08Ae: Memory Management

Instructions

Read through the exercise and answer the questions. Turn in your answers on Canvas once done.

These are all fine:

- Print out pages, fill out, scan/photograph to turn in.
- Turn PDF into image files, fill out in paint program, save and turn in.
- Write out answers on a separate sheet of paper, scan/photograph to turn in. (Please number the answers)
- Type out answers / draw answers in a separate file, save and turn in. (Please number the answers)

Memory and programs

A program that is currently running has four main parts of memory:

SOMEPROGRAM.EXE
Heap
Stack
Static/Global
Code

The operating system itself is responsible for giving memory to programs, so we only have to worry about memory management at the *program level*. (You will learn more about how Operating Systems deal with memory in an Operating Systems course).

The Stack is where our normal variables get allocated. This includes declaring variables inside a function, or as part of a function's parameter list. When we **allocate space in the stack**, it will **automatically be deallocated** once we leave the function. The stack has **limited space**. If you accidentally call the same function in an infinite loop, you will eventually get a **stack overflow** once the stack space is out of available memory.

The Heap is a special part of memory. We have **(virtually) unlimited heap space** to allocate memory in. However, with the heap, we must **manually allocate and deallocate memory** ourselves, and all the memory must be accessed via a **pointer variable**.

The Code portion of memory is where the program's code is stored.

The Static/Global portion of memory is where global variables are stored (but you SHOULDN'T be using global variables!) and static variables (which you'll learn about later on.)

Q1: Stack, Global, Code space

Identify which parts of the following program would be part of the **stack space**, **global space**, or **code space**.

```
1  #include <iostream>
2  using namespace std;
3
4  const int TOTAL_CARS = 5;
5
6  int main()
7  {
8      int participants;
9      cout << "How many people? ";
10     cin >> participants;
11
12     int peoplePerCar = participants / TOTAL_CARS;
13
14     cout << "Fit " << peoplePerCar
15         << " people per car." << endl;
16
17     return 0;
18 }
```

a. `const int TOTAL_CARS`
○ Stack ○ Global ○ Code

b. `int participants`
○ Stack ○ Global ○ Code

c. `int peoplePerCar`
○ Stack ○ Global ○ Code

d. The program code
○ Stack ○ Global ○ Code

Memory addresses

Binary and decimal number systems

Binary is a base-2 number system, meaning we have the values 0 and 1 to work with.

Decimal is a base-10 number system (the one we use every day), meaning we have values 0 through 9 to work with.

To specify the base we're talking about, we can write binary numbers as $(\texttt{0})_2$ to specify that it's base-2. A decimal number can be written with a subscript of 10 like $(0)_{10}$ to show that it's a decimal number.

Bits and bytes

A single bit in memory can store a value of `0` or `1`. If we want to store more data, we need more bits.

If we have two bits, we can store four values:
`00` (0), `01` (1), `10` (2), `11` (3).

If we have three bits, we can store eight values:
`000` (0), `001` (1), `010` (2), `011` (3), `100` (4), `101` (5), `110` (6), `111` (7).

The amount of data we can store is equivalent to 2^n, where we have n bits.

With most programming we do, we work with **bytes** instead of bits, where 1 byte = 8 bits. With 1 byte, we can store 0 to $2^8 - 1$, or 0 through 255.
0 is $(\texttt{0000 0000})_2$ and 255 is $(\texttt{1111 1111})_2$.

Variable sizes

When we create a variable in our program, memory is set aside in RAM to store the value we're going to store. How much RAM allocated is based on the size of the variable you've declared: `bool` and `char` are 1 byte, `int` and `float` are 4 bytes, and a `double` is 8 bytes.

Q2: Sizes of data types

This table represents a series of bytes in memory:

Address	0	1	2	3	4	5	6	7

Shade in the appropriate amount of blocks for each data type, starting with position 0 and going forward however many bytes are needed for the data type.

a. A `bool`:

0	1	2	3	4	5	6	7

b. An `int`:

0	1	2	3	4	5	6	7

c. A `char`:

0	1	2	3	4	5	6	7

d. A `double`:

0	1	2	3	4	5	6	7

e. A `float`:

0	1	2	3	4	5	6	7

* Note that actual memory addresses don't use decimal numbers (0 through 9) to represent memory addresses. Usually, these are base-16 numbers (0-9, A-F).

Address-of operator &

In C++, we can use the **address-of operator** (&) to access the address of any variable.

```
1 int num;          // Declare integer
2 cout << &num;     // Display the address
```

The variable's memory address will be different each time we run the program; Whatever memory is free and available will be allocated once the variable is declared. The memory address will look like a hexidecimal number (base-16 number) like this: `0x7ffd3a24cc70` (this is because converting between binary and hex is easy).

0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb
num											

(*Note: Addresses may not always start at an address ending with 0; these are just examples.)

If we're creating an array, each element of the array will be **contiguous in memory**. An array of 3 integers will essentially be like declaring 3 different integers together, but they will be neighbors in memory.

```
1 int arr[3];
2 cout << &arr[0] << endl;          // Would show address 0
3 cout << &arr[1] << endl;          // Would show address 4
4 cout << &arr[2] << endl;          // Would show address 8
```

0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb
arr[0]				arr[1]				arr[2]			

Q3: Address of variables

Given four variables declared:

```
int age;
char choice;
float price;
bool quit;
```

We have a diagram of memory where these variables were allocated:

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b
...	...	age				...	...	...	...	quit	...

0x0c	0x0d	0x0e	0x0f	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
price				...	...	...	choice	...	...	...	...

Identify the memory address for each:

a. `&age`

b. `&choice`

c. `&price`

d. `&quit`

Pointer variables

There are a special type of variables in C++ called **pointers**. Pointer variables are just another type of variable - you give it a name, and it stores some value - except instead of storing strings or ints or floats, a **pointer stores a memory address**. You can assign a value to a pointer by using the address-of operator to access another variable's address.

```
1 int myNum;                    // Normal integer variable
2 int * myPointer = &myNum;     // Store myNum's address
```

You can update a pointer's value any time throughout the program; during some parts, it might point to `integer1`, other times it might point to `integer2`.

Q4: Pointing to addresses

Given the diagram of memory:

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b
...	...	age1				...	...	age2			

0x0c	0x0d	0x0e	0x0f	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
age3				age4				...	...	...	

A variable has been declared...

```
1 int * myPointer;
```

Identify the address stored in the pointer when the pointer is pointing to different variables' addresses:

a. `myPointer = &age1;`
myPointer is pointing to age1, so the address it's storing is:

b. `myPointer = &age2;`
myPointer is pointing to age2, so the address it's storing is:

c. `myPointer = &age3;`
myPointer is pointing to age3, so the address it's storing is:

Dereferencing pointers to get values

Why would we even bother to store the address of a variable? Well, we can use the **de-reference** operator to look inside that memory address to **read and write** in that location.

```
1 // Output what is currently stored in the memory address
2 cout << *myPointer;
3
4 // Overwrite the value at that address
5 cin >> *myPointer;
```

Q5: Pointing to addresses

These variables have been declared and assigned values:

```
1 int age1 = 35;
2 int age2 = 18;
3 int age3 = 24;
4 int age4 = 52;
```

Here they are in memory:

0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b
...	...	age1				...	...	age2			

0x0c	0x0d	0x0e	0x0f	0x10	0x11	0x12	0x13	0x14	0x15	0x16	0x17
age3				age4				...	...	...	

Given this information, identify the **value** stored when...

a. `myPointer` is pointing to memory address 0x08. The value is:

b. `myPointer` is pointing to memory address 0x02. The value is:

c. `myPointer` is pointing to memory address 0x10. The value is: