CS1010 Tutorial 5

Agenda for Today

- Pointer Recap (30 min)
  - A small break (5 min)

- Problem Set 14 (10 min)
- Problem Set 15 (15 min)
- Assignment 2 Comments (15 min)
- Past Year PE1 (45 min)

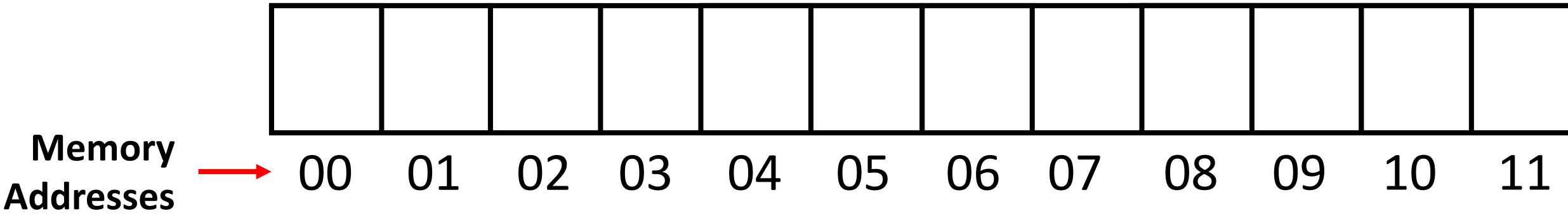
The Pointer

***

Computer Memory

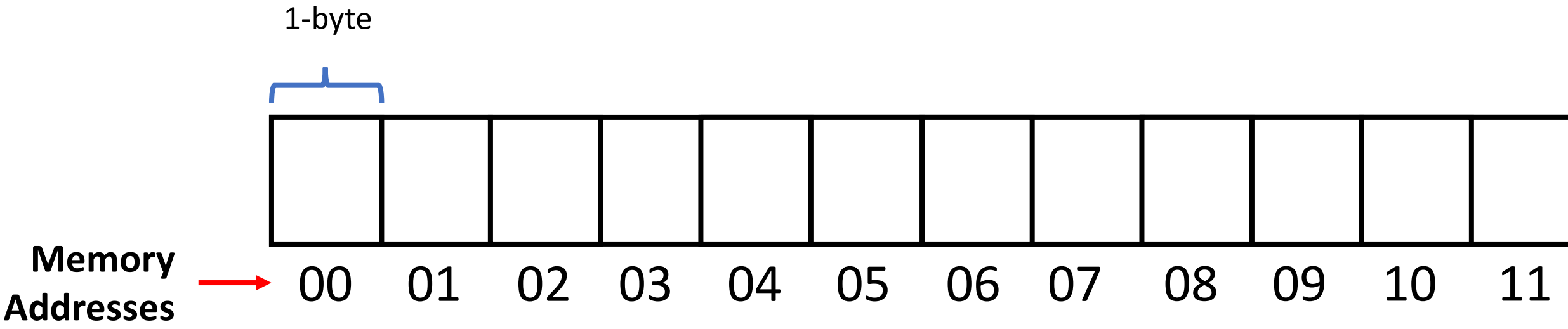
- In a computer, we need to *store* stuff in a location while we work on them
  - To do that, we need the *place* to store stuff - **computer memory**

- At the lowest level, think of computer memory as a very long table indexed by integers



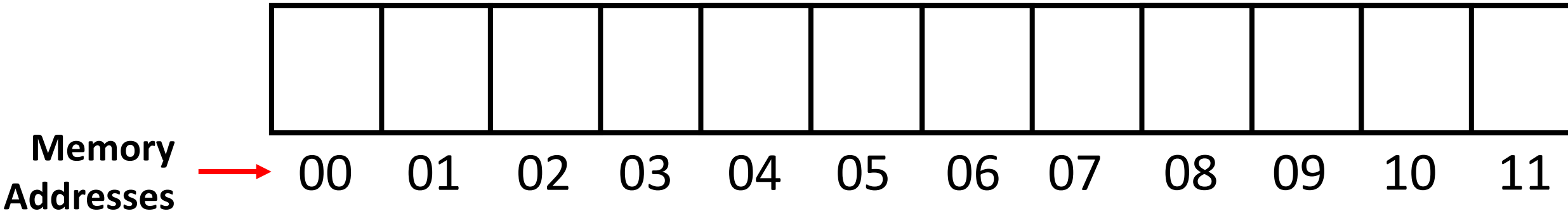
Computer Memory

- Each cell represents a “box” that we can store stuff in
- The size of each box is exactly **1-byte**



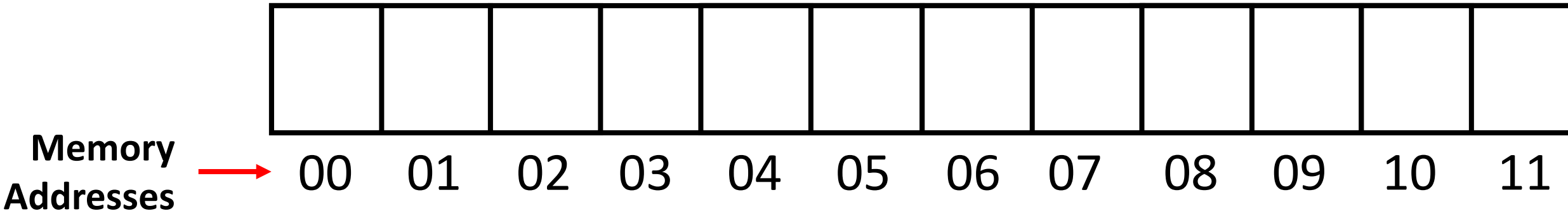
Computer Memory

- In modern computers with many GiBs of RAM, there can be billions of different memory addresses
- With **4GiB** of RAM, there are exactly **4294967296** possible memory locations
- The diagram below is the start of a very, very long table



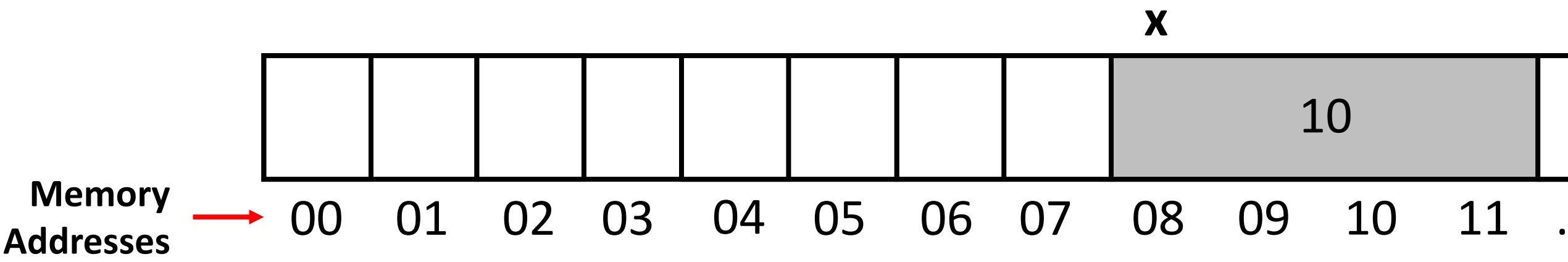
Computer Memory - Variables

- A **variable** is simply a “name” given to a memory address
- When you do **`int x = 10`**, **x** is the “name”, the value **10** is stored at *some* memory address
- We reference the value **10** by using the name **x** in code, instead of directly writing out pure memory addresses



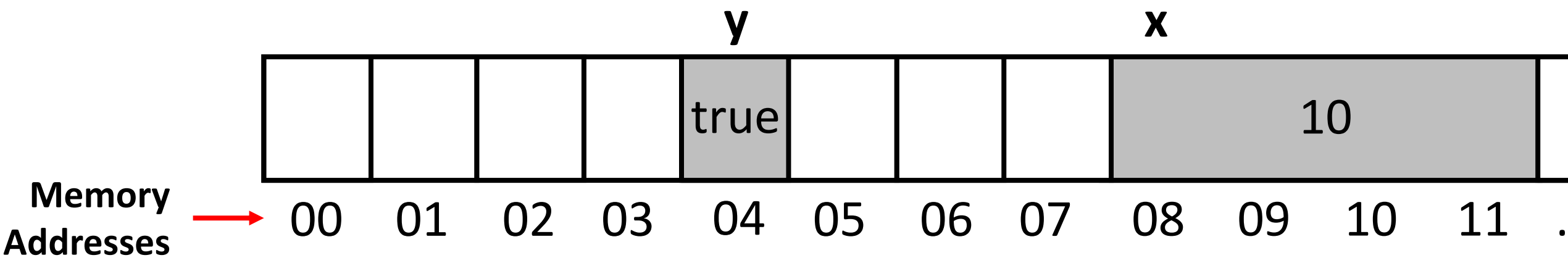
Computer Memory - Variables

- Example: **int x = 10**, x is a 32-bit integer
- What happens under the hood?
  - Say the computer uses the memory addresses {8, 9, 10, 11} to store the value 10
  - It assigns **x** to the start of the series of addresses (8) and stores the value **10** within the cells



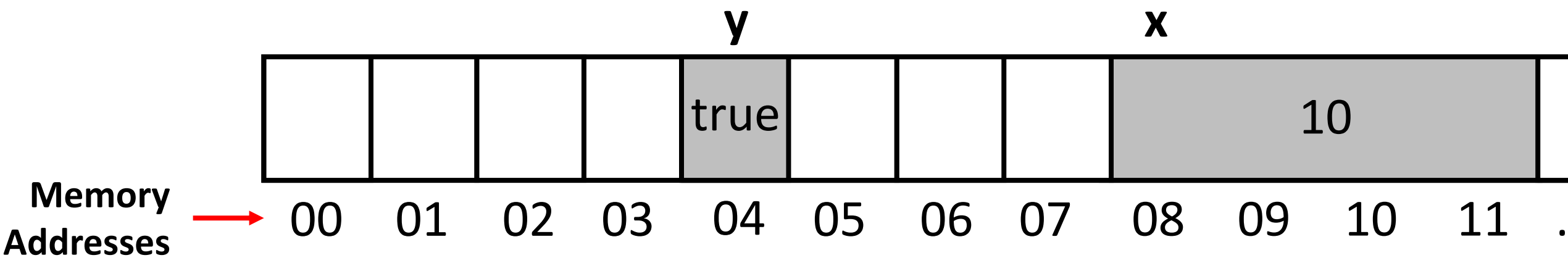
Computer Memory - Variables

- Example: **bool y = true**, x is a boolean variable (1-byte)
- What happens under the hood?
  - Say the computer uses the memory address {4} to store the value true
  - It assigns **y** to be the address (4) and replaces the cells with **true**



Computer Memory - Variables

- Now, when we refer to **x** or **y**, we reference their *values*
- **int z = x** means “assigning the value in **x** into **z**” (which is 10)



Recap so far

- Computer memory is a very long table indexed by integers
- A variable is a “name” given to a memory address
- We reference the value stored at a particular memory address through its name

The Pointer

- A pointer is a (slightly) special variable

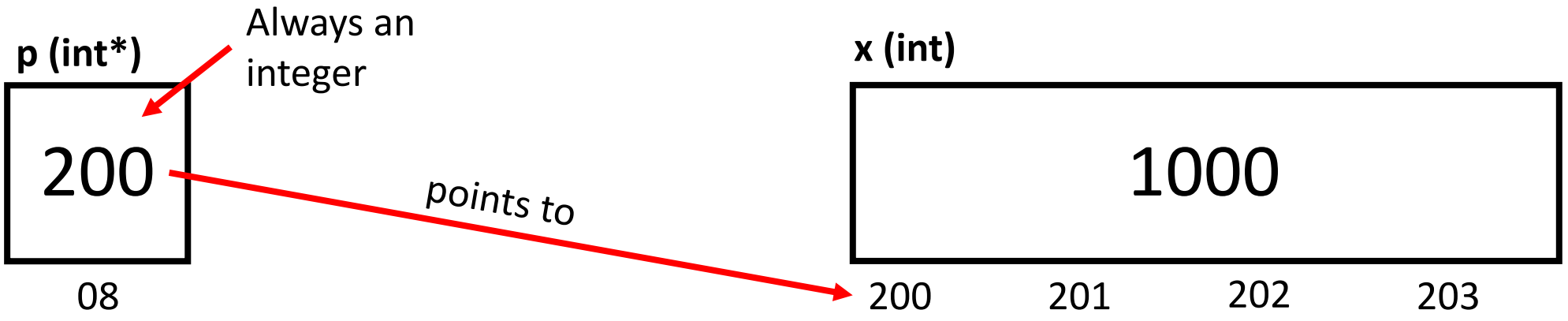
- Like normal variables, it has a value
- Like normal variables, it has a memory address

- The difference is that **the value it stores is a memory address**
  - There's some special syntax to dealing with them in C

Typing of a Pointer

- The value a pointer stores is **always** an integer

- The *type* of a pointer (e.g **int***) refers to the type stored at the address contained by the pointer



Typing of a Pointer

- The type before the * *always* refers to the type that the memory address is pointing to

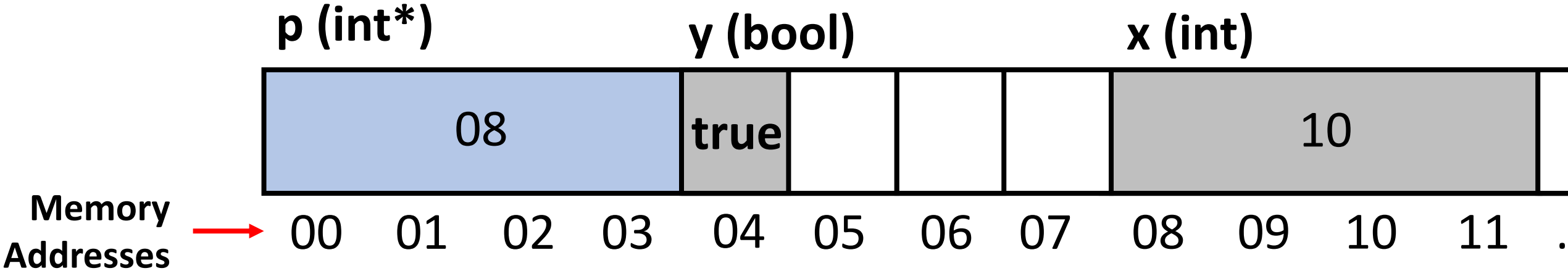
- **long *p;**
  "**p** is a pointer that points to a variable of type **long"**

- **bool *p**
  - "**p** is a pointer that points to a variable of type **bool"**

- **double *p**
  - "**p** is a pointer that points to a variable of type **double"**

The Pointer in Computer Memory

- Say we do **int *p = 8**, which the computer stores at addresses {0,1,2,3}
- This means that **p** is a name given to address {0,1,2,3}, which stores an integer (8), which is the address of **x**



The Address-Of operator

- Denoted by an **&**
- Remember that a variable is just a name given to a memory address
- Returns a **pointer** containing the **address** of the variable

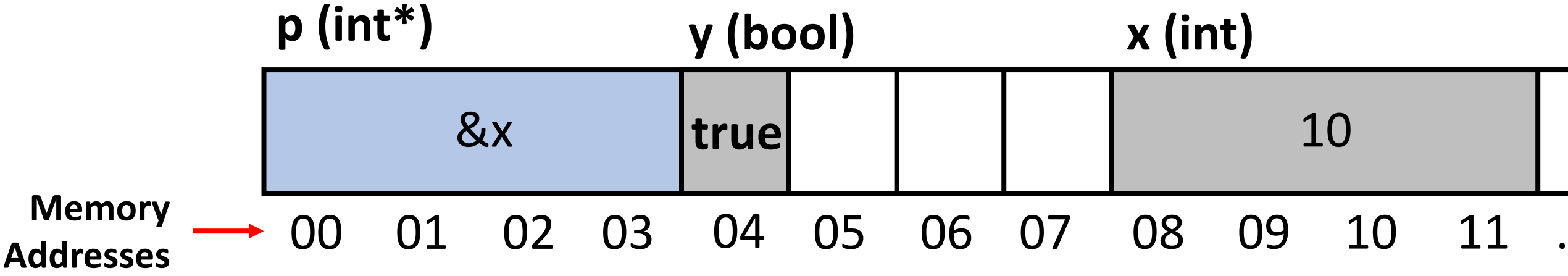
The Address-Of operator

- If **long x = 10**
- Then **&x** returns a pointer type, which contains the value of the memory address of **x**, which contains the value **10**
- Therefore, the type of **&x** will be **long***

The Pointer in Computer Memory

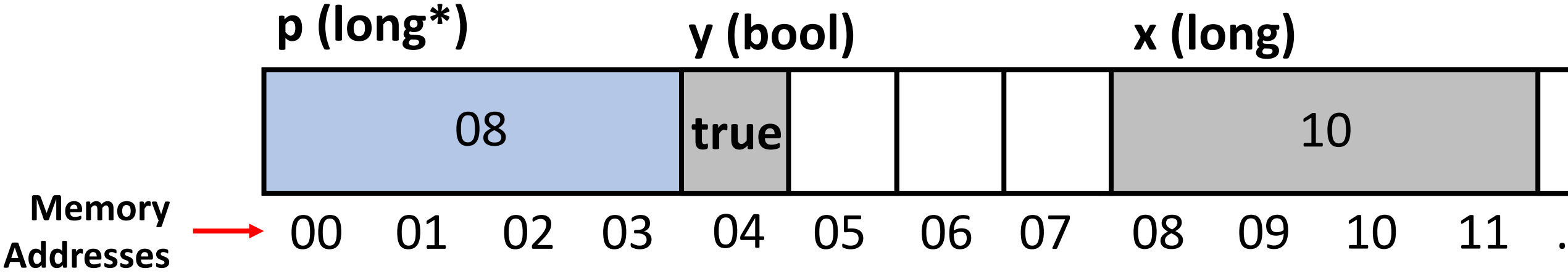
- Instead of using an integer, lets do **int *p = &x**

- This directly says that “I want **p** to be a pointer variable, that has value of the memory address of **x**”



The Address-Of operator

- In this case
  - **&p** returns a pointer which holds the value **0**
  - **&y** returns a pointer which holds the value **4**
  - **&x** returns a pointer which holds the value **8**



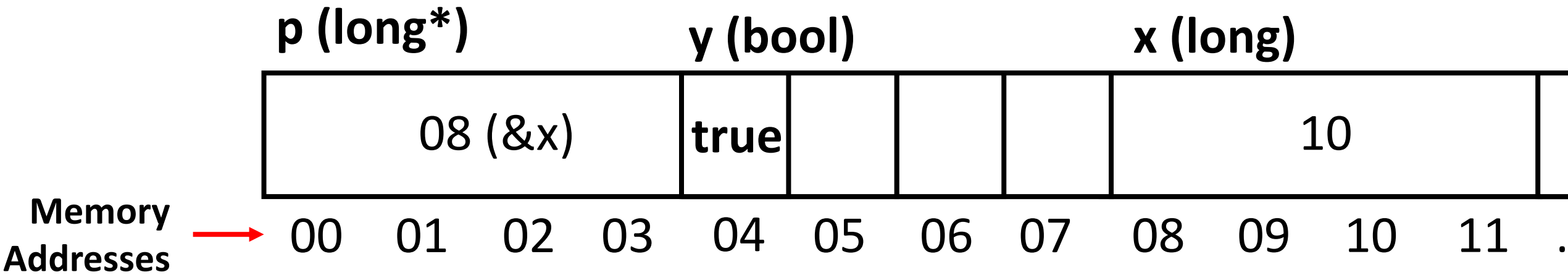
The Dereference Operator

- Denoted by an asterisk (*)
- Only can be used on pointer types
- Think of it as “I want to get the value at the address pointed to by this pointer)

The Dereference Operator

- **long *p = &x;**
- **cs1010_println_long(*p); // prints out 10**

- The * takes the memory address inside **p**, goes to it (&x), then retrieves the value stored inside it, which is **10**



```
long x = 10;
long *p = &x;

// prints the contents of p (usually a 64-bit integer)
// i.e the memory address held by p
cs1010_println_long(p);

// dereferences the memory address held by p
// p contains the memory address of x
// gets the content of x
cs1010_println_long(*p)
```

A confusing syntax

- **`long *x = &y;`**
- is equivalent to

```
long *x;
x = &y;
```

AND NOT

```
long *x;
*x = &y;
```

"Retrieve the value pointed at by **x**, then store the address of y inside it"
(Doesn't make sense, ***x** is a value, not a variable)

Arrays

- How we represent a **list** of elements
- Declare using **[]** operator
- **long array[5];** declares an array with 5 elements of **long**
- Arrays are **zero-indexed**
  - First element is at index 0
  - Followed by 1, 2, 3 and 4

Arrays - Syntax

- What are the different cases of using **[]**?

- **long array[5];**
  - Declare an array of 5 elements of long
- **array[5] = 1;**
  - Assign the value 1 to the 6^{th} element of array
- **foo(array[5]);**
  - Pass the 6^{th} element of array into **foo()**

Arrays - Syntax

- We use **long a[10]** to declare an array
- To pass in the array itself to functions, we use the name of the array itself (in this case, **a**)
- We “subscript” the name of the array with **[]** (e.g **a[5]**) to retrieve specific elements from the array

Array Decay

- The type **long a[10];** is a bit special
- These are considered arrays
- When we pass them into functions,
  - e.g **find_max(a);**
- The array “decays” into a pointer

Array Decay

- Meaning that after **foo(a);**
- What used to be the “array” is now a pointer to the first element in the array
- We can still use **a[2]** or **a[4]**, etc like normal to access the different elements of the array

Reading Array Values with CS1010 I/O

- To read an array of **10** space-separated values from the standard input, do

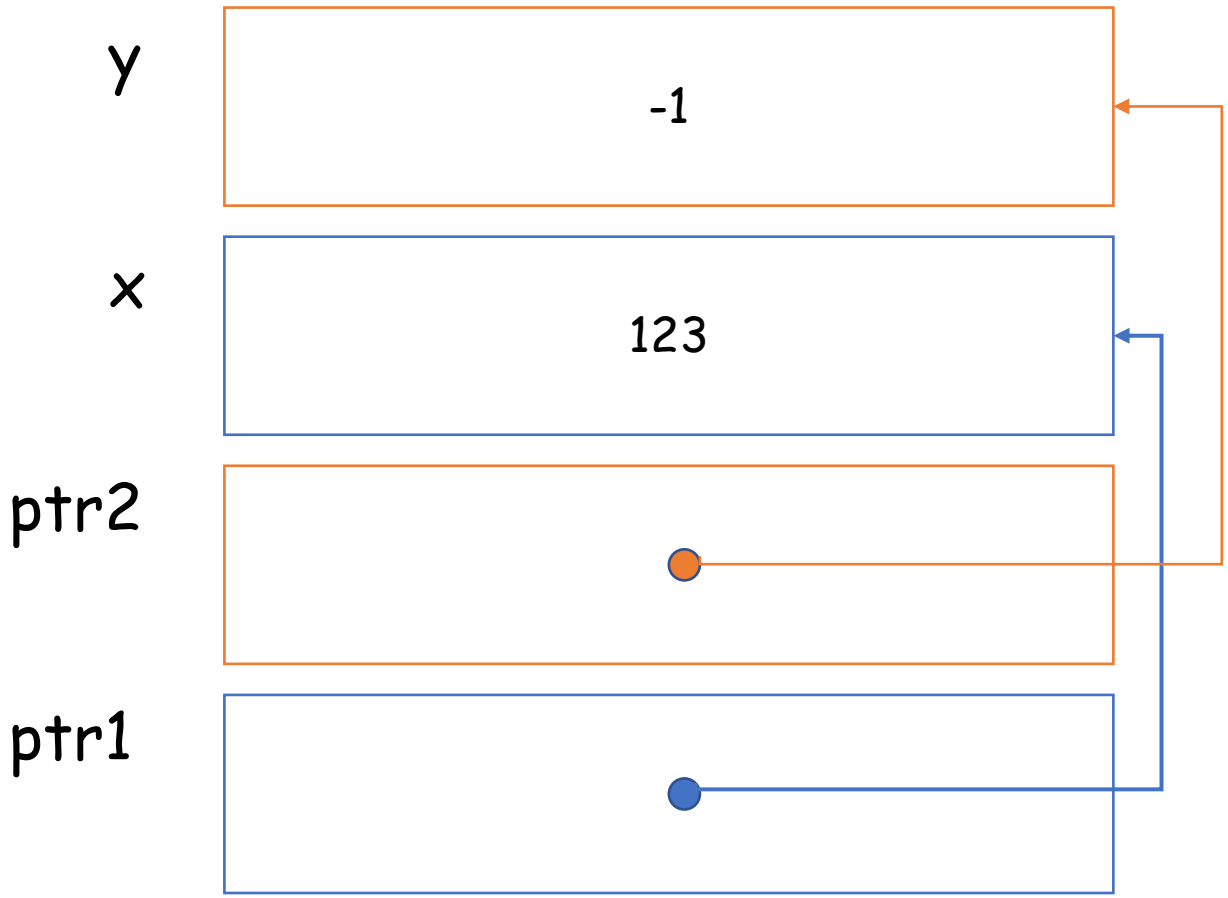
- **long *a = cs1010_read_long_array(10);**

- **a** is a pointer, and points to the first element in the list
  - **a[0]** is the first
  - **a[1]** is the second
  - etc

Problem 14.1

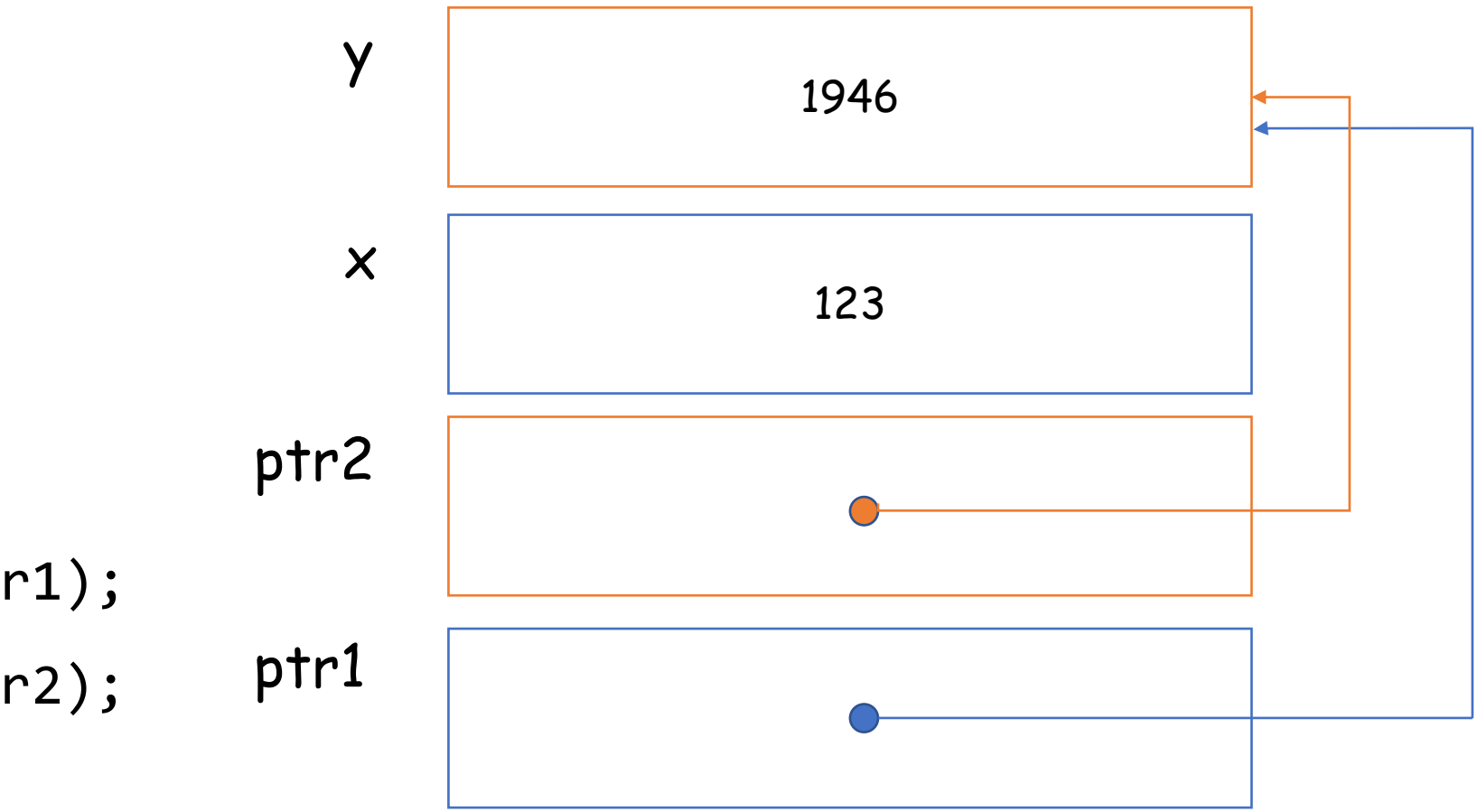
```
long *ptr1;
long *ptr2;
long x;
long y;


ptr1 = &x;
ptr2 = &y;
*ptr1 = 123;
*ptr2 = -1;
cs1010_println_long(x);
cs1010_println_long(y);
cs1010_println_long(*ptr1);
cs1010_println_long(*ptr2);
```



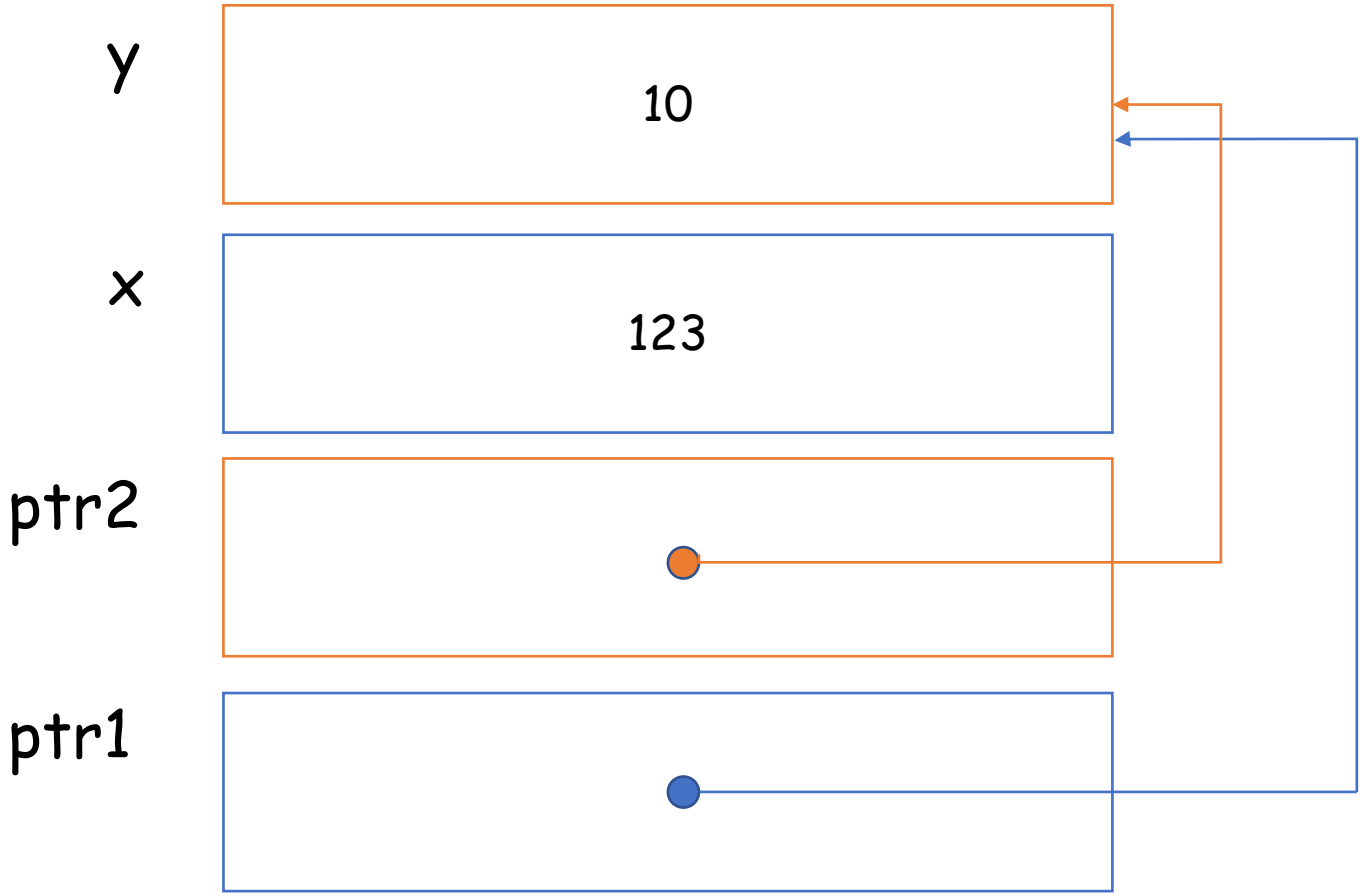
```
ptr1 = ptr2;
*ptr1 = 1946;


cs1010_println_long(x);
cs1010_println_long(y);
cs1010_println_long(*ptr1);
cs1010_println_long(*ptr2);
```



```
y = 10;


cs1010_println_long(x);
cs1010_println_long(y);
cs1010_println_long(*ptr1);
cs1010_println_long(*ptr2);
```

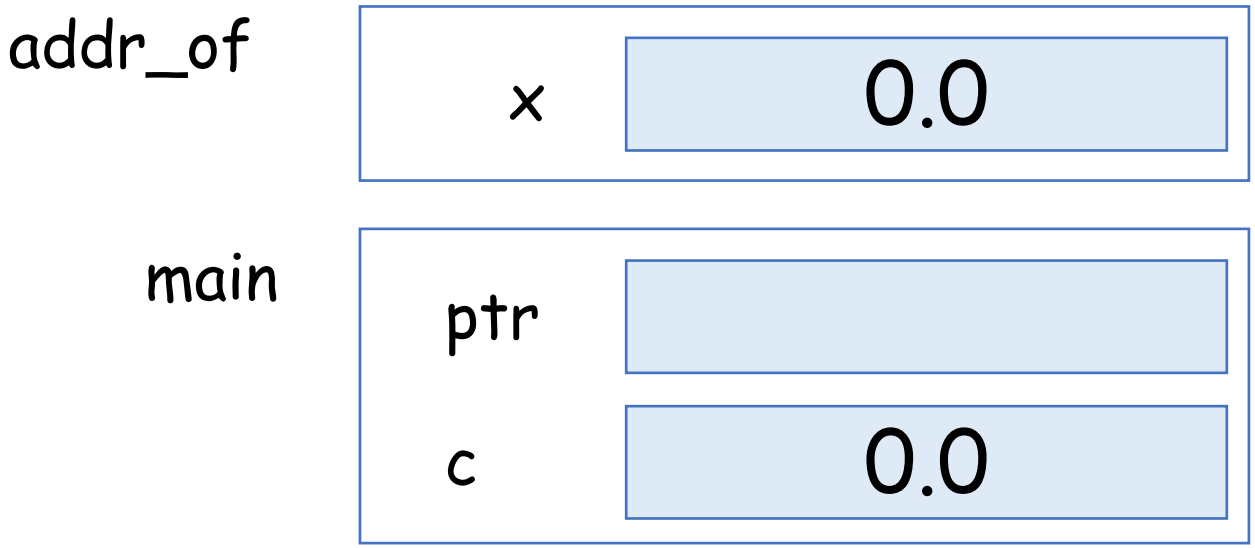


Problem 14.2

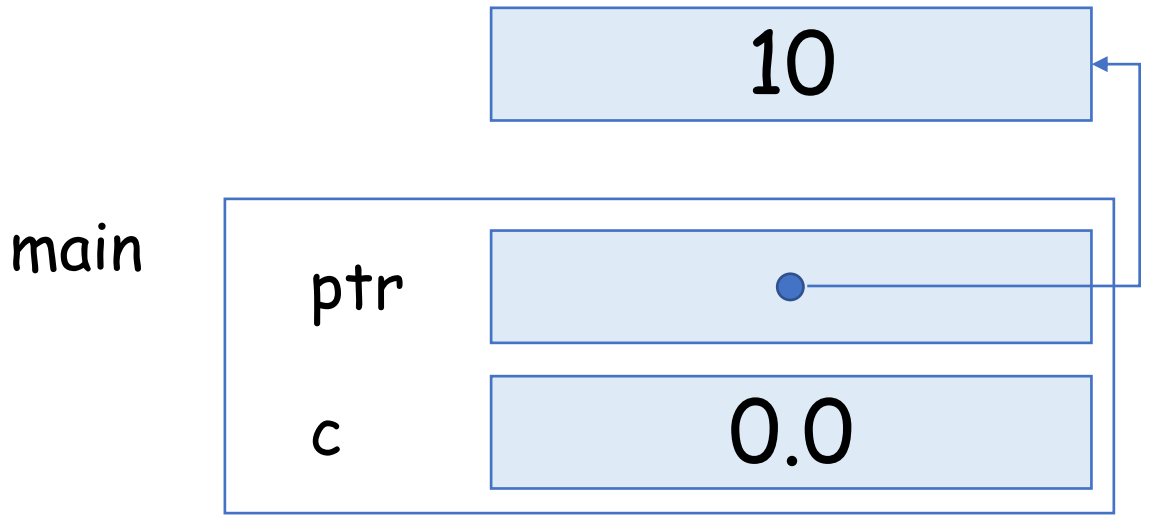
```c
double *addr_of(double x) {
  return &x;
}
int main() {
  double c = 0.0;
  double *ptr;
  ptr = addr_of(c);
  *ptr = 10;
}
```

```c
double *addr_of(double x) {
  return &x;
}
int main() {
  double c = 0.0;
  double *ptr;
  ptr = addr_of(c);
  *ptr = 10;
}
```

```c
double *addr_of(double x) {
  return &x;
}
int main() {
  double c = 0.0;
  double *ptr;
  ptr = addr_of(c);
  *ptr = 10;
}
```

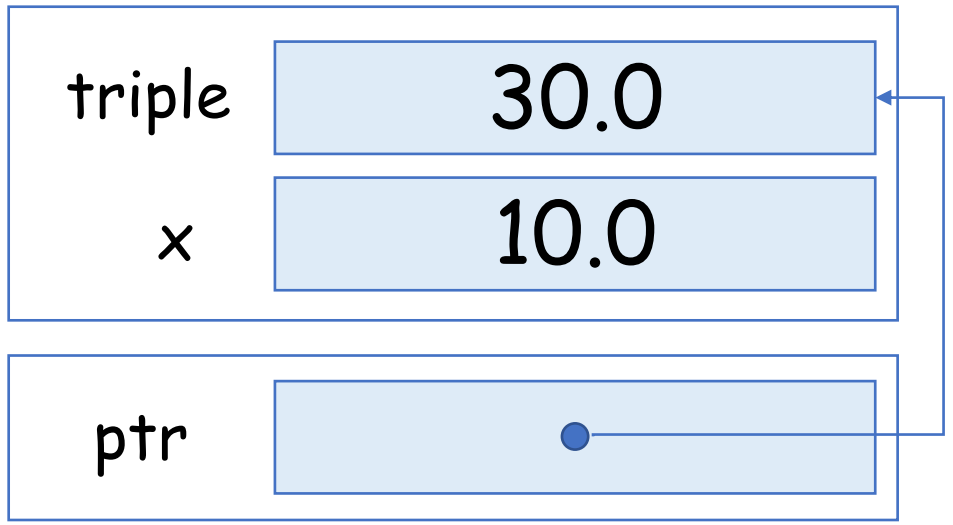


```c
double *addr_of(double x) {
  return &x;
}
int main() {
  double c = 0.0;
  double *ptr;
  ptr = addr_of(c);
  *ptr = 10;
}
```

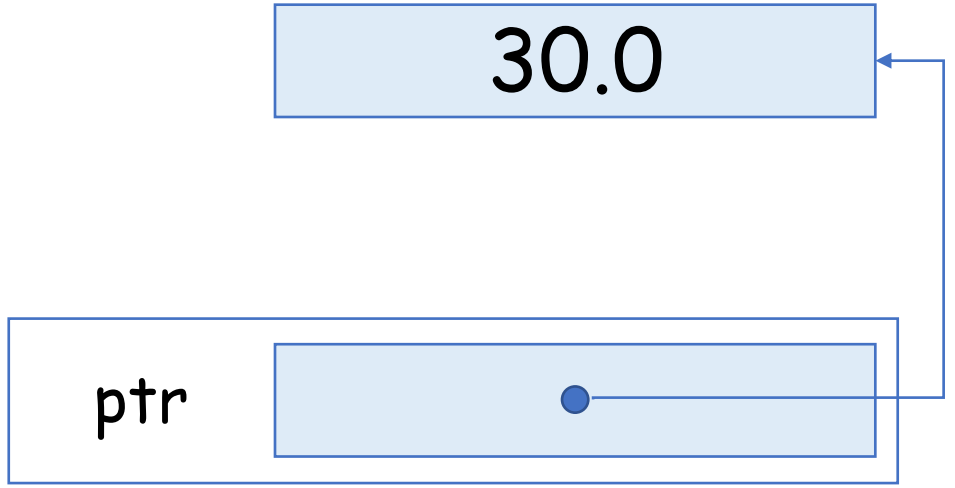


```
double *triple_of(double x) {
  double triple = 3 * x;
  return &triple;
}
int main() {
  double *ptr;
  ptr = triple_of(10);
  cs1010_println_double(*ptr);
}
```

```
double *triple_of(double x) {
  double triple = 3 * x;
  return &triple;
}
int main() {
  double *ptr;
  ptr = triple_of(10);
  cs1010_println_double(*ptr);
}
```



```c
double *triple_of(double x) {
  double triple = 3 * x;
  return &triple;
}
int main() {
  double *ptr;
  ptr = triple_of(10);
  cs1010_println_double(*ptr);
}
```



Problem 15.1

Write the function average that takes an array of k integers and returns the average of the values in the array.

Problem 15.1

- Very simple, we've done something similar in the first few lectures

```
double average(long *list, long k)
{
    double sum = 0;
    for (long i = 0; i < k; i += 1) {
        sum += list[i];
    }
    return sum / k;
}
```

Problem 15.2

- Explain why the following would lead to senseless output:

Problem 15.2

```
int main()
{
    long a = 0;
    cs1010_println_long(max(&a, 10));
}
```

Problem 15.2

```
int main()
{
    long a = 0;
    cs1010_println_long(max(&a, 1));
}
```

Problem 15.3

```
long max(long *list, long length)
{
    long max_so_far;
    long *curr;

    max_so_far = *list;
    curr = list + 1;
    for (long i = 1; i != length; i += 1) {
        if (*curr > max_so_far) {
            max_so_far = *curr;
        }
        curr += 1;
    }
    return max_so_far;
}
```

Assignment 2 Comments

Collatz

- Generally done well
- Nothing much to say…

Triangle

- Common mistake: **not printing out the correct number of trailing spaces**
- Each row has x number of '#'
- BUT also has y number of leading and trailing '#'
- **Must print correct number**

Prime

- Take note of your **IS_PRIME** function
  - Should return the correct answer for *all* possible values of n
  - **Make sure to fix for the $n = 1$ case!!**
  - No deduction of marks for this, but please fix it
    - I'd wager that the PE will use the **IS_PRIME** function

- Generally, correct bounds were found
  - Only need to check until $\sqrt{n}$ to check for primality

Pattern

- Similar to Triangle, trailing spaces should be printed out correctly
- Try to break the problem down into smaller sub-problem
  - How to get the leading number of a row
  - How to get the following numbers of a row given the leading number
  - Etc
- Generally well-done, though

Practical Exam Briefing

Good luck

Practical Exam Format

- March 6th Saturday, 9am to 12noon
- Exam duration: 2hrs and 30min
- Scope: Unit 1 to 12 (Loops), Assignments 1-2, Tutorials 1-4
- 5 questions ranging from very easy to hard
- Code can only be written in **vim**

What to expect per question

- Q1 is normally to do with C data-types
- Q2 is normally if…else statements and maybe loops
- Q3 is a recursion question
- Q4 is complex loops
- Q5 is a hard problem, may have or may not have recursion

The General Strategy

- If you've been struggling (heavily) with CS1010 so far

- **Try to complete the first 2 questions completely**
  - Make sure they are bug-free
  - Make sure there are no compiler warnings

- Questions 3 and 4 are a mixed bag
  - For the past-year PE, I felt that Q4 was easier than Q3
  - If you get stuck on Q3, try out Q4 before the end of the examination

The General Strategy

- **For stronger students**
  - Make sure to complete the first 3 questions convincingly
- If you get stuck on Q4, you could *try* Q5
  - But I recommend just sticking to debugging Q4
- Q5 is normally to differentiate the *A* from the *A+*
- If you can’t get to or complete Q5, don’t worry too much

Past Year PE1

My Sensing

- “Okay” paper
- Less focus on recursion than previous years
- Q5 > Q3 > Q4 > Q2 > Q1

Practical: Do Q4 – Factors

- Go to the module website
  - https://nus-cs1010-2021-s2.github.io/website/pe1.html

- And accept **pe98**

- **ssh** into the PE node and run **~cs1010/get-pe98**

- Try out **Question 4 – Factors**
  - If you already have, try to do Question 5

Breakdown of Q4

Given a positive integer n, we wish to factorize n into its prime factors.

Recall that factors of n are positive integers no larger than n that n can be divided by.

For example: 1, 2, 3, 4, 6, 8, 12, 24 are all the factors of 24. Among these factors, 2 and 3 are prime numbers so they are the prime factors.

Breakdown of Q4

- What **sub-problems** are there in this question?
  - How do you determine if a number is prime?
    - We know how to do this from AS02
  - Once we find a prime factor of n, how do we know how many times it can divide n by?
    - Write a loop that counts how many times n and can divided by the prime factor
  - Once we find a prime factor of n, how do we find the result of dividing n by this prime factor by however many times needed?
    - Write a loop that returns the result of dividing n by the prime factor as many times as possible
  - How do we print the relevant information to the output?