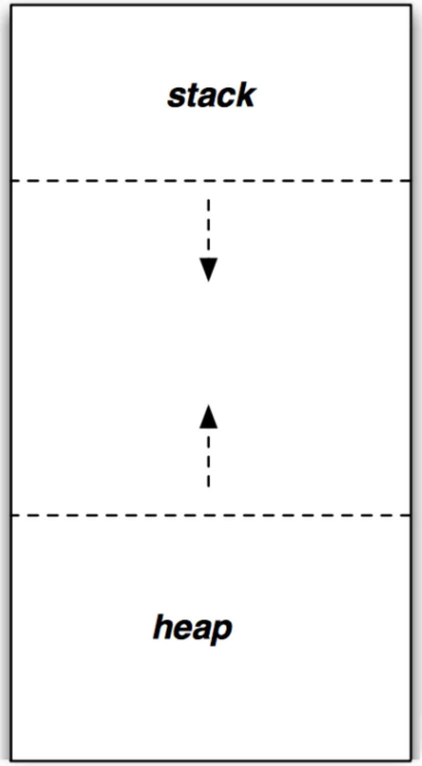
The National School of Artificial Intelligence
المدرسة الوطنية العليا للذكاء الاصطناعي



Data Structure & Algorithms 1

CHAPTER 6: DYNAMIC MEMORY ALLOCATION (INTRODUCTION)

Sep – Dec 2023

Introduction

Motivation:

We often don't know how much space we will need to store things at "compile time" → `int array[Max-size]`

Dynamic memory allocation is the allocation of memory at "run time“

Introduction

Differences between Static & Dynamic Memory Allocation:

- Dynamically allocated memory is kept on the memory **heap** (also known as the free store)
- Dynamically allocated memory can't have a "name" it must be referred to
- **`Declarations`** are used to statically allocate memory,

  the **`new`** operator is used to dynamically allocate memory

Introduction to Pointer

What is a pointer?

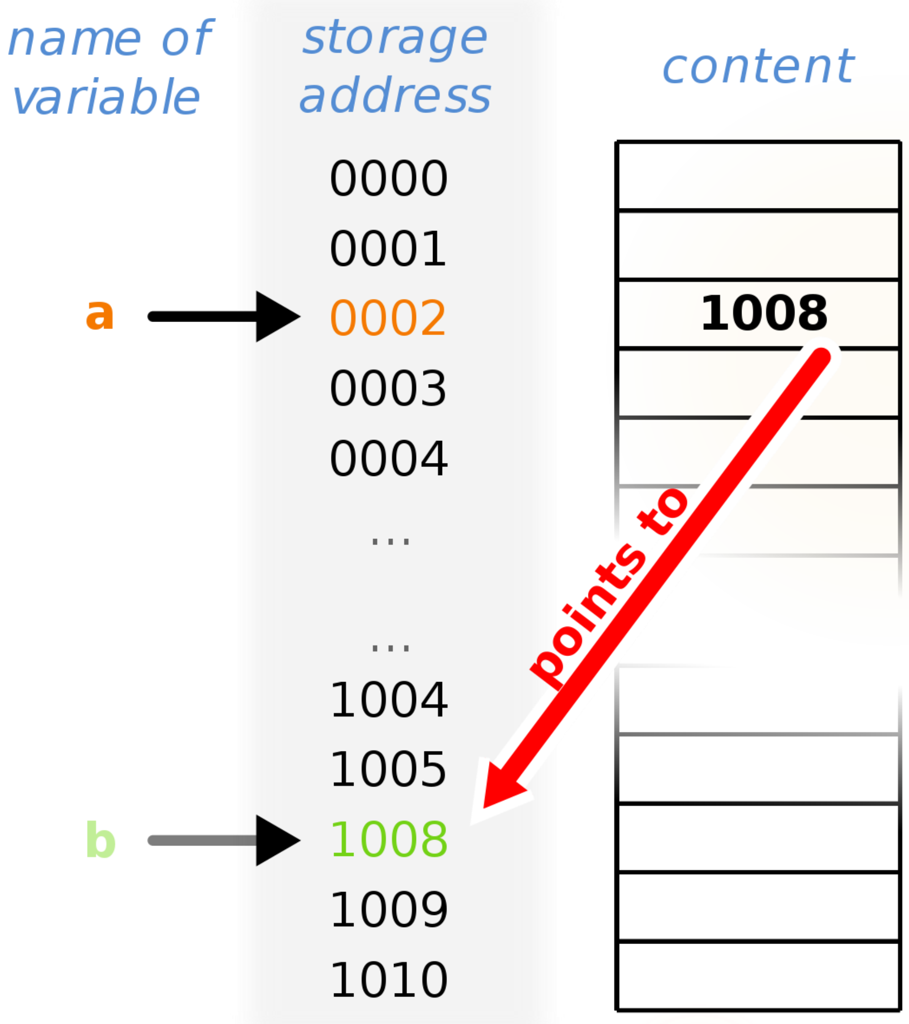
a memory address!

a variable that store memory address!

Introduction to Pointer

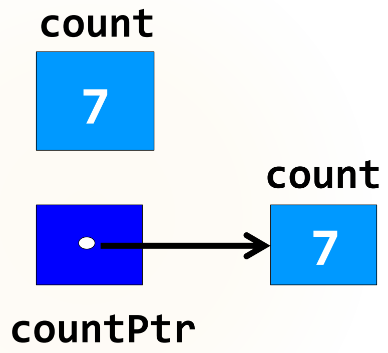
A pointer **a** pointing to the memory address associated with a variable **b**,

i.e., **a** contains the memory address 1008 of the variable **b**.



Introduction to Pointer

- Pointer variables
  - Normally, variable contains specific value (direct reference)
  - Pointers contain address of variable that has specific value (indirect reference)



- Pointer declarations
  - `*` indicates variable is pointer

```
int *myPtr;
```

  declares pointer to `int`, pointer of type `int *`

  - Multiple pointers require multiple asterisks

```
int *myPtr1, *myPtr2;
```

- Pointer initialization
  - Initialized to `0`, `NULL`, or address
    - `0` or `NULL` points to nothing

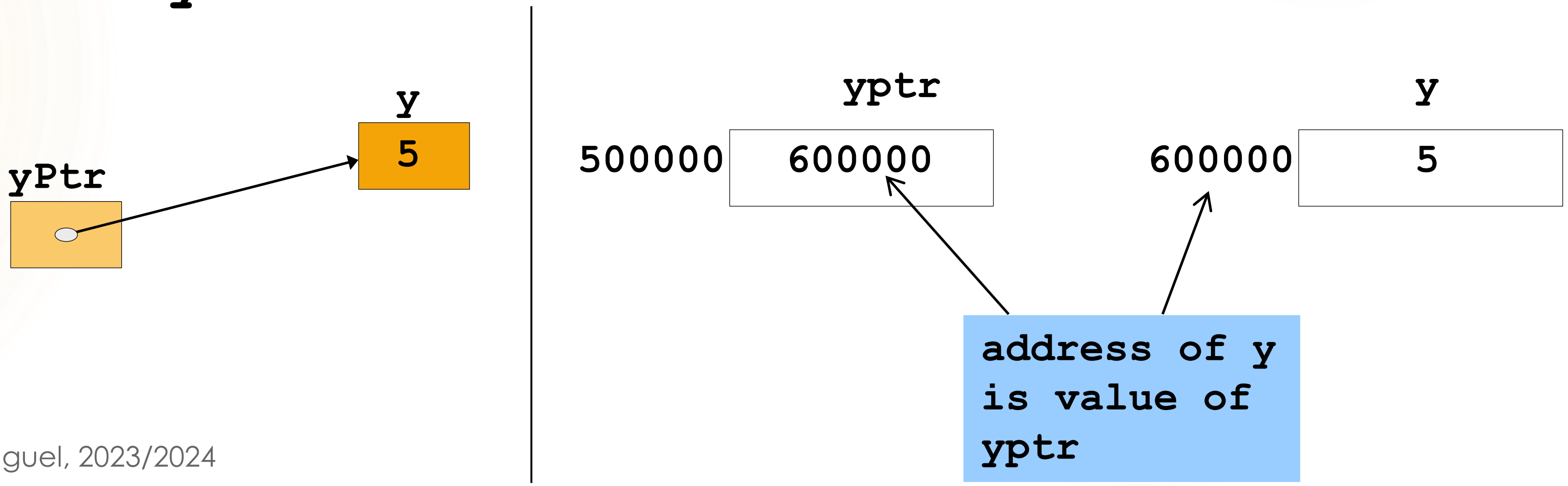
Pointer Operators

& (address operator)

- Returns memory address of its operand
- Example:

```
int y = 5;
int *yPtr;
yPtr = &y;    // yPtr gets address of y
```

- **`yPtr`** “points to” **`y`**



address of y is value of yptr

Pointer Operators

* (indirection/dereferencing operator)

- Utilized with the asterisk symbol **(*)**
- Yields a synonym for the object pointed to by its operand
- Example: `*yPtr` returns `y` (as `yPtr` points to `y`)
- The dereferenced pointer is an `lvalue`
- Example: `*yPtr = 9; //assigns 9 to y`
- The operators * and & are inverses of each other

Pointer Operators

```
// Using the & and * operators.
#include <iostream>

using std::cout;
using std::endl;

int main()
{
   int a;      // a is an integer
   int *aPtr;  // aPtr is a pointer to an integer

   a = 7;
   aPtr = &a;  // aPtr assigned address of a

   cout << "The address of a is " << &a
        << "\nThe value of aPtr is " << aPtr;

   cout << "\n\nThe value of a is " << a
        << "\nThe value of *aPtr is " << *aPtr;

   cout << "\n\nShowing that * and & are inverses of "
        << "each other.\n&*aPtr = " << &*aPtr
        << "\n*&aPtr = " << *&aPtr << endl;

```

***** and **&** are inverses of each other

Pointer Operators

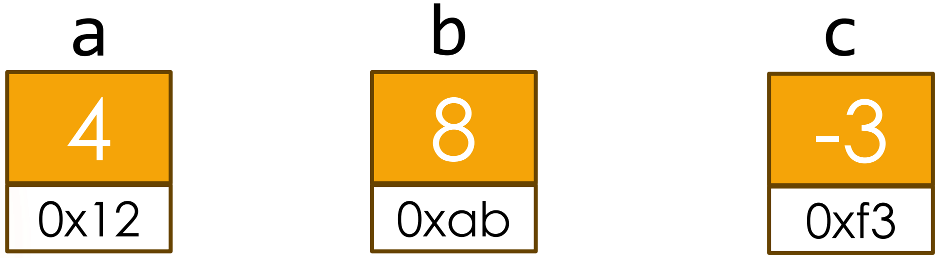
```
   return 0;  // indicates successful termination

} // end main
```

```
The address of a is 0012FED4
The value of aPtr is 0012FED4

The value of a is 7
The value of *aPtr is 7

Showing that * and & are inverses of each other.
&*aPtr = 0012FED4
*&aPtr = 0012FED4
```

`*` and **`&`** are inverses; same result when both applied to **`aPtr`**

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
        a++;
        (*c)--;
        b += *c;
        cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
        int a = 4;
        int b = 8;
        int c = -3;
        cout << a << " " << b << " " << c << " " << endl;
        mystery(c, a, &b);
        cout << a << " " << b << " " << c << " " << endl;
        return 0;
}
```



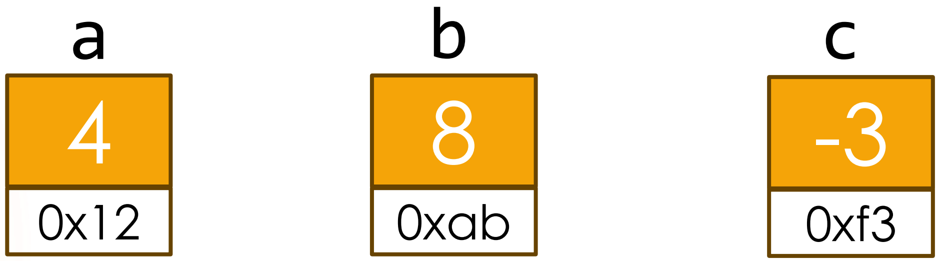
a | b | c
4 | 8 | -3
0x12 | 0xab | 0xf3

Answer:

[redacted]

Mystery Function: What prints out?

```cpp
void mystery(int a, int& b, int* c) {
        a++;
        (*c)--;
        b += *c;
        cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
        int a = 4;
        int b = 8;
        int c = -3;
        cout << a << " " << b << " " << c << " " << endl;
        mystery(c, a, &b);
        cout << a << " " << b << " " << c << " " << endl;
        return 0;
}
```

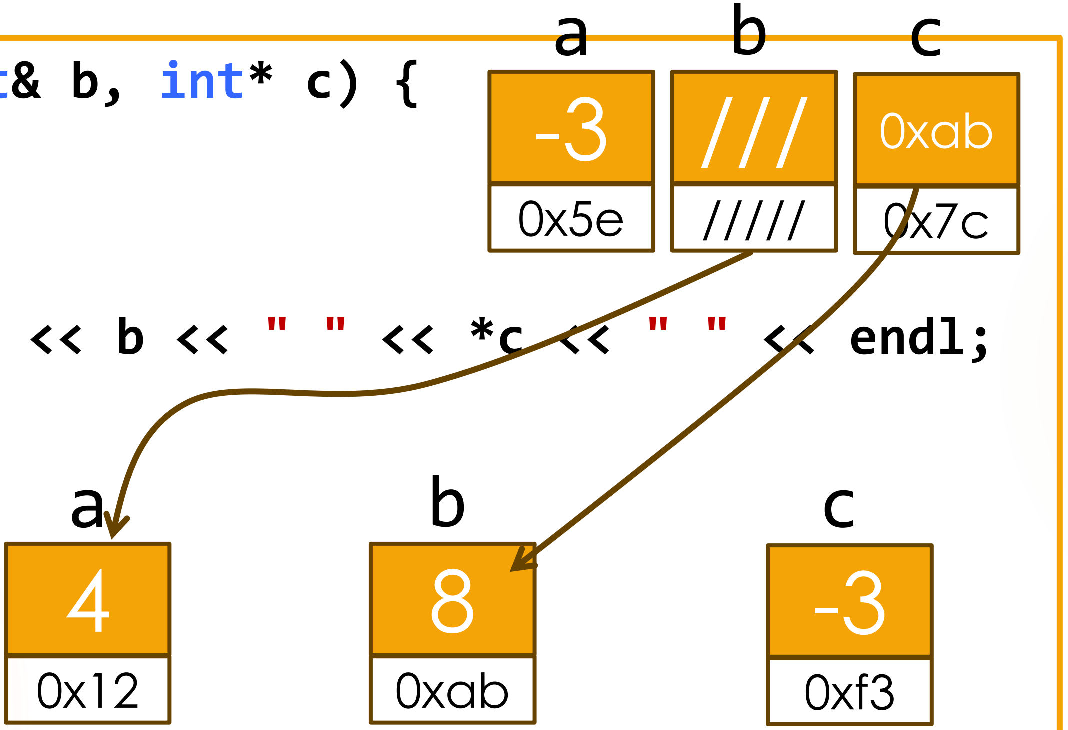


Answer:

```
4          8          -3
```

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
      a++;
      (*c)--;
      b += *c;
      cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
      int a = 4;
      int b = 8;
      int c = -3;
      cout << a << " " << b << " " << c << " " << endl;
      mystery(c, a, &b);
      cout << a << " " << b << " " << c << " " << endl;
      return 0;
}
```

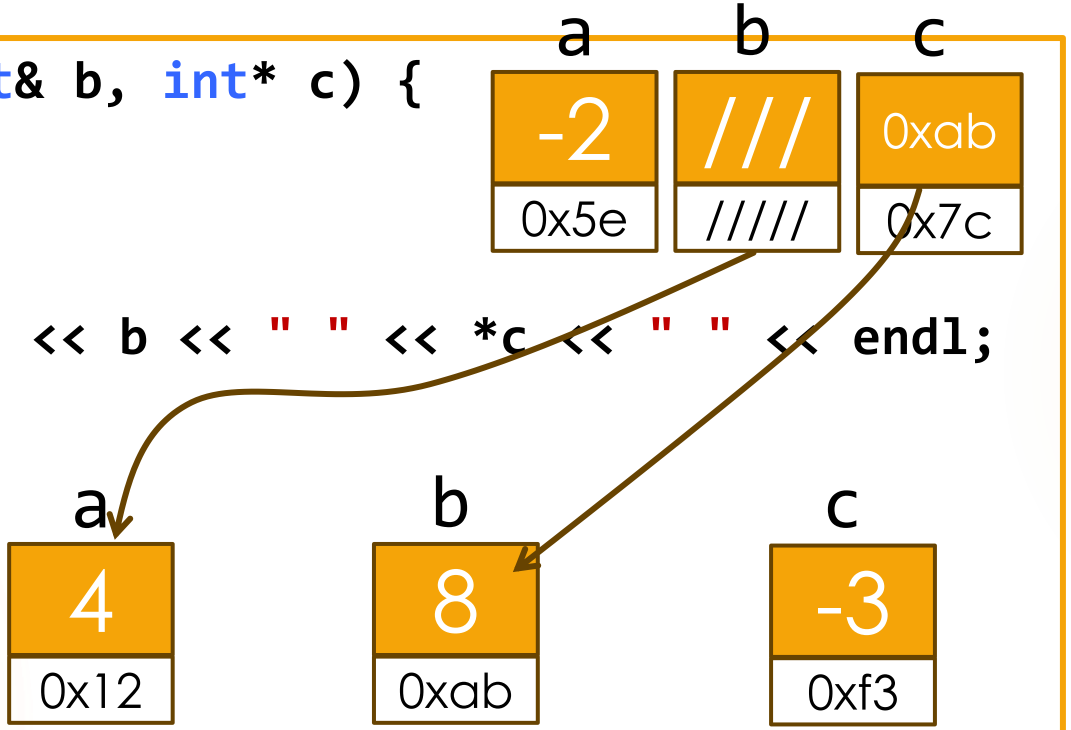


Answer:

```
4          8          -3
```

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
      a++;
      (*c)--;
      b += *c;
      cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
      int a = 4;
      int b = 8;
      int c = -3;
      cout << a << " " << b << " " << c << " " << endl;
      mystery(c, a, &b);
      cout << a << " " << b << " " << c << " " << endl;
      return 0;
}
```

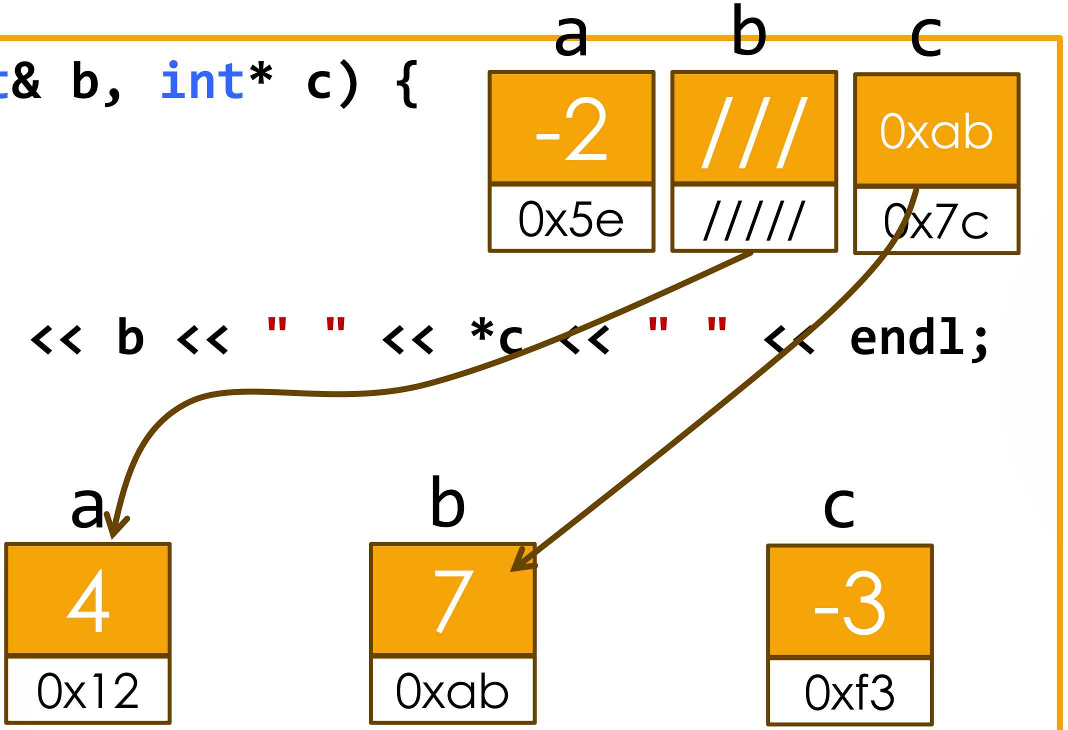


Answer:

```
4         8         -3
```

Mystery Function: What prints out?

```cpp
void mystery(int a, int& b, int* c) {
        a++;
        (*c)--;
        b += *c;
        cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
        int a = 4;
        int b = 8;
        int c = -3;
        cout << a << " " << b << " " << c << " " << endl;
        mystery(c, a, &b);
        cout << a << " " << b << " " << c << " " << endl;
        return 0;
}
```

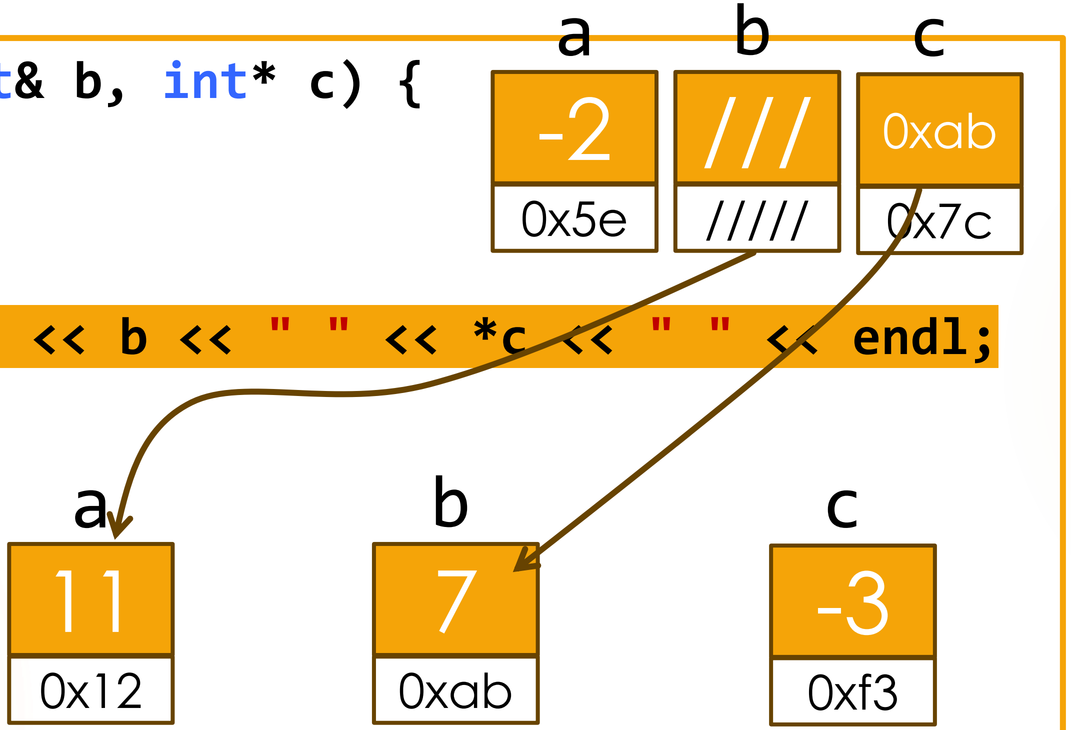


Answer:

```
4          8          -3
```

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
        a++;
        (*c)--;
        b += *c;
        cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
        int a = 4;
        int b = 8;
        int c = -3;
        cout << a << " " << b << " " << c << " " << endl;
        mystery(c, a, &b);
        cout << a << " " << b << " " << c << " " << endl;
        return 0;
}
```

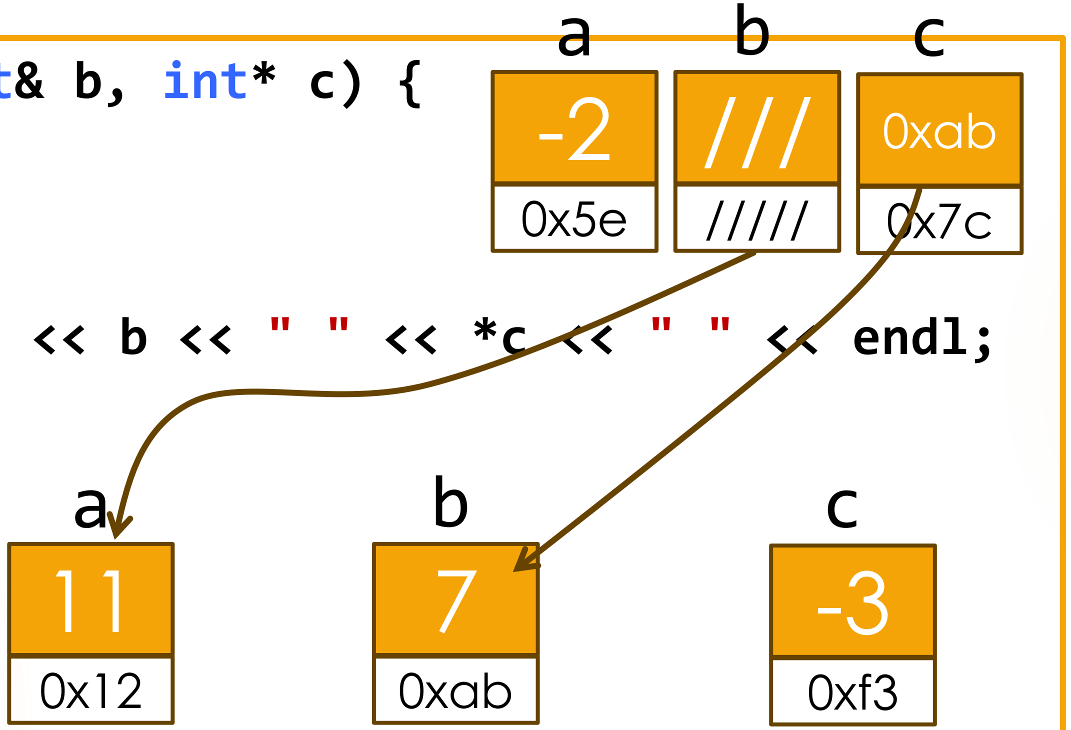


Answer:

```
4       8       -3
-2      11       7
```

Mystery Function: What prints out?

```
void mystery(int a, int& b, int* c) {
      a++;
      (*c)--;
      b += *c;
      cout << a << " " << b << " " << *c << " " << endl;
}

int main() {
      int a = 4;
      int b = 8;
      int c = -3;
      cout << a << " " << b << " " << c << " " << endl;
      mystery(c, a, &b);
      cout << a << " " << b << " " << c << " " << endl;
      return 0;
}
```



Answer:

```
4      8      -3
-2     11      7
11     7      -3
```

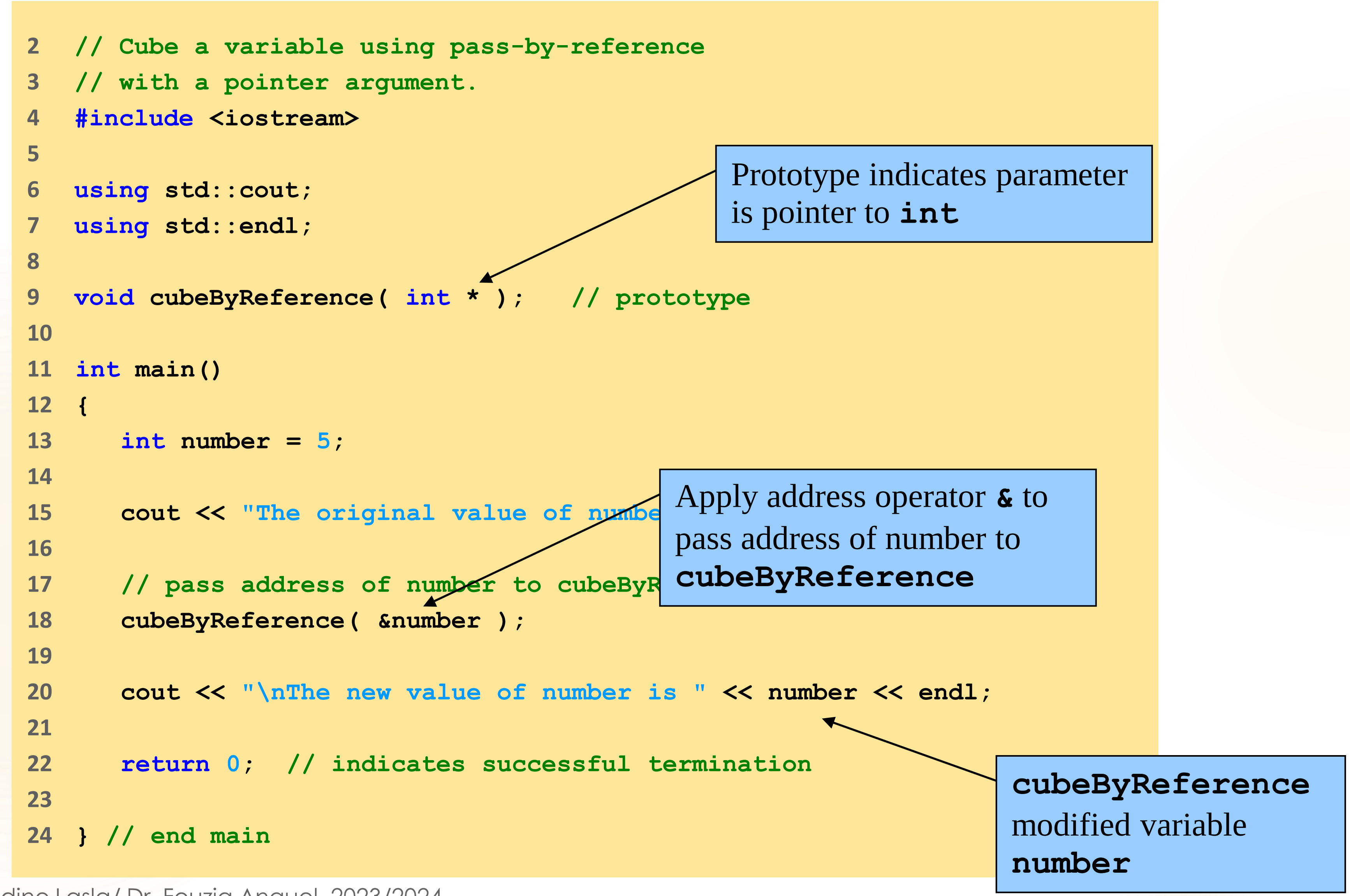
Calling Functions by Reference

- **3 ways to pass arguments to function**
  1. Pass-by-value
  2. Pass-by-reference with reference arguments
  3. Pass-by-reference with pointer arguments
- `return` can return one value from function
- **Arguments passed to function using reference arguments**
  - Modify original values of arguments
  - More than one value "returned"

Calling Functions by Reference

- **Pass-by-reference with pointer arguments**
  - Simulate pass-by-reference
    - Use pointers and indirection operator
  - Pass address of argument using **&** operator
  - Arrays not passed with **&** because array name already pointer
  - ***** operator used as alias/nickname for variable inside of function

Calling Functions by Reference



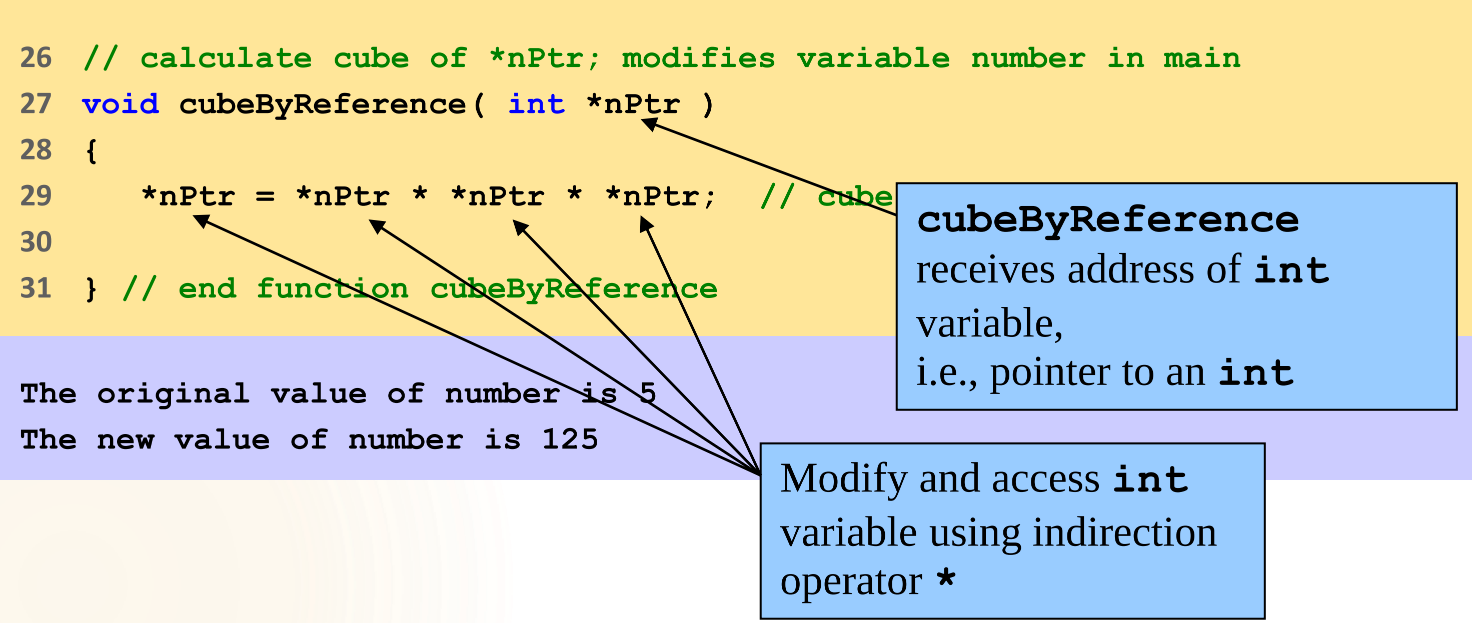
```
// Cube a variable using pass-by-reference
// with a pointer argument.
#include <iostream>

using std::cout;
using std::endl;

void cubeByReference( int * );   // prototype

int main()
{
   int number = 5;

   cout << "The original value of numbe

   // pass address of number to cubeByR
   cubeByReference( &number );

   cout << "\nThe new value of number is " << number << endl;

   return 0;  // indicates successful termination

} // end main
```

Prototype indicates parameter is pointer to **int**

Apply address operator **&** to pass address of number to **cubeByReference**

cubeByReference modified variable **number**

Calling Functions by Reference



```c
// calculate cube of *nPtr; modifies variable number in main
void cubeByReference( int *nPtr )
{
   *nPtr = *nPtr * *nPtr * *nPtr;  // cube

} // end function cubeByReference
```

```
The original value of number is 5
The new value of number is 125
```

cubeByReference receives address of **int** variable,
i.e., pointer to an **int**

Modify and access **int** variable using indirection operator *

Difference between pass-by-reference with reference and pointer

- **Syntax**:
  - In pass-by-reference with reference arguments, you use the **&** symbol to specify a reference parameter,
  - In pass-by-reference with pointer arguments, you use the ***** symbol to specify a pointer parameter.
- **Nullability**:
  - Pointers can have a <u>null value</u>, meaning they don't point to any valid memory location.
  - References <u>must always refer to a valid object</u> and cannot be null. This means that passing a null pointer to a function can led to runtime errors if the function tries to dereference the pointer.

Difference between pass-by-reference with reference and pointer

- **Pointer arithmetic**:
  - Pointers allow you to perform pointer arithmetic, which can be useful in some cases, such as iterating over arrays or linked lists.
  - References do not support pointer arithmetic.
- **Memory management**:
  - Pointers can be used to manage dynamic memory allocation and deallocation, which cannot be accomplished with references. For example, you can use **`new`** and **`delete`** operators to dynamically allocate and deallocate memory for a pointer,
  - reference must always refer to an existing object.

Pointer Expressions & Arithmetic

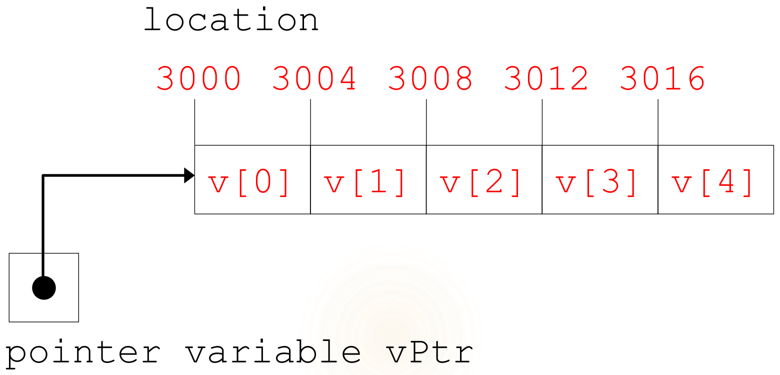
Pointer arithmetic

- Increment/decrement pointer **(++** or **--**)
- Add/subtract an integer to/from a pointer(**+** or **+=** , **-** or **-=**)
- Pointers may be subtracted from each other
- Pointer arithmetic <u>meaningless</u> unless performed on <u>pointer to array</u>

- 5 element **`int`** array on a machine using 4 byte **`int`**s
  - **`vPtr`** points to first element **`v[0]`**, which is at location 3000

    `vPtr = 3000`
  - **`vPtr += 2`**; sets **`vPtr`** to **`3008`**

    `vPtr` points to **`v[ 2 ]`**



Pointer Expressions & Arithmetic

- Subtracting pointers
  - Returns number of elements between two addresses

```
vPtr2 = v[ 2 ];
vPtr  = v[ 0 ];
vPtr2 - vPtr → 2
```

- Pointer assignment
  - Pointer can be assigned to another pointer if both of same type
  - If not same type, cast operator must be used
  - Exception: pointer to **`void`** (type **`void *`**)
    - Generic pointer, represents any type
    - No casting needed to convert pointer to **`void`** pointer
    - **`void`** pointers cannot be dereferenced

Pointer Expressions & Arithmetic

- Pointer comparison
  - Use equality and relational operators
  - Comparisons meaningless unless pointers point to members of same array
  - Compare addresses stored in pointers
  - Example: could show that one pointer points to higher numbered element of array than other pointer
  - Common use to determine whether pointer is 0 (does not point to anything)