**

(c) [4 POINTS] What is the size (in bytes) of the structure if the fields in part (b) are rearranged to have minimum wasted space?

24

Question 3B. Data and Stack Frames [12 POINTS]

(a) [4 POINTS] Consider the following C program, in which `H` and `J` are constants expresses with `#define` directives, and the corresponding assembly code generated by the gcc compiler. By inspecting the assembly code, try to find the values of `H` and `J`.

```
int arr1[H][J];
int arr2[J][H];

void modify_array(int x, int y) {
    arr2[y][x] = 2*arr1[x][y]-1;
}
```

```
00000000004004f0 <modify_array>:
  4004f0:       48 63 ff                movslq %edi,%rdi
  4004f3:       48 63 f6                movslq %esi,%rsi
  4004f6:       48 8d 14 3f             lea    (%rdi,%rdi,1),%rdx
  4004fa:       48 8d 04 f7             lea    (%rdi,%rsi,8),%rax
  4004fe:       48 c1 e7 04             shl    $0x4,%rdi
  400502:       48 29 d7                sub    %rdx,%rdi
  400505:       48 01 f7                add    %rsi,%rdi
  400508:       8b 14 bd 60 10 60 00    mov    arr1(,%rdi,4),%edx
  40050f:       8d 54 12 ff             lea    -0x1(%rdx,%rdx,1),%edx
  400513:       89 14 85 20 12 60 00    mov    %edx,arr2(,%rax,4)
  40051a:       c3                      retq
```

`H = 8`

`J = 14`

(b) [4 POINTS] Determine the offset of each field and the total size (in bytes) of the structure given below, considering the alignment requirements of a 64-bit machine.

Structure	c1	s1	d1	c2	f1	s2	p1	Total

`struct mystruct {`
`  char c1;`
`  short s1;`
`  int d1;`
`  char c2;`
`  float f1;`
`  short s2;`
`  int *p1;`
`}`
	0	**2**	**4**	**8**	**12**	**16**	**24**	**32**

(c) [4 POINTS] What is the size (in bytes) of the structure if the fields in part (b) are rearranged to have minimum wasted space.

24

Question 4. Buffer Overflow [14 POINTS]

Elliott Alderson decided to perform attacks against FSociety machines to check the integrity of their security measures. As one of the new members, Tyrell Wellick has written a password checker that he thinks is unbreakable! Below is the front-end to his program:

```
int main(int argc, char *argv[]) {
    char passwd[20];
    printf("Password:");
    scanf("%s",passwd);
    if (check(passwd)) {
        enterFSociety();
        exit(0);
    }
    printf("Sorry, you are not awake!");
    return 0;
}
```

(a) [3 POINTS] Elliott does not believe his eyes when he sees this. He thinks that it is so brittle. Briefly explain how Elliott can hack this program using a buffer overflow attack. Please explain with 2-3 sentences.

Elliot can enter a string long enough to overflow the buffer to overwrite the main function's return address so that it points to the address of the exploit.

Tyrell then objects that it is not that easy because he runs his program on a special system where the stack is not executable. In other words, it is impossible to execute any code on the stack, making a typical buffer overflow attack is not possible.

After a minute of silence, Elliott decides he can indeed perform a *return-to-libc* attack – utilizing an already present code in the program which can enable you execute arbitrary instructions. Here is some background on this attack type:

The `system(char *command)` function is a part of C standard library and it can be used to execute the string `command` as if it is typed in the shell of the operation system.

Using `gdb`, Elliott discovers the following:

```
(gdb) print system
$1 = {<text variable, no debug info>} 0xfc3406ac <system>
```

He knows that in every executable program, the environment variables are loaded at runtime, and guess what it involves the current `SHELL`:

```
(gdb) print (char *) 0xfc2356e2
$2 = 0xfc2356e2 "SHELL=/bin/bash"
```

Using this information, Elliot now knows that he can launch a shell from Tyrell's program, proving that he can in fact execute any arbitrary code that he wants!

(b) [3 POINTS] What are the addresses of the `system()` function and the string `"/bin/bash"`?

Address of the `system()` function: `0xfc3406ac`
Address of the string `"/bin/bash"`: `0xfc2356e8`

(c) [5 POINTS] Specify what Elliott can pass as input to Tyrell's program in order to cause it to launch a shell screen. In your answer, you don't need to be too specific about the sizes or lengths, but keep in mind where the arguments go in 32-bit mode.

```
some garbage bytes | old %ebp | addr. of system() | 4 byte garbage | addr. of "/bin/bash"
```

(d) [3 POINTS] Explain how Elliot's exploit string will enable to execute a shell on Tyrell's program.

The `main` function will rather return to the `system()` function with the argument `"/bin/bash"`, which will open a new shell to input any arbitrary command(s).

-1: If no mention of system() or "/bin/bash".

Question 5A. Locality [15 POINTS]

Consider the following C function that calculates the sum of the rows of a 2D matrix and assume that the elements of the array `sum` have been initialized to zero.

```
void sumRows(int nCols, int nRows, int sum[nRows], int A[nRows][nCols]) {
        for (int j = 0; j < nCols; j++) {
           for (int i = 0; i < nRows; i++) {
              sum[i] = sum[i] + A[i][j];
           }
        }
}
```

(a) [3 POINTS] Does the function `sumRows` exhibit *temporal locality*?

Temporal locality with respect to code: Yes. `sum[j]` computed and used many times.
Temporal locality with respect to array `A`: No (not a single element `A[i][j]` accessed more than once.)
Temporal locality with respect to array `sum`: Yes (`sum[i]` accessed frequently inside the inner loop).

(b) [3 POINTS] Does the function `sumRows` exhibit *spatial locality*?

Spatial locality with respect to code: Yes (the codes are executed in a contiguous section of the memory)
Spatial locality with respect to array `A`: No (The inner loop traverses the column elements).
Spatial locality with respect to array `sum`: Yes (Inner loop traverse `sum` with stride 1).

Now consider the following alternative implementation `sumRows2` where the loops are permuted.

```
void sumRows2(int nCols, int nRows, int sum[nRows], int A[nRows][nCols]) {
        for (int i = 0; i < nRows; i++) {
           for (int j = 0; j < nCols; j++){
              sum[i] = sum[i] + A[i][j];
           }
        }
}
```

(c) [3 POINTS] Does the function `sumRows2` exhibit *temporal locality*?

Temporal locality with respect to code: Yes. `sum[j]` computed and used many times.
Temporal locality with respect to array `A`: No (not a single element `A[i][j]` accessed more than once.)
Temporal locality with respect to array `sum`: Yes (`sum[i]` accessed frequently inside the inner loop).

(d) [3 POINTS] Does the function `sumRows2` exhibit *spatial locality*?

Spatial locality with respect to code: Yes (the codes are executed in a contiguous section of the memory)
Spatial locality with respect to array `A`: Yes (the inner loop traverses `A` via stride 1).
Spatial locality with respect to array `sum`: Yes (the inner loop traverse `sum` via stride 1).

(e) [3 POINTS] The functions `sumRows` and `sumRows2` accomplish the same task. Which one is more cache friendly?

`sumRows2` has more locality, hence more cache friendly.

Question 5B. Locality [15 POINTS]

Consider the following C function that calculates the sum of the columns of a 2D matrix and assume that the elements of the array `sum` have been initialized to zero.

```
void sumCols(int nCols, int nRows, int sum[nCols], int A[nRows][nCols]) {
        for (int j = 0; j < nCols; j++) {
           for (int i = 0; i < nRows; i++) {
              sum[j] = sum[j] + A[i][j];
           }
        }
}
```

(a) [3 POINTS] Does the function `sumCols` exhibit *temporal locality*?

Temporal locality with respect to code: Yes. `sum[j]` computed and used many times.
Temporal locality with respect to array `A`: No (not a single element `A[i][j]` accessed more than once.)
Temporal locality with respect to array `sum`: Yes (`sum[j]` accessed frequently inside the inner loop).

(b) [3 POINTS] Does the function `sumCols` exhibit *spatial locality*?

Spatial locality with respect to code: Yes (the codes are executed in a contiguous section of the memory)
Spatial locality with respect to array `A`: No (The inner loop traverses the column elements).
Spatial locality with respect to array `sum`: Yes (Inner loop traverse `sum` with stride 1).

Now consider the following alternative implementation `sumCols2` where the loops are permuted.

```
void sumCols2(int nCols, int nRows, int sum[nCols], int A[nRows][nCols]) {
        for (int i = 0; i < nRows; i++) {
           for (int j = 0; j < nCols; j++){
              sum[j] = sum[j] + A[i][j];
           }
        }
}
```

(c) [3 POINTS] Does the function `sumCols2` exhibit *temporal locality*?

Temporal locality with respect to code: Yes. `sum[j]` computed and used many times.
Temporal locality with respect to array `A`: No (not a single element `A[i][j]` accessed more than once.)
Temporal locality with respect to array `sum`: Yes (`sum[j]` accessed frequently inside the inner loop).

(d) [3 POINTS] Does the function `sumCols2` exhibit *spatial locality*?

Spatial locality with respect to code: Yes (the codes are executed in a contiguous section of the memory)
Spatial locality with respect to array `A`: Yes (the inner loop traverses `A` via stride 1).
Spatial locality with respect to array `sum`: Yes (the inner loop traverse `sum` via stride 1).

(e) [3 POINTS] The functions `sumCols` and `sumCols2` accomplish the same task. Which one is more cache friendly?

`sumCols2` has more locality, hence more cache friendly.

Question 6. Cache Memories [13 POINTS]

(a) [5 POINTS] Suppose that you are working on a system with the following specifications:

- The device is a 12-bit machine, i.e., the physical addresses are 12 bits wide.
- The memory is byte addressable, and memory accesses are to 1-byte words.
- The device contains a 4-way set associative cache (E = 4: Four lines per set)

Below table shows the current contents of the cache:

4-way Set Associate Cache (E = 4: Four lines per set)

Index	Valid	Tag	Byte 0	Byte 1	Valid	Tag	Byte 0	Byte 1	Valid	Tag	Byte 0	Byte 1	Valid	Tag	Byte 0	Byte 1
0	1	39	29	42	1	78	19	1A	0	7F	80	81	0	BC	36	34
1	0	FD	01	5F	1	D3	0F	10	1	3D	5E	5F	1	4F	A3	A6
2	1	A3	C4	54	1	BC	3A	3B	1	EC	4E	4C	1	D4	3E	4A
3	1	3A	CD	3E	0	0E	54	55	0	C3	33	12	1	57	6A	E2
4	0	22	69	65	1	D2	37	38	0	23	4C	45	1	22	7E	4C
5	0	CA	E4	39	0	CA	7E	7F	0	7B	C2	A2	1	CA	23	25
6	1	2C	1B	E4	0	3E	A9	AA	1	F9	D3	45	0	FD	FF	00
7	0	0F	40	DD	0	CD	E1	E2	0	D9	D4	F6	0	BB	63	64

For an access request to memory address `0xCAB`, fill in the following table with the corresponding values for the offset *(O)*, index *(I)*, tag *(T)*, and whether a cache hit or miss occurred, and if it is a hit, what value will be returned?

	Value
Cache Offset (O)	0x1
Cache Index (I)	0x5
Cache Tag (T)	0xCA
Cache Hit? (Y/N)	Y
Cache Byte value	0x25

As a reporter working at MacRumors, you are given access to one of Apple's new iPhone prototypes. However, there is no information about the cache memories integrated to the device, and your goal is to perform some experiments to figure out some properties of its cache memory. Considering that the cache is empty (cold), here is the report of some address traces and whether it is a MISS or HIT.

Address Trace	Hit/Miss
0x000	MISS
0x001	HIT
0x002	HIT
0x003	HIT
0x004	HIT
0x005	HIT
0x006	HIT
0x007	HIT
0x008	MISS

(b) [2 POINTS] What is the block size of the cache (in bytes)?

8 bytes. The first one is a miss, but the request loads the next 7 sequential bytes as well.

(c) [2 POINTS] The following table shows your observations from your second set of experiments, in which you incremented the addresses of the memory accesses with a certain pattern:

Address Trace	Hit/Miss
`0x000`	MISS
`0x008`	MISS
`0x000`	HIT
`0x010`	MISS
`0x000`	HIT
`0x018`	MISS
`0x000`	HIT
`0x020`	MISS
`0x000`	HIT
(some more trials…)	
`0x100`	MISS
`0x000`	MISS

What is the total size of the cache (in bytes)?

32*8 =256 bytes. Only after 32 trials, `0x000` becomes a MISS.

(d) [2 POINTS] The following table shows your observations from your third set of experiments:

Address Trace	Hit/Miss
`0x000`	MISS
`0x100`	MISS
`0x000`	HIT
`0x200`	MISS
`0x000`	HIT
`0x300`	MISS
`0x000`	HIT
`0x400`	MISS
`0x000`	MISS

What is the associativity of the cache, i.e., the number of lines per set?

4. If we increment the accesses by the size of the cache, i.e., `0x100`, the access to `0x000` becomes a miss after the 4th increment.

(e) [2 POINTS] What is the number of bits required to encode the cache index?

3 bits. There are 256 / (4*8) = 8 sets. Hence 3 bits is enough.

Question 7A. Linking

(a) [7 POINTS] The four stages of compiling a C program involves *preprocessing*, *compiling*, *assembly*, and *linking*. Select all the steps involved *during* the given events.

Compilation				Event
preprocessing	*compiling*	*assembly*	*linking*	Generation of assembly code
preprocessing	*compiling*	*assembly*	*linking*	Generation of relocation entries
preprocessing	*compiling*	*assembly*	*linking*	Generation of object code
preprocessing	*compiling*	*assembly*	*linking*	Insertion of code(s) from the contents of other files
preprocessing	*compiling*	*assembly*	*linking*	Resolution of relocation entries
preprocessing	*compiling*	*assembly*	*linking*	Figuring out which variables will go into the `.rodata`, `.data`, and `.bss` sections
preprocessing	*compiling*	*assembly*	*linking*	Merging separate code and data sections into single sections.

(b) [3 POINTS] Consider the following executable `main`, generated with compiling and linking two source files, `main.c` and `foo.c`, using the command `gcc -o main main.c foo.c`

Considering the contents of `main.c` and `foo.c` given below,

```c
/* main.c */
#include <stdio.h>

static int x = 0;
int y = 1;
int z = 2;

void foo();

int main()
{
    int y = 4;
    foo();
    printf("x=%d, y=%d, z=%d\n", x, y, z);
    return 0;
}
```

```c
/* foo.c */
int x, y, z;

void foo()
{
    x = 9;
    y = 8;
    z = 7;
}
```

what will be the output of `main` when executed?

`x=0, y=4, z=7`

Question 7B. Linking [10 POINTS]

(a) [7 POINTS] The four stages of compiling a C program involves *preprocessing*, *compiling*, *assembly*, and *linking*. Select all the steps involved *during* the given events.

Compilation				Event
preprocessing	*compiling*	*assembly*	*linking*	Generation of assembly code
preprocessing	*compiling*	*assembly*	*linking*	Generation of relocation entries
preprocessing	*compiling*	*assembly*	*linking*	Generation of object code
preprocessing	*compiling*	*assembly*	*linking*	Insertion of code(s) from the contents of other files
preprocessing	*compiling*	*assembly*	*linking*	Resolution of relocation entries
preprocessing	*compiling*	*assembly*	*linking*	Figuring out which variables will go into the `.rodata`, `.data`, and `.bss` sections
preprocessing	*compiling*	*assembly*	*linking*	Merging separate code and data sections into single sections.

(b) [3 POINTS] Consider the following executable `main`, generated with compiling and linking two source files, `main.c` and `foo.c`, using the command `gcc -o main main.c foo.c`

Considering the contents of `main.c` and `foo.c` given below,

```c
/* main.c */
#include <stdio.h>

int x = 0;
int y = 1;
static int z = 2;

void foo();

int main()
{
    int y = 6;
    foo();
    printf("x=%d, y=%d, z=%d\n", x, y, z);
    return 0;
}
```

```c
/* foo.c */
int x, y, z;

void foo()
{
    x = 9;
    y = 8;
    z = 7;
}
```

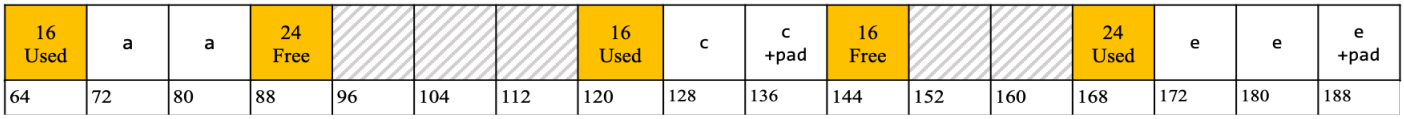
what will be the output of `main` when executed?

`x=9, y=6, z=2`

Question 8. Heap Allocators [10 POINTS]

Consider the following memory allocation operations, which are performed on the following heap with *an implicit free-list allocator*. Also note that the header is of 8 bytes and the allocated chunks of memory should be multiples of 8 bytes. For the sake of simplicity, the memory addresses are given in decimal numbers.

```
void *a = malloc(16);
void *b = malloc(20);
void *c = malloc(12);
void *d = malloc(12);
void *e = malloc(20);
free(b);
free(d);
```



(a) [2 POINTS] Does the heap currently have *internal* fragmentation?

Yes. There is padding added to the memory chunk allocated for c and e.

(b) [2 POINTS] Does the heap currently have *external* fragmentation?

Yes. There are non-consecutive free blocks.

(c) [2 POINTS] Suppose that the next call to the heap allocator is `malloc(4)`. Specify the address that the malloc function returns if a first-fit approach is employed? What if the best-fit approach is used?

Using first-fit: 96
Using best-fit: 152

(d) [2 POINTS] Considering the layout of the heap shown at the top, what is the maximum size one can ask for the heap allocator for `realloc(a,size)`?

24. The implicit free-list allocator does not perform in-place realloc and always move the data to a different location. The maximum such chunk is of 24 bytes.

(e) [2 POINTS] Suppose that you are considering an alternative heap allocator implementation, which uses an explicit free-list allocator. Does this change increase the throughput for allocation requests or not?

Yes. Because it is linear in the number of only free chunks.