CPSC 221
Basic Algorithms and Data Structures

Crash Course on Arrays, Pointers, and Structs

Hassan Khosravi
January – April 2015

Learning goals

- Become familiar with addresses and pointers in C++.
- Describe the relationship between addresses and pointers.
- Become familiar with arrays in C++.
- Define and use records (structs) in an implementation with dynamic memory allocation.
- Demonstrate how dynamic memory management is handled in C++ (e.g., allocation, deallocation, memory heap, run-time stack).
- Gain experience with pointers in C++ and their tradeoffs and risks (dangling pointers, memory leaks).

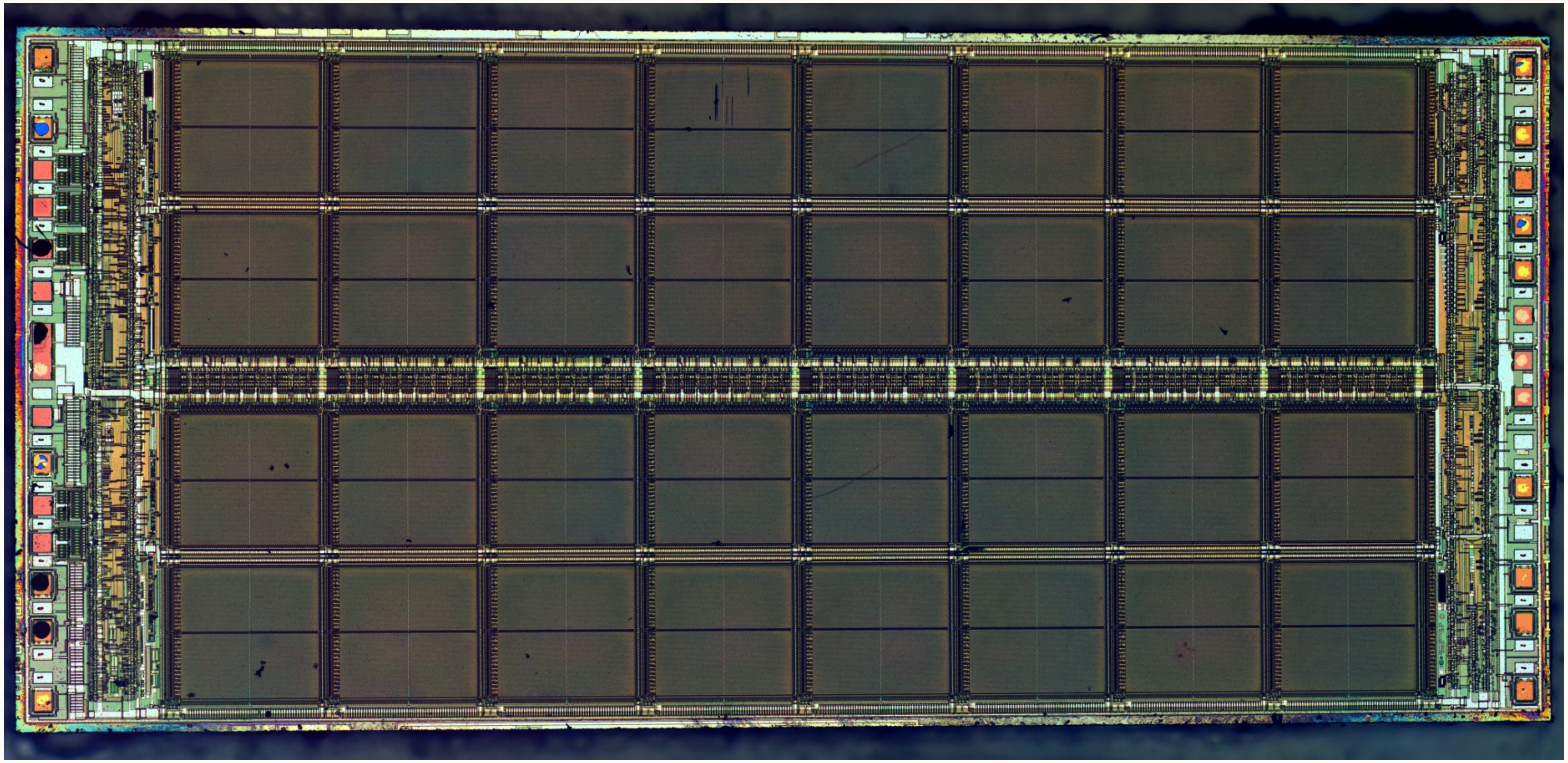
Why Arrays?

- Arrays are a very low-level data structure, that basically matches the underlying memory.
  - Good: They are very efficient!
  - Bad: They have unpleasant limitations.

Fact: Bits are real!

- Every bit of memory in your program is stored in an actual physical location on a silicon chip.
- These physical memory bits are organized into rectangular arrays, and you can quickly read/write any bit by giving its location as a **numerical** address.
- (Google DRAM to see some pictures of what memory really looks like.)

Die Photo of 1Mb DRAM



License: Creative Commons Attribution 3.0
Downloaded from Wikimedia Commons.

Source: http://zeptobars.ru/en/read/how-to-open-microchip-asic-what-inside

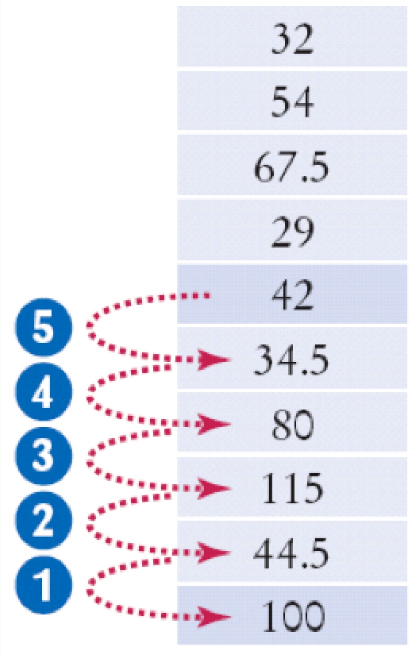
Consequences of Bits Being Real

- If you know the address where your data is, you can quickly access its memory.
- If you don’t know the address, you can’t find the data easily.
- You must work to move data. You can’t just “squeeze in” some more bits between data you’ve already stored.

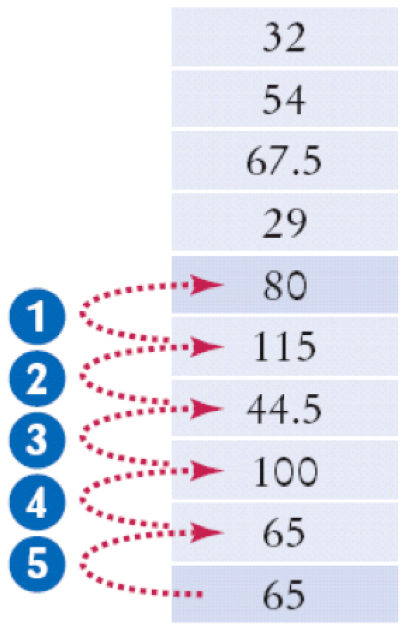
Arrays

- **Arrays have a fixed size.** They cannot grow or shrink.
- You can't insert things or delete things from the middle of an array.

Inserting into an array without holes (in which order matters)



Deleting from an array without holes (in which order matters)



Arrays vs. Java’s ArrayList

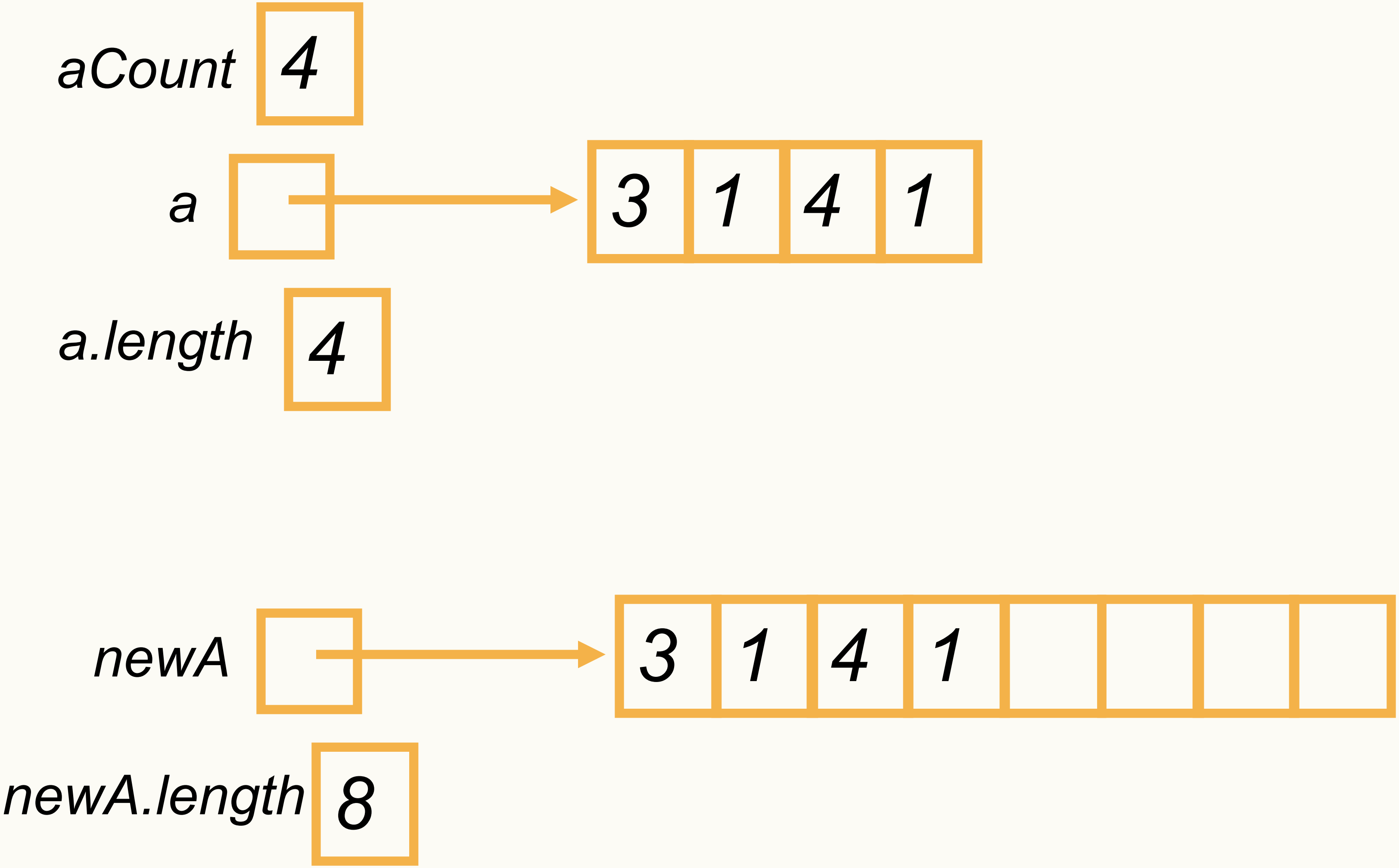
- Java provides an ArrayList class that does let you do those things. That makes programming easier.
- (But Java ArrayLists are doing things behind the scenes to make things nicer for you to program…)

- But how do you allow arrays to grow?

Real-Life Analogy: Moving Homes

- A house (or condo, apartment, etc.) has a fixed size. What happens when your family grows and you need more space?
  - Answer: You buy a bigger place, and then you pack up and move all your stuff to the new place, and get rid of your old home.
- An array has a fixed size. What happens when your list grows and you need more space?
  - Answer: You allocate a bigger array, and then you pack up and move all your stuff to the new array, and get rid of your old array

Making Your Own ArrayList



aCount 4

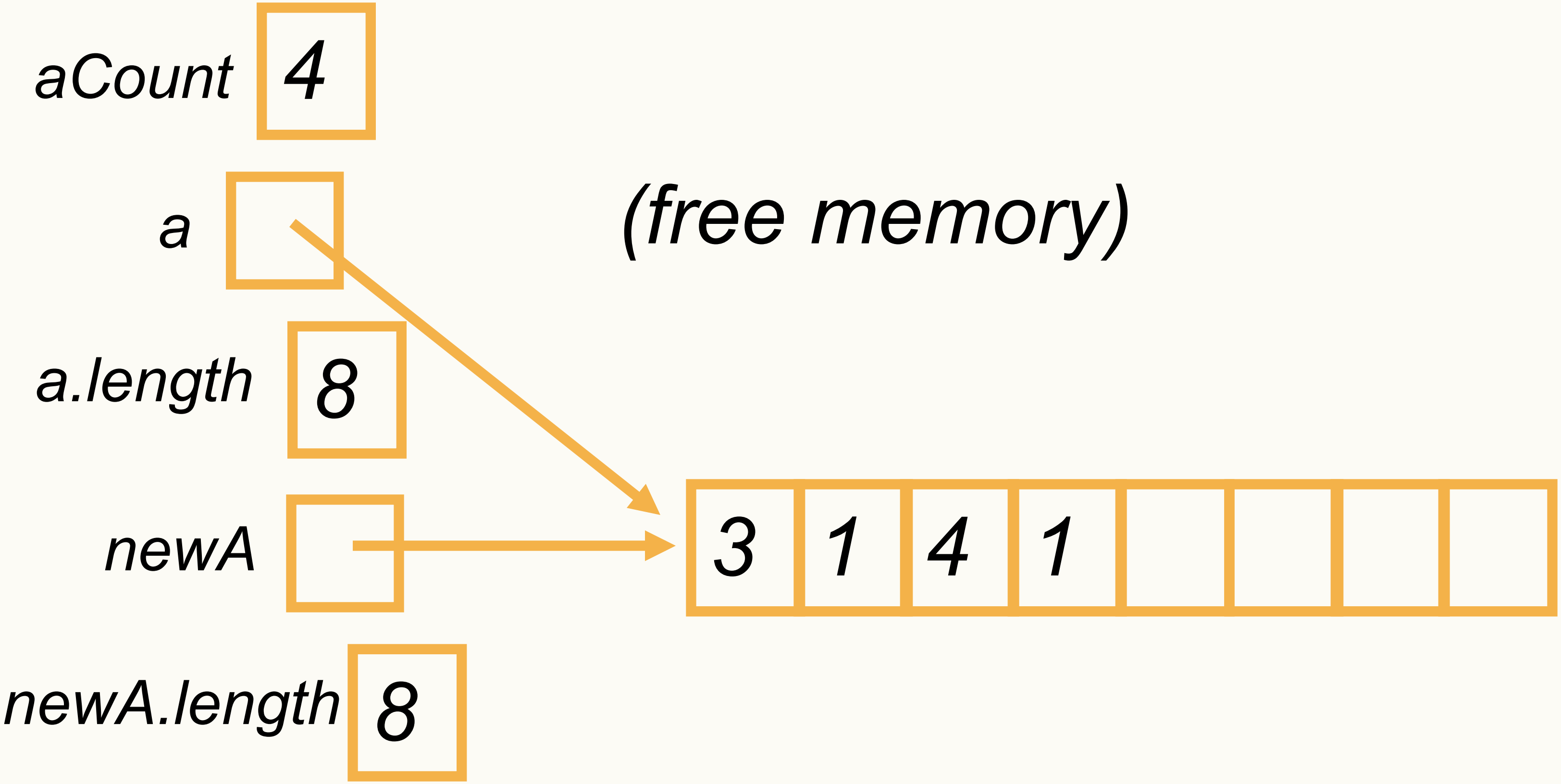
a → 3 1 4 1

a.length 4

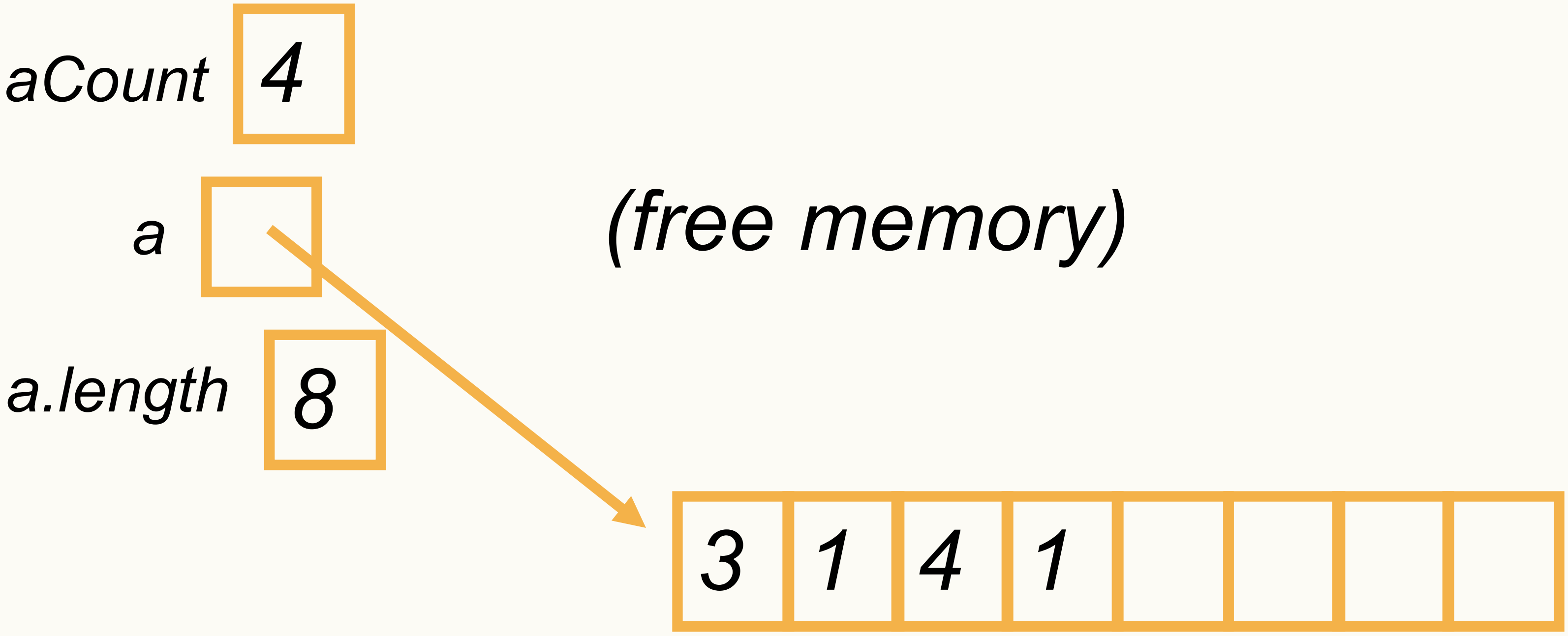
newA → 3 1 4 1

newA.length 8

Making Your Own ArrayList



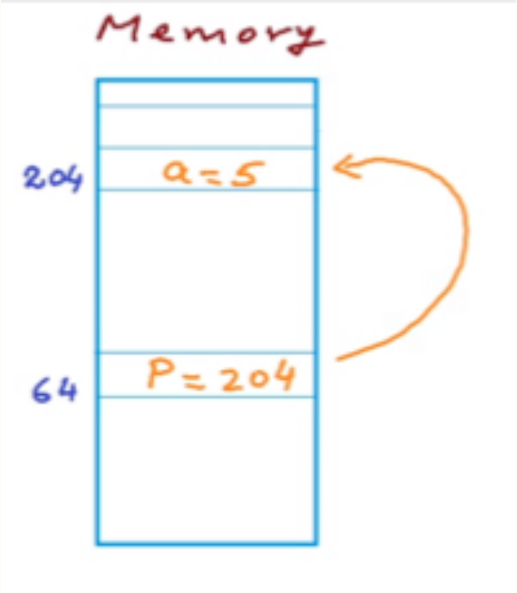
Making Your Own ArrayList



Addresses, &, and pointers

- A pointer is a data type that contains the address of the object in memory, but it is *not* the object itself.

```
int a = 5;
int *p = &a;
```



- In this example p is a pointer, which is storing the address of a.

Addresses, &, and pointers (cont)

- We declare a pointer to an object in the following way:

```
          dataType *identifier;
```

- For example, to declare a pointer to an integer, we can do the following:

```
int *intPtr;   or   int* intPtr;   or   int * intPtr;
```

- **Warning:** The declaration:

```
int *var1, var2;
```

… declares `var1` to be a *pointer* to an integer, but `var2` to be an *integer*! To declare both as pointers, do the following, or just do one per line:

```
int *var1, *var2;    Or     int* var1;   int* var2;
```

Department of Computer Science Undergraduate Events

More details @ https://my.cs.ubc.ca/students/development/events

Simba Technologies Tech Talk: Big Data: Volume, Variety & Velocity

Wed., Jan 7
5:30 pm, DMP 110

Deloitte Career Talk: What is IT Consulting

Wed., Jan 14
5:30 pm, DMP 110

CS Speed Mentoring & Townhall Event

Thurs., Jan 15
5:30 pm
Rm X860, ICICS/CS Bldg.

CPSC 221 Administrative Notes

- Lab1: Jan 12 – Jan 16
  - Posted on the course website
  - If you still haven't registered for a lab section, please do so as soon as possible

- Office hours to help you setup C++ on your laptops Friday Jan 9
  - Lynsey: Mac, 11-12, DLC
  - Kai Di: Windows, 11-1, DLC
  - Daniel: Mac, 1-3, DLC

So, Where Were We?

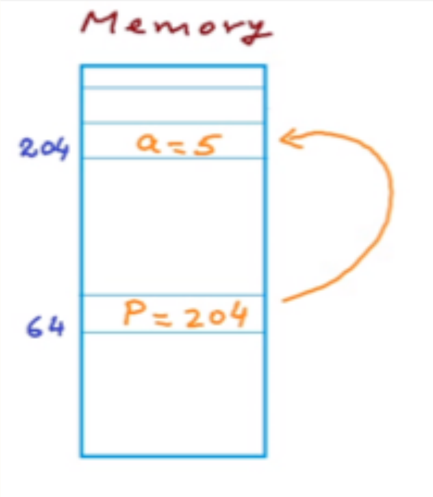
- We did a fun example on Fibonacci to show you performance matters!

- We talk about arrays
  - Arrays have a fixed size
  - Java ArrayLists are doing things behind the scenes to make things nicer for you to program…

Addresses, &, and pointers

- Now that p is pointing to a, how do we reference the object pointed to by p?

```
int a = 5;
int *p = &a;
```



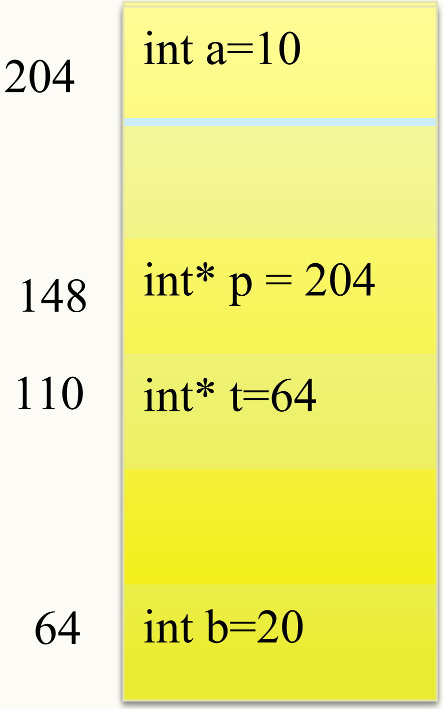
- This is achieved using the * (dereferencing) operator

```
std::cout << p << std::endl; // 204
std::cout << *p << std::endl; // 5
```

- **IMPORTANT** – The two different stars
  – int *p or int* p → declares an integer pointer p
  – *p = a → uses the * operator to dereference p

Clicker Question

```
int a = 10;
int b = 20;
int* p = &a;
int* t = &b;
```



204

int a=10

148

int* p = 204

110

int* t=64

64

int b=20

After performing `p = &b`

A: The value of `p` changes

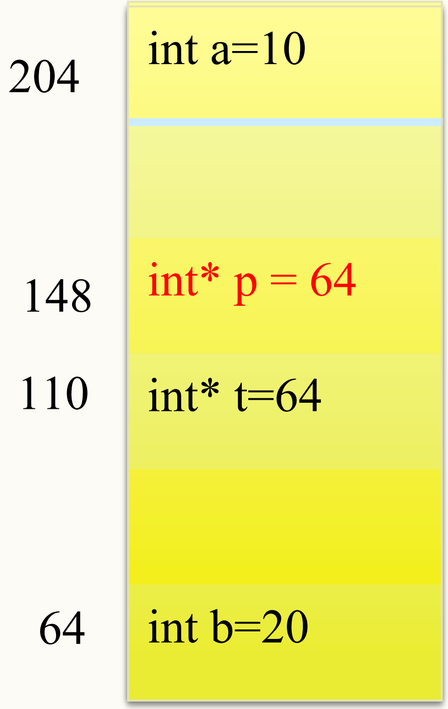
B: The value of `*p` changes

C: both A and B

D: none of the above

Clicker Question (answered)

```
int a = 10;
int b = 20;
int* p = &a;
int* t = &b;
```



204 int a=10

148 int* p = 64

110 int* t=64

64 int b=20

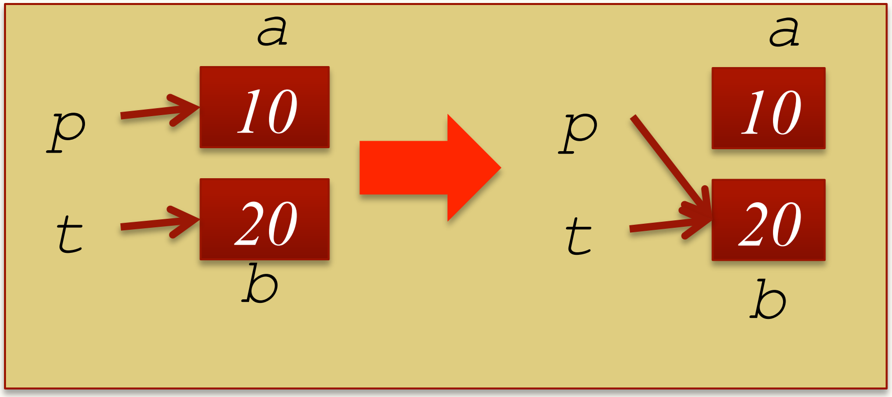
After performing `p = &b`

A: The value of `p` changes

B: The value of `*p` changes

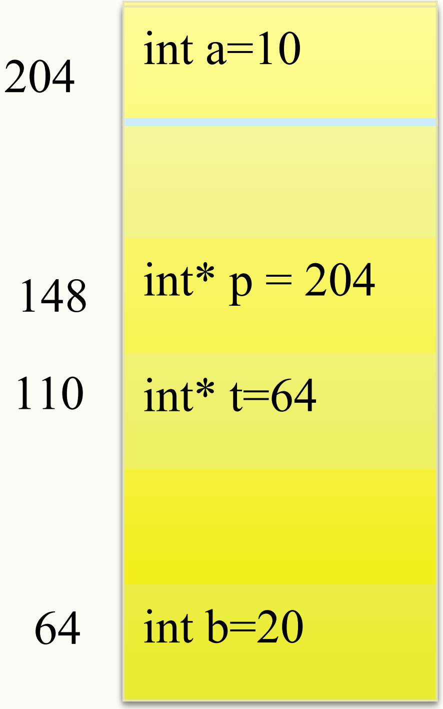
C: both A and B

D: none of the above



Clicker Question

```
int a = 10;
int b = 20;
int* p = &a;
int* t = &b;
```



`*p=b` and `p=&b`

A: Are literally the same;

B: both change the value of *p to 20

C: both end up making p be equal to t

D: Both B and C

Clicker Question (answered)

```
int a = 10;
int b = 20;
int* p = &a;
int* t = &b;
```

`*p=b` and `p=&b`

A: Are literally the same;

B: both change the value of *p to 20

C: both end up making p be equal to t

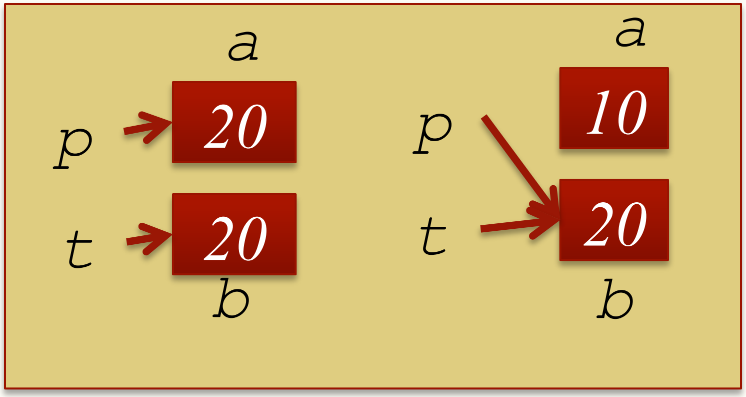
D: Both B and C

*p=b

204	int a=20
148	int* p = 204
110	int* t=64
64	int b=20

p=&b

204	int a=10
148	int* p = 64
110	int* t=64
64	int b=20



Records (Structures)

- A structure is declared using the keyword struct followed by a structure name. All the variables of a structure are declared within the structure. A structure type is defined by using the given syntax.

```
struct Employee{
     int          empNum;
     string       name;
     double       salary;
} ;
```

- The structure definition does not allocate any memory. It just gives a template that conveys to the C++ compiler how the structure is laid out in memory and gives details of the member names.

Records (Structures)

- Memory is allocated for the structure when we declare a variable of the structure. For example, we can define a variable of an employee by writing

```
Employee  boss1;
```

- Initializing a structure means assigning some constants to the members of the structure.

```
Employee    former_boss = {5000, "Derek", 99250.75};
```

- Each member of a structure can be used just like a normal variable, but its name will be a bit longer. A structure member variable is generally accessed using the '.' (dot operator).

```
Employee new_boss;
new_boss.empNum = 1000;
new_boss.name = "Ralph";
new_boss.salary = 125750.99;
```

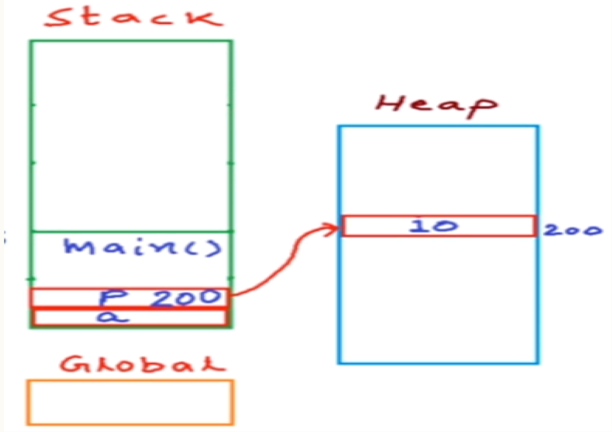
Dynamic Memory Allocation

- When our program runs, we can request extra space on-the-fly (i.e., when we need it—even large amounts!) from the memory heap.
  - We'll use two functions to handle our request for memory (called "allocation") from the heap, and our return of that memory (when we don't need it anymore—called "deallocation"):

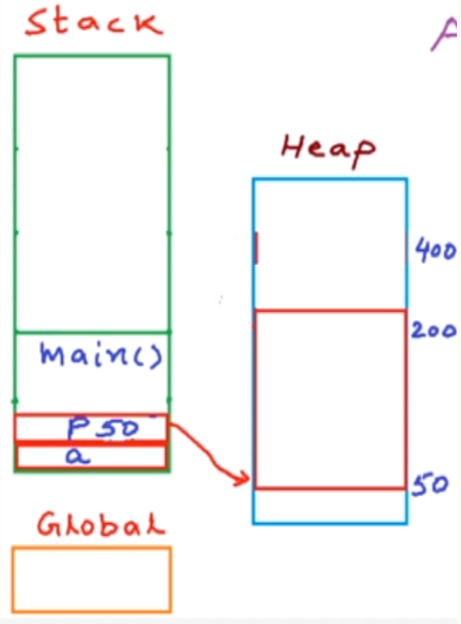
```
int* p = new int(3);
delete p;
```

Heap example

```
int main() {
 int a; /* on stack */
 int* p = new int; /* on heap */
 *p = 10;
}
```



```
int main() {
 int a; /* on stack */
 int* p = new int[20]; /* on heap */
}
```



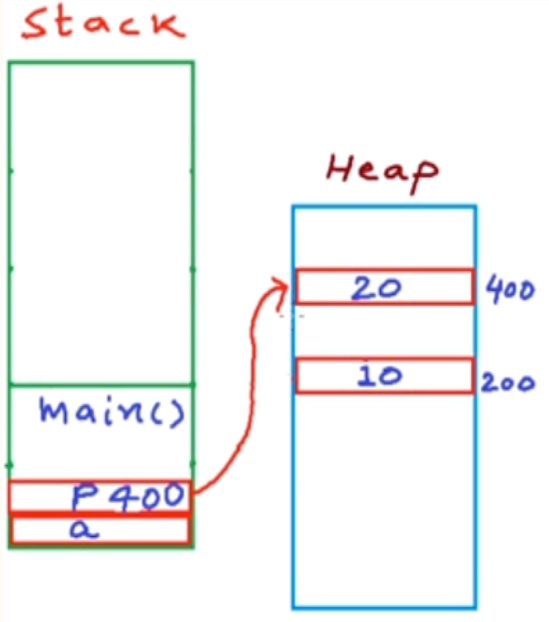
Example taken from here

Memory Leaks

- Keep track of the memory you allocate in a program; otherwise, you won't be able to reference it again! (or free it!)

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int a;
    int* p = new int;
    *p =10;
    p = new int;
    *p = 20;
    …
}
```

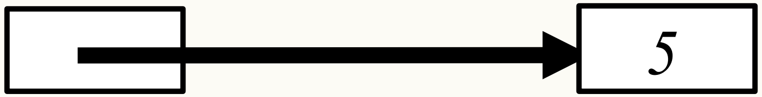


Memory Leaks

```
int main(void)
{
  int* getMemory;
  while (1){   /* endless loop */
    getMemory = new int[100000000000];
    if (getMemory == NULL){
       std::cout<< "no more memory available"<<std::endl;
       return -1;
    }
  std::cout<< "got memory at "<< getMemory<<std::endl;
  }
 return 0;
}
```

Dangling pointer

- When we’re done with the object we **delete** it, which reclaims the memory



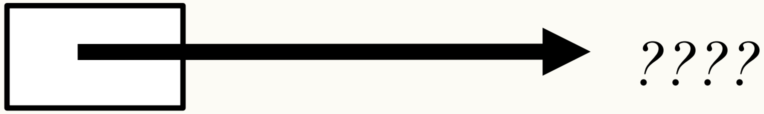
`delete p;`

```
int* p = new int(5);
delete p;
std::cout <<*p;// 5 is printed
```

What actually happens is that section of memory is marked as OK to overwrite, but it’s still there in memory, at least until its overwritten.

Dangling pointer

- If we don't change our pointer so that it no longer refers to the deleted object, it is now referring to deallocated memory.
- The system may later re-allocate that memory and the pointer will behave unpredictably when dereferenced.



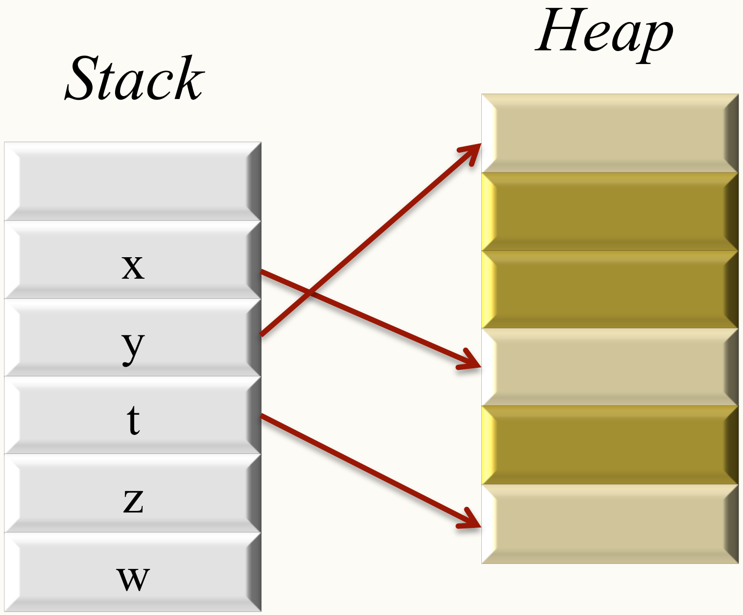
- Such a pointer is called a **dangling pointer** and leads to bugs that can be subtle and brutally difficult to find.

Example

- What is printed to the screen, and clearly identify any memory leaks and dangling pointers.

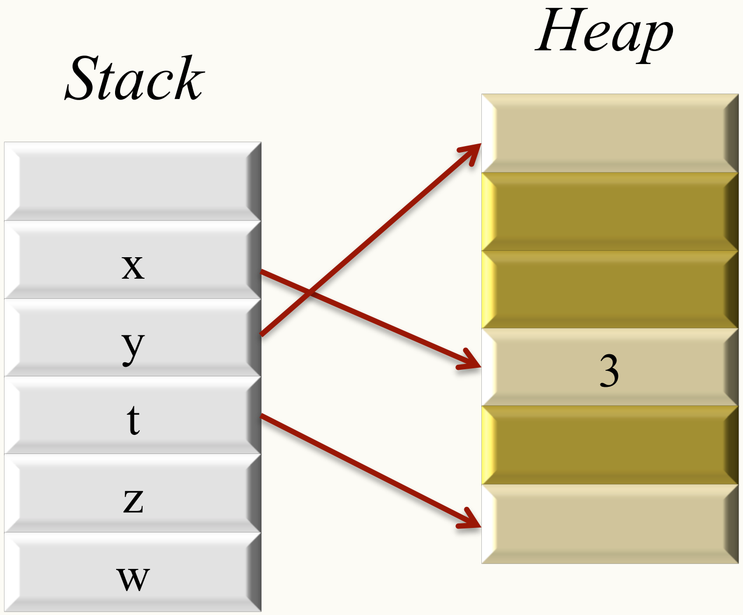
```
int* x = new int;
int* y = new int;
int* t = new int;
int z;
int w;

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
delete x;
*t = 2;
y = &z;
x = y;
delete t;
std::cout << *x << " "<< *y << " "<< z <<" "<< w ;
```



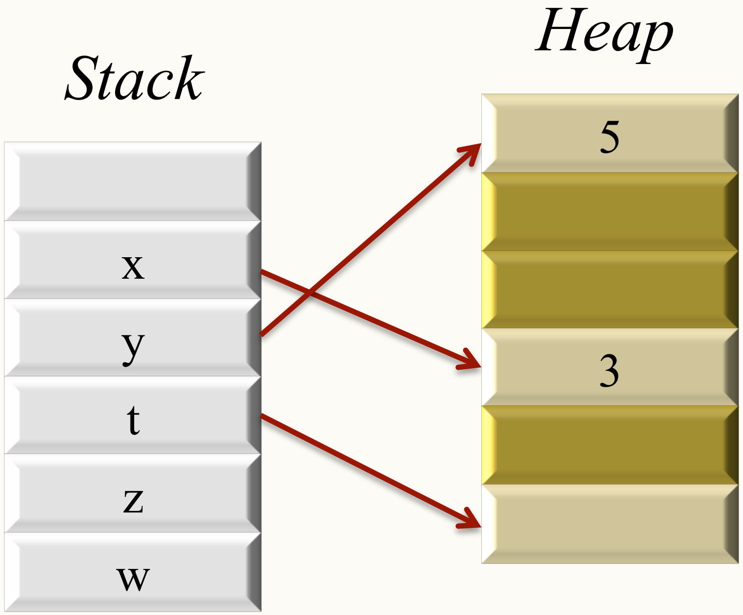
```
int* x = new int;
int* y = new int;
int* t = new int;
int z;
int w;

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
delete x;
*t = 2;
y = &z;
x = y;
delete t;
std::cout << *x << " "<< *y << " "<< z <<" "<< w ;
```



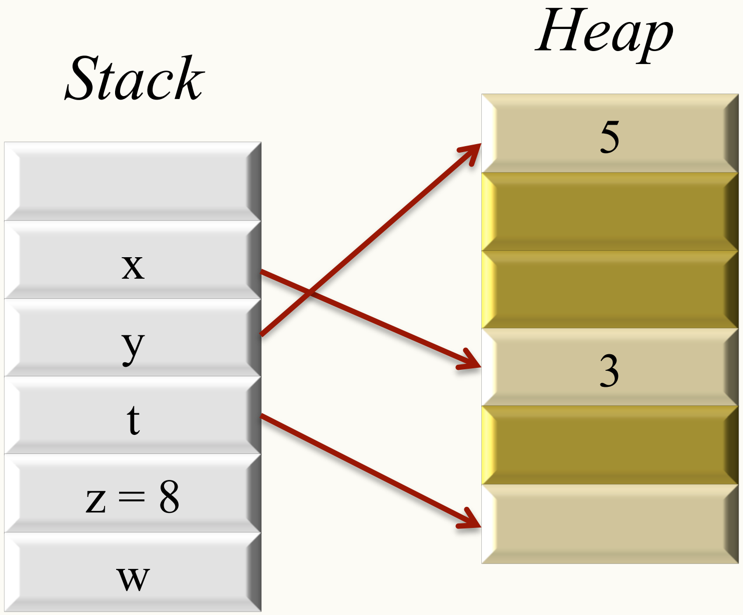
```
int* x = new int;
int* y = new int;
int* t = new int;
int z;
int w;

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
delete x;
*t = 2;
y = &z;
x = y;
delete t;
std::cout << *x << " "<< *y << " "<< z <<" "<< w ;
```



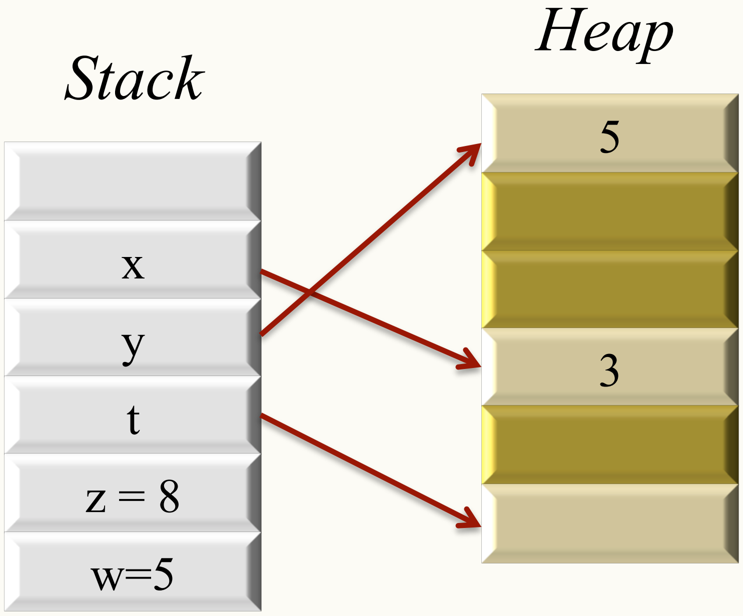
```
int* x = new int;
int* y = new int;
int* t = new int;
int z;
int w;

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
delete x;
*t = 2;
y = &z;
x = y;
delete t;
std::cout << *x << " "<< *y << " "<< z <<" "<< w ;
```



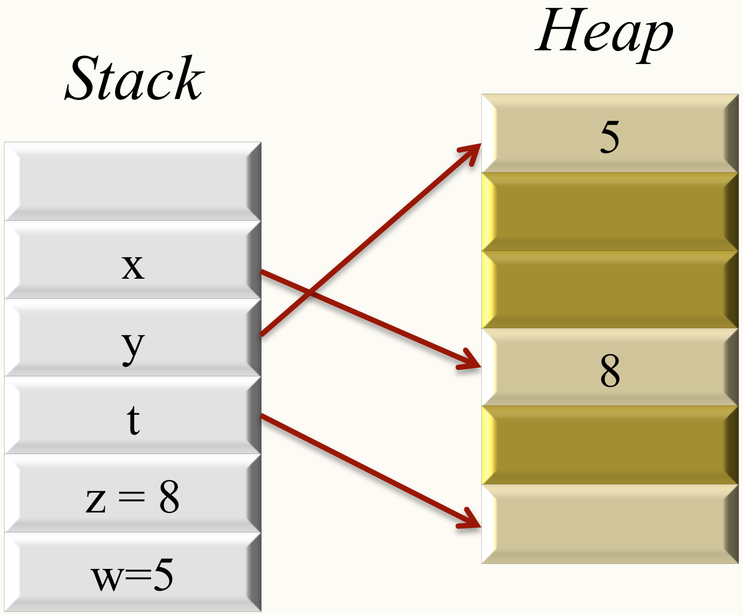
```
int* x = new int;
int* y = new int;
int* t = new int;
int z;
int w;

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
delete x;
*t = 2;
y = &z;
x = y;
delete t;
std::cout << *x << " "<< *y << " "<< z <<" "<< w ;
```



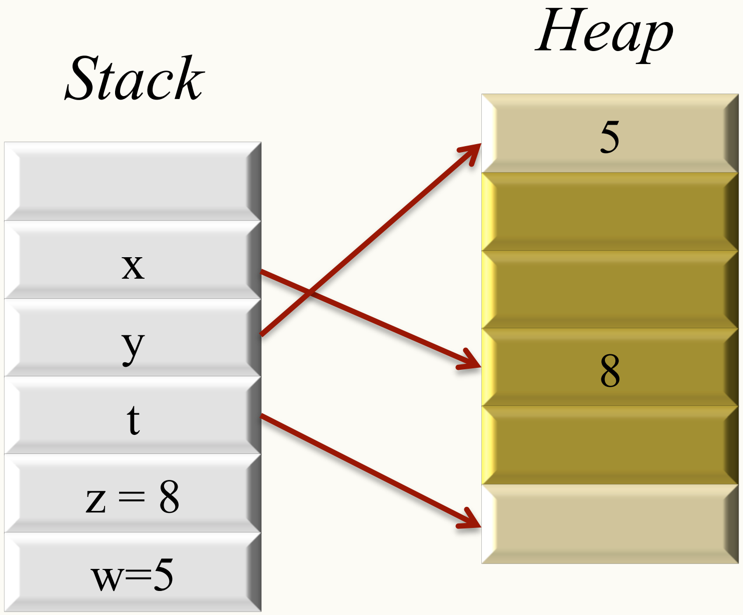
```
int* x = new int;
int* y = new int;
int* t = new int;
int z;
int w;

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
delete x;
*t = 2;
y = &z;
x = y;
delete t;
std::cout << *x << " "<< *y << " "<< z <<" "<< w ;
```



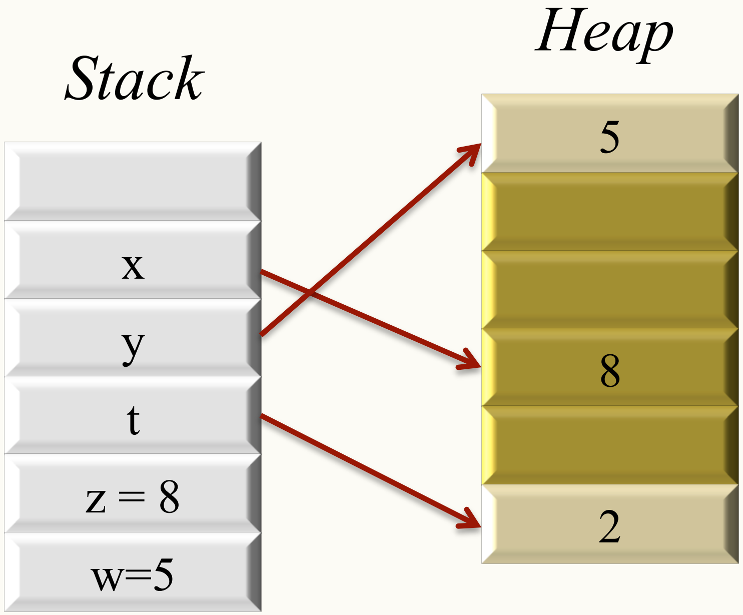
```
int* x = new int;
int* y = new int;
int* t = new int;
int z;
int w;

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
delete x;
*t = 2;
y = &z;
x = y;
delete t;
std::cout << *x << " "<< *y << " "<< z <<" "<< w ;
```



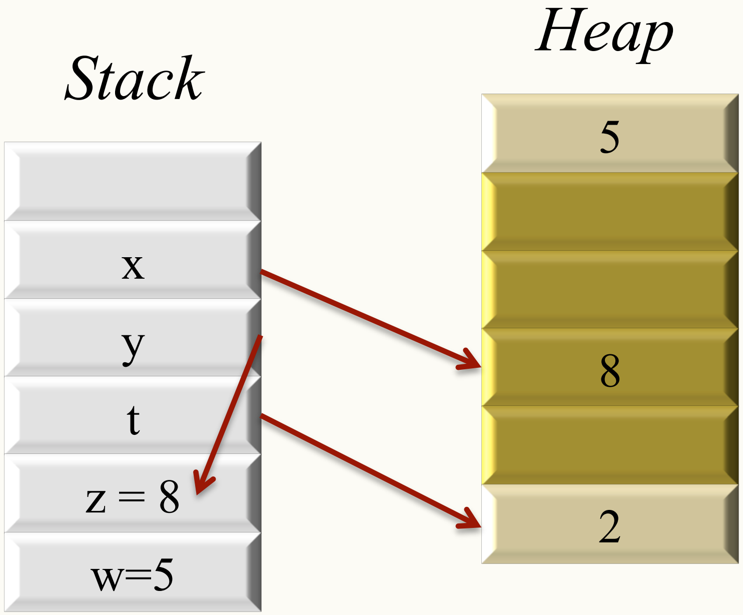
```
int* x = new int;
int* y = new int;
int* t = new int;
int z;
int w;

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
delete x;
*t = 2;
y = &z;
x = y;
delete t;
std::cout << *x << " "<< *y << " "<< z <<" "<< w ;
```



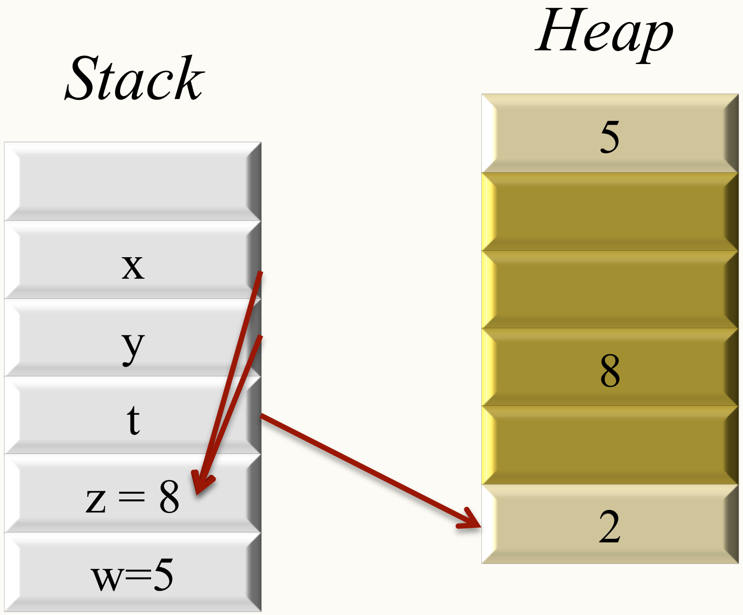
```
int* x = new int;
int* y = new int;
int* t = new int;
int z;
int w;

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
delete x;
*t = 2;
y = &z;
x = y;
delete t;
std::cout << *x << " "<< *y << " "<< z <<" "<< w ;
```



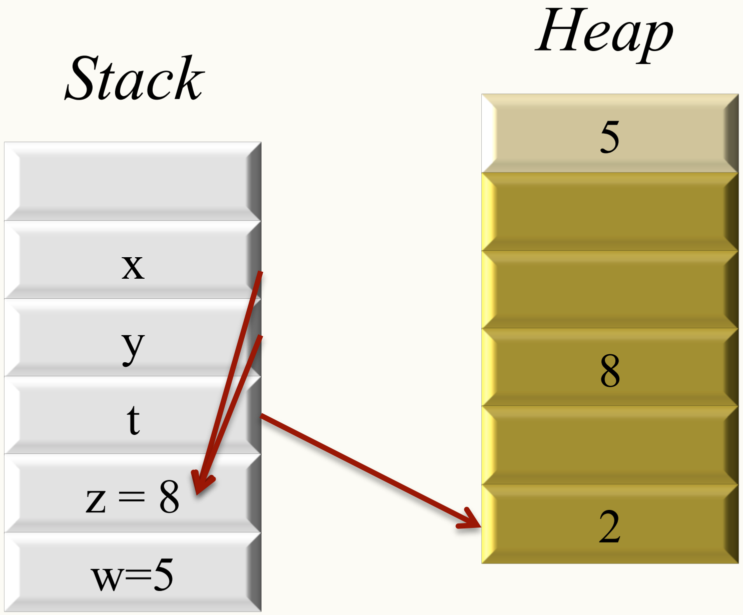
```
int* x = new int;
int* y = new int;
int* t = new int;
int z;
int w;

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
delete x;
*t = 2;
y = &z;
x = y;
delete t;
std::cout << *x << " "<< *y << " "<< z <<" "<< w ;
```



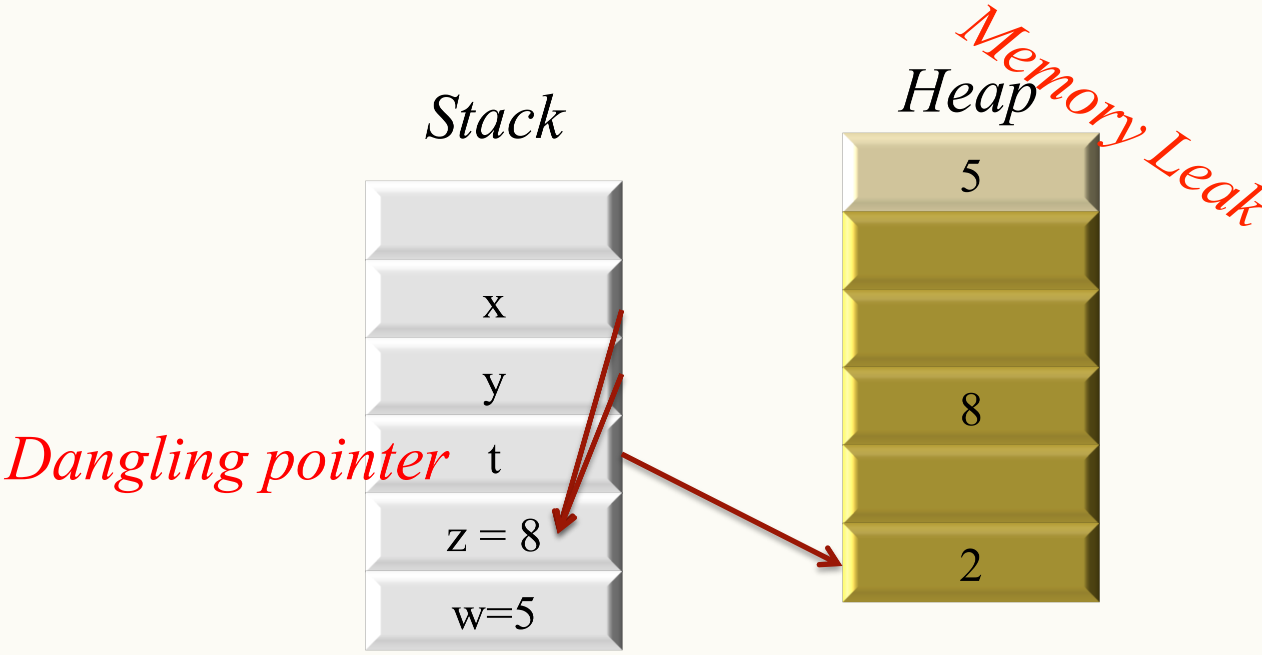
```
int* x = new int;
int* y = new int;
int* t = new int;
int z;
int w;

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
delete x;
*t = 2;
y = &z;
x = y;
delete t;
std::cout << *x << " "<< *y << " "<< z <<" "<< w ;
```



```
int* x = new int;
int* y = new int;
int* t = new int;
int z;
int w;

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
delete x;
*t = 2;
y = &z;
x = y;
delete t;
std::cout << *x << " "<< *y << " "<< z <<" "<< w ;
```

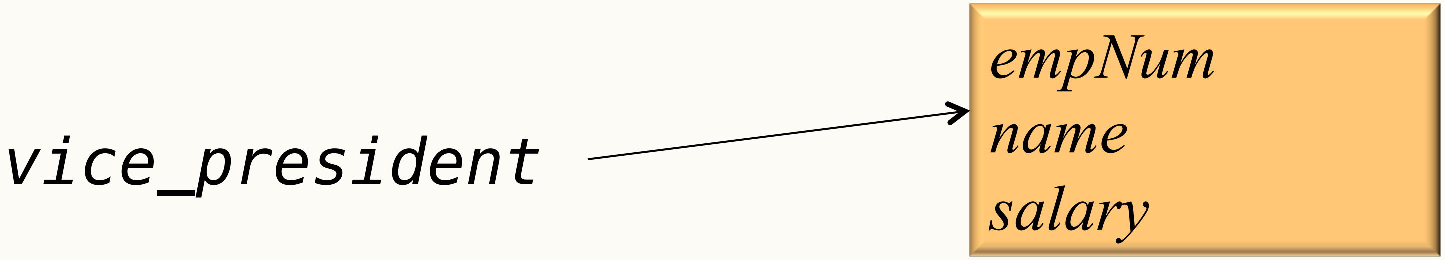


8 8 8 5

Declaring a struct through pointers

- Like in other cases, a pointer to a structure is never itself a structure, but merely a variable that holds the address of a structure. The syntax to declare a pointer to a structure can be given as

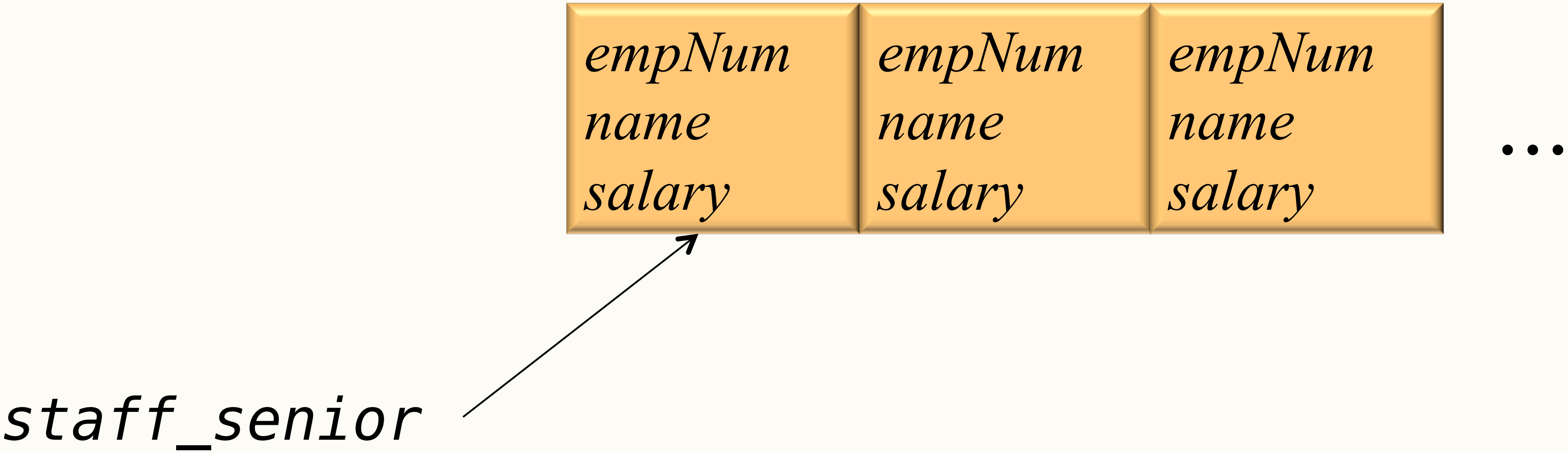
```
 Employee *vice_president = new Employee();
(*vice_president).empNum = 10000.00; /* one way */
vice_president->empNum = 1; /* another way */
vice_president->salary = 105000.00;
```



Declaring an array of struct through pointers

```
Employee *  staff_senior = new Employee[50];
int i = 4;
staff_senior[i].empNum = 100 + i;

/* another way of accessing the data,
   via pointer arithmetic (parentheses needed) */
(staff_senior + i)->salary = 80000;
(*(staff_senior + i)).salary *= 1.0 /* 5% pay increase*/
```



empNum
name
salary

empNum
name
salary

empNum
name
salary

…

staff_senior

Learning goals revisited

- Become familiar with addresses and pointers in C++.
- Describe the relationship between addresses and pointers.
- Become familiar with arrays in C++.
- Define and use records (structs) in an implementation with dynamic memory allocation.
- Demonstrate how dynamic memory management is handled in C++ (e.g., allocation, deallocation, memory heap, run-time stack).
- Gain experience with pointers in C++ and their tradeoffs and risks (dangling pointers, memory leaks).