CPSC 259: Data Structures and Algorithms for Electrical Engineers

Pointers, Arrays, and Dynamic Memory Allocation

Textbook References:
(a) Etter: Chapter 6, pages 283-294, 308-316
(b) Thareja first edition: Chapter 3, pages 97-109
(c) Thareja second edition: 1.11, 3.7-3.8

Learning Goals for This Unit

- Describe the purpose of a pointer data type.
- Describe the relationship between addresses and pointers.
- Explain the difference in parameter passing for call-by-value versus call-by-reference.
- Explain the purpose of dynamic memory allocation. Give examples of where dynamic memory allocation is particularly useful, and examples of where it is not (i.e., where static allocation (at compile time) is better).
- Gain experience with pointers in C and describe their tradeoffs and risks (e.g., dangling pointers, memory leaks).
- Demonstrate how dynamic memory management is handled in C (e.g., allocation and deallocation from the memory heap).

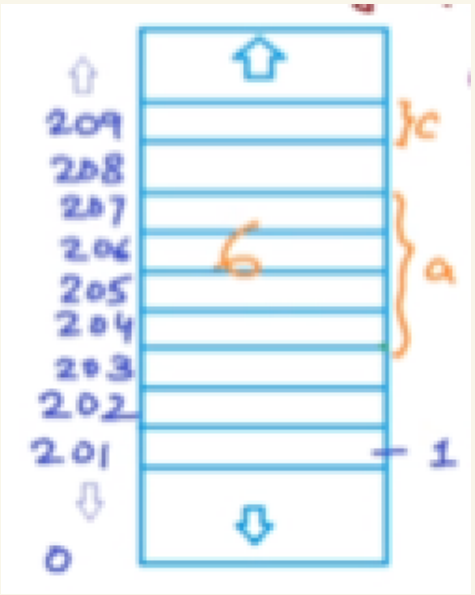
Addresses and Pointers

- When implementing data structures in C, we often need to consider *addresses* and *pointers*.
- You can think of addresses in memory in a way analogous to rooms in a hotel, or houses in a city. Each room or house has a specific location that is identified by its room number or street address. For example, if we want to put something in a specific house, then we need to know *which* house. The address tells us which house it is.
- Similarly, each storage location in memory (RAM) has an address associated with it. The address is the location in memory where a given variable or identifier stores its data.

Variable declaration

- Each byte of memory has a unique *address*.

```
int a;
char c;
a = 5;
a++;
```



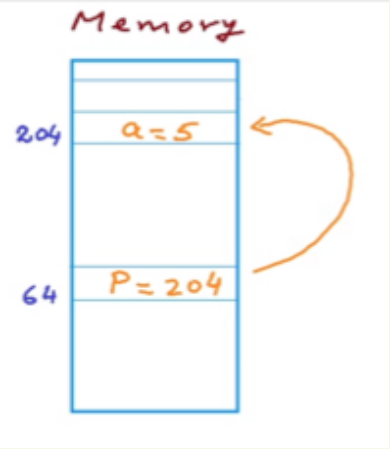
- `a` is an identifier that references a block or chunk of memory that is used to store the integer *value* 6. Rather than referencing this block using a hard-to-remember numeric *address*, we use the identifier instead; this makes programming much easier.
- At *compile time*, the compiler knows how much memory to allocate to `a` (i.e., 4 bytes for an integer). Four bytes can hold integers in the range of approximately +/− 2 billion. Why?

4 bytes = 32 bits
The first bit is the sign; the other 31 bits allow for 2^{31}= 2,147, 483, 648

Addresses, &, and pointers

- You're already familiar with addresses from the `scanf` statement. Recall that the `scanf` function required us to provide the *address* of a location, rather than just the identifier. The address was identified using the "address of" operator, which is &.
  - `scanf("%d", &a);`

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

Addresses, &, and pointers (cont)

- Consider the following code segment:

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

- How do we get `p` to contain the address of (i.e., "to point to") `a`? This is achieved using the `&` (address-of) operator:

```
p = &a;
```

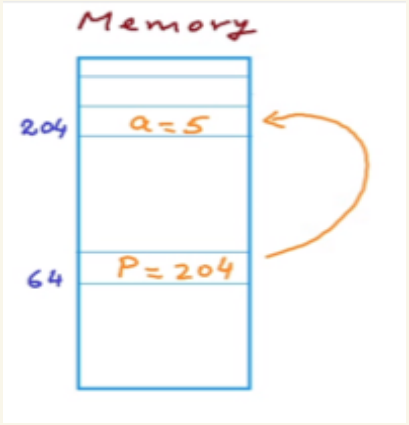
- Now that `p` is pointing to `a`, how do we reference the object pointed to by `p`? By using the `*` (dereferencing) operator:

```
printf("dereferencing p = %d \n",*p); // 5
```

IMPORTANT – The two different stars

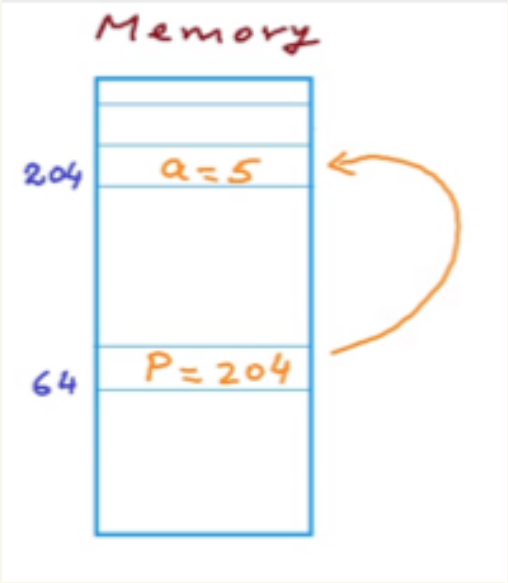
int* p → declares an integer pointer p

*p = a → uses the * operator to dereference p



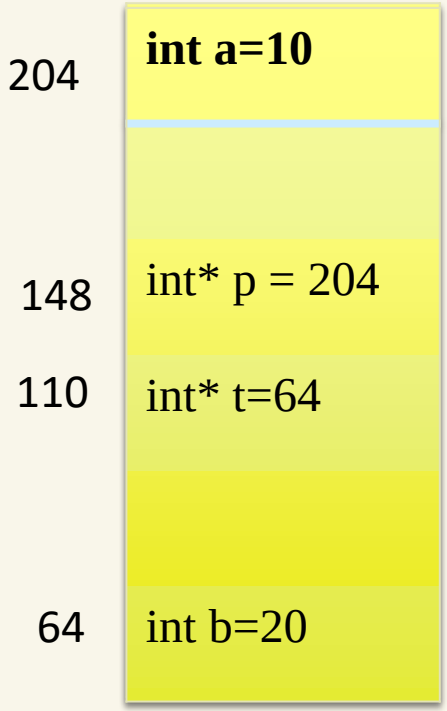
Addresses, &, and pointers example

```
 int a = 5;
int* p = &a;
printf("value of a = %d \n",a); // 5
printf("address of a = %p \n", &a); // 204
printf("value of p = %p \n", p); // 204
printf("dereferencing p = %d \n",*p); // 5
```



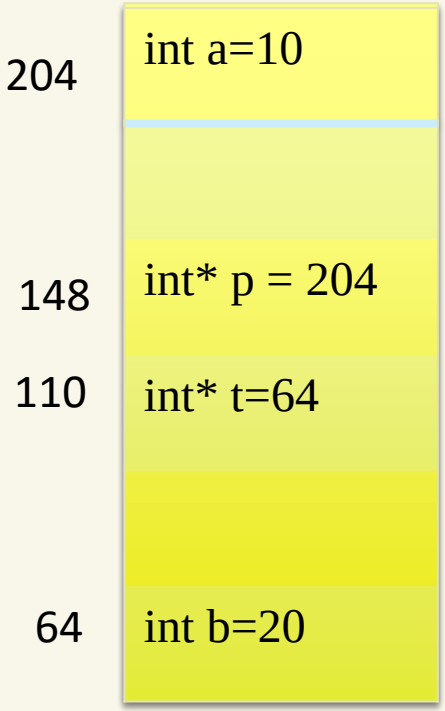
Pointers example

```
int a=10;
int b= 20;
int* p= &a;
int* t= &b ;
*p = 12 // changes the value
stored at 204 to 12
```



Clicker Question

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



After performing `a=b` the content of the memory at

A: 204 will change

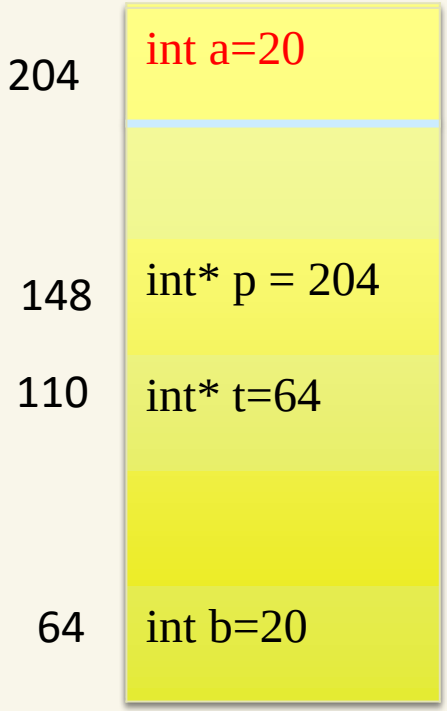
B: 148 will change

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;
```



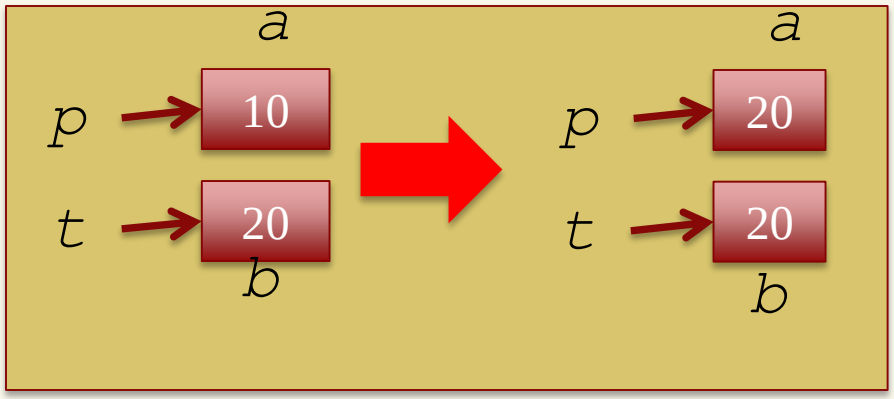
After performing `a=b` the content of the memory at

A: 204 will change

B: 148 will change

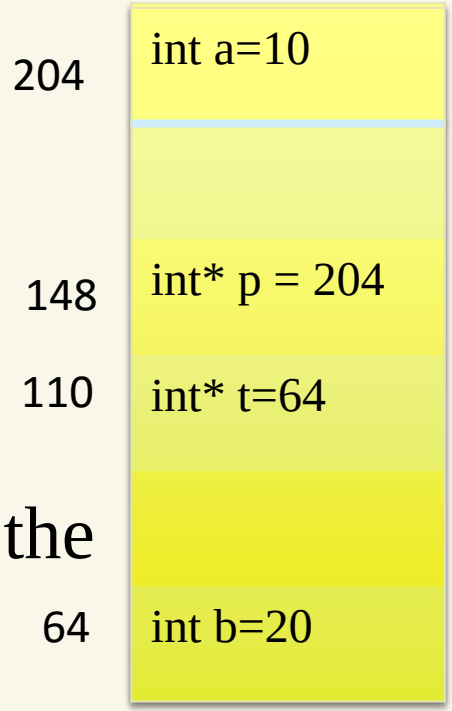
C: both A and B

D: none of the above



Clicker Question

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



After performing `*p = *t` the content of the memory at

A: 204 will change

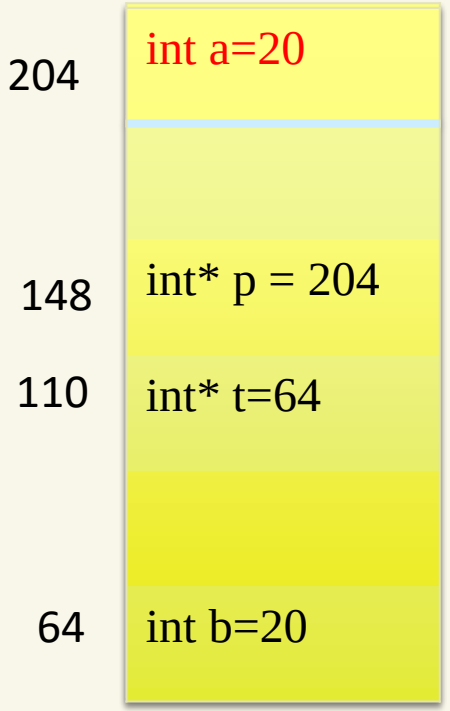
B: 148 will change

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;
```



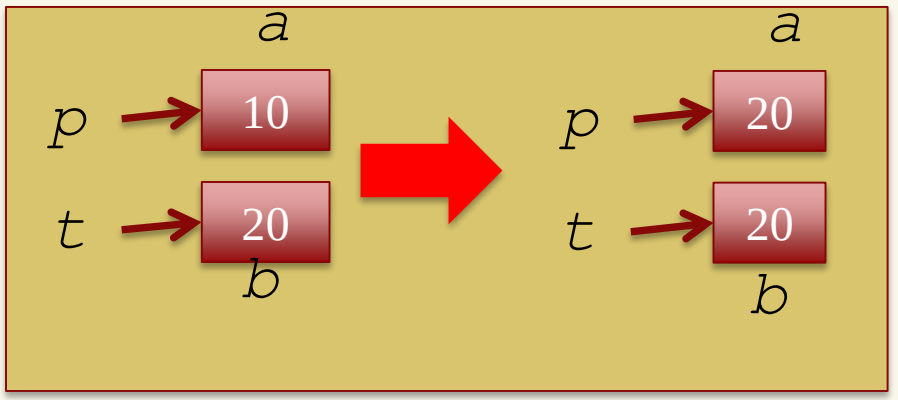
After performing `*p = *t` the content of the memory at

A: 204 will change

B: 148 will change

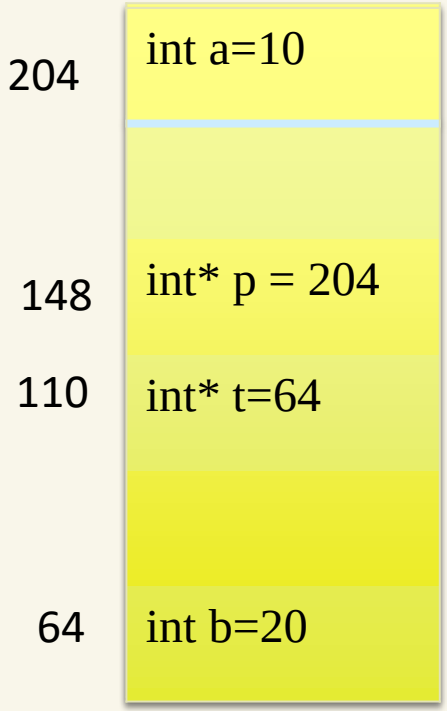
C: both A and B

D: none of the above



Clicker Question

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



After performing `p = &b`

A: The value of `p` changes

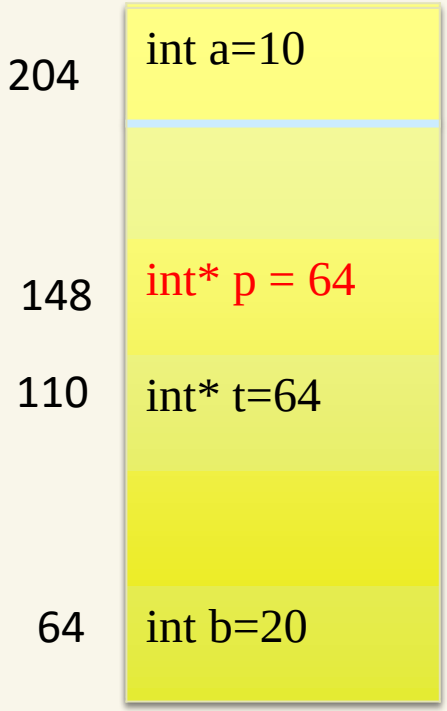
B: The value of `*p` changes

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;
```



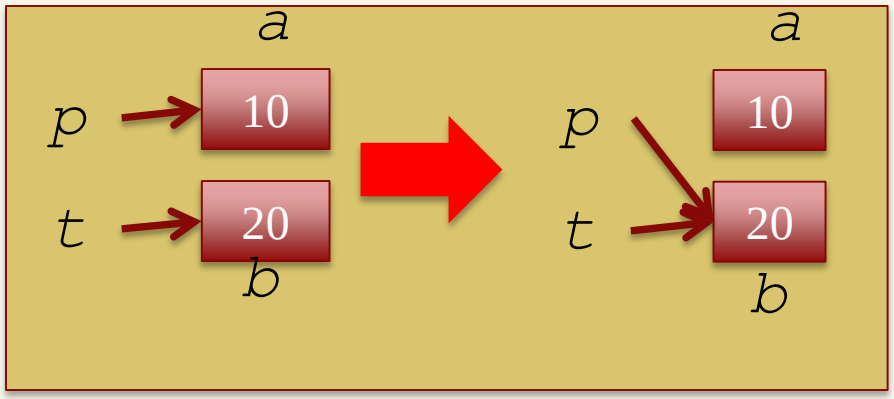
After performing `p = &b`

A: The value of `p` changes

B: The value of `*p` changes

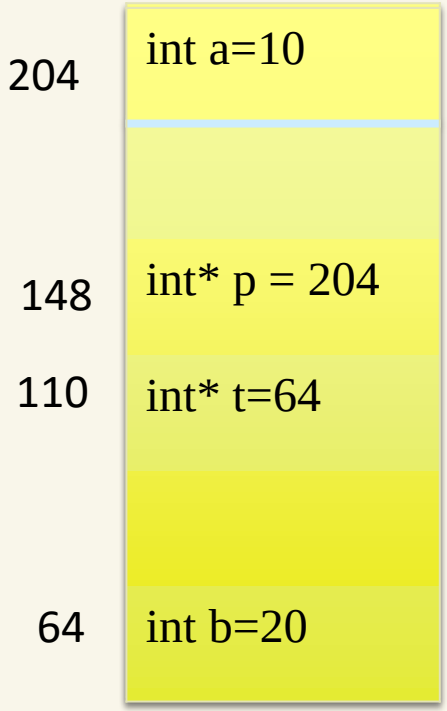
C: both A and B

D: none of the above



Clicker Question

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



After performing `p = &b;` *p=a

A: the value of `p` changes

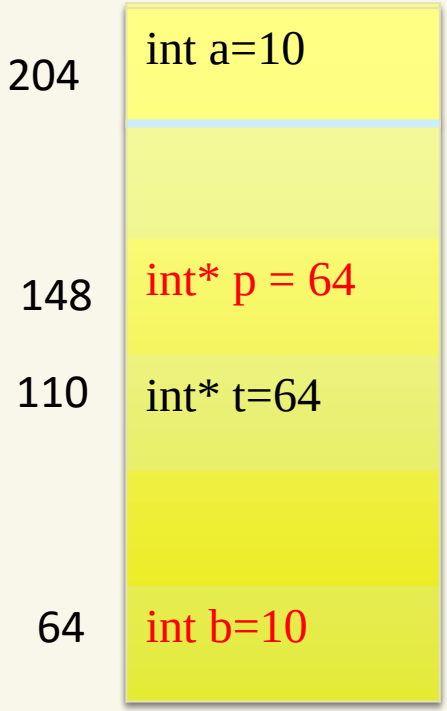
B: the value of `b` changes

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;
```



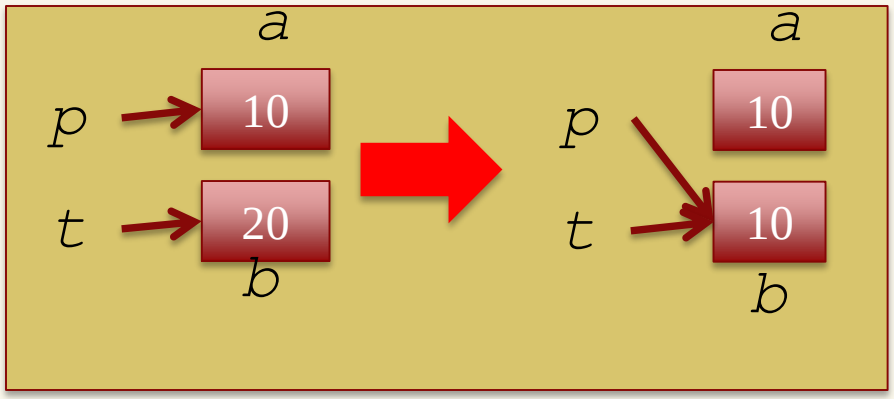
After performing `p = &b;` *p=a

A: the value of `p` changes

B: the value of `b` changes

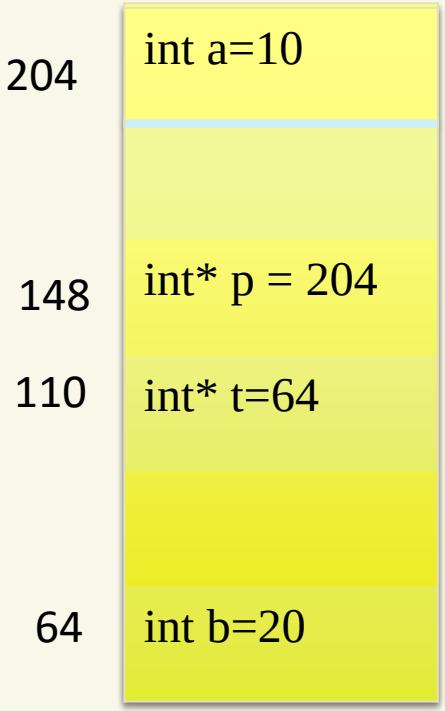
C: both A and B

D: none of the above



Clicker Question

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



After performing `p = t;`

A: the address of `a` cannot be retrieved

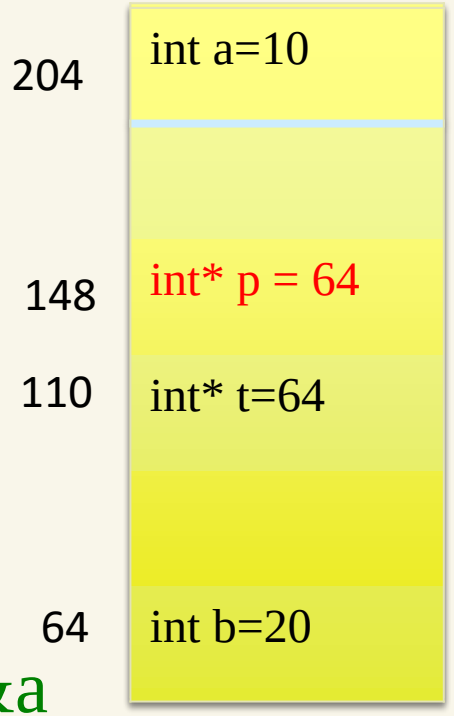
B: changing `*t` will change `*p`

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;
```



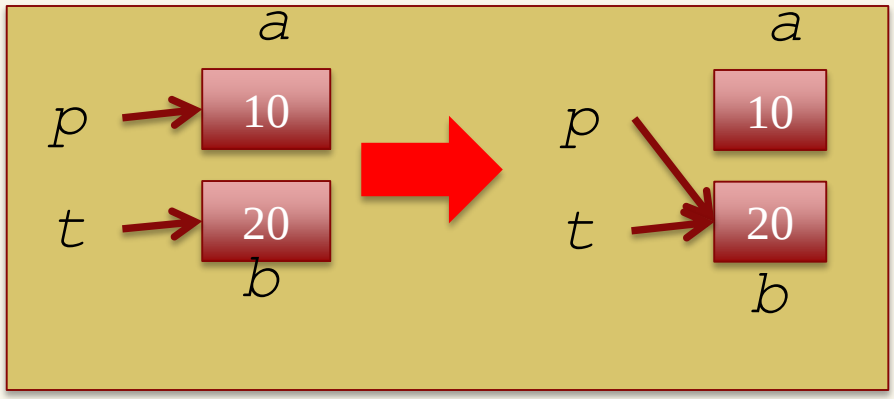
After performing `p = t;`

A: the address of `a` cannot be retrieved // &a

B: changing `*t` will change `*p`

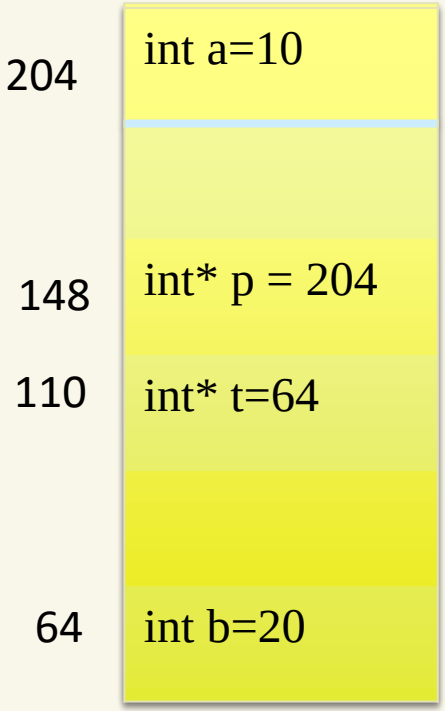
C: both A and B

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;
```

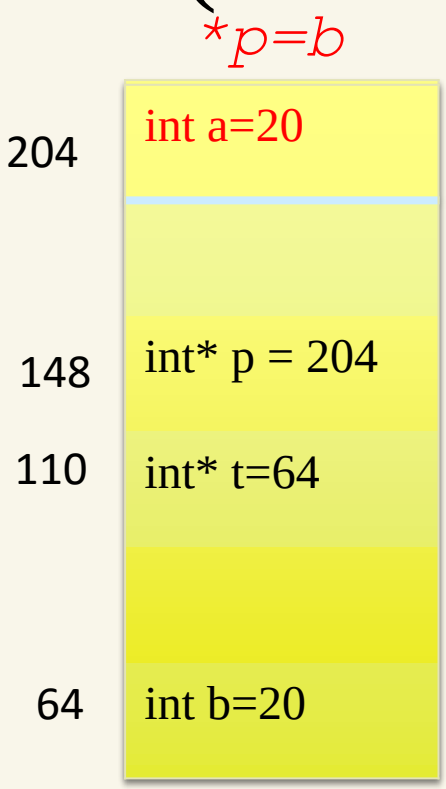
*p=b and p=&b

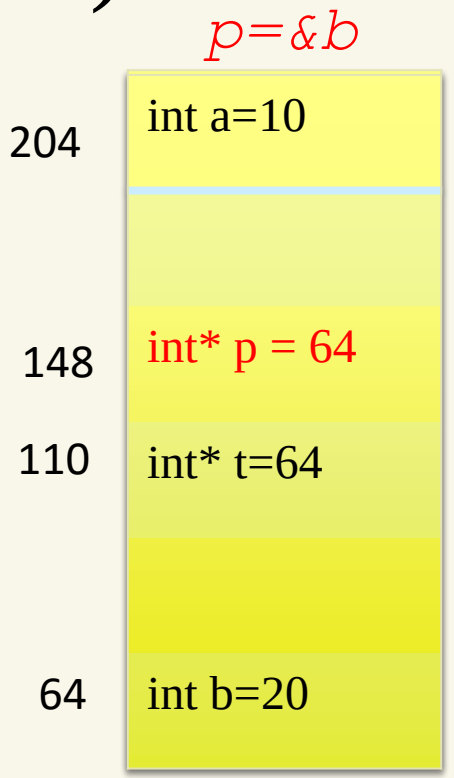
A: Are literally the same;

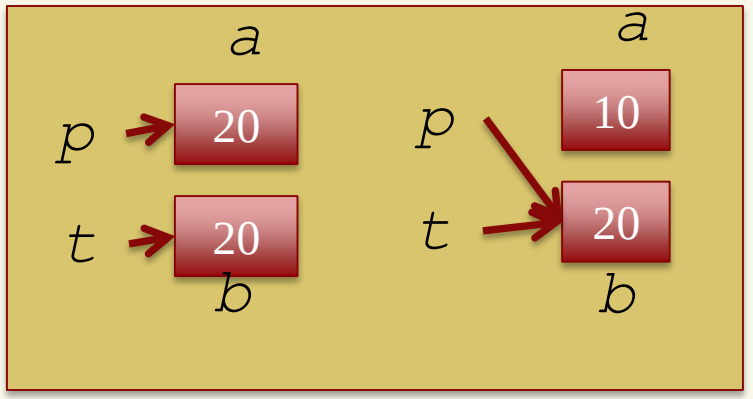
B: both change the value of *p to 20

C: both end up making p be equal to t

D: Both B and C







Question

```
int main()
{
    int a = 5;
    int* p = &a; // assume 0x7fff5fbff8cc

    printf("value of p = %p \n", p);
    printf("dereferencing p = %d \n",*p);
    p++;

    printf("value of p+1 = %p \n", p);
    printf("dereferencing p+1 = %d \n",*p);
}
```

What is the value of p and *p after p++ is executed?

`p+1 = 0x7fff5fbff8d0,  *(p+1)= garbage`

More on Strings later, but a quick example

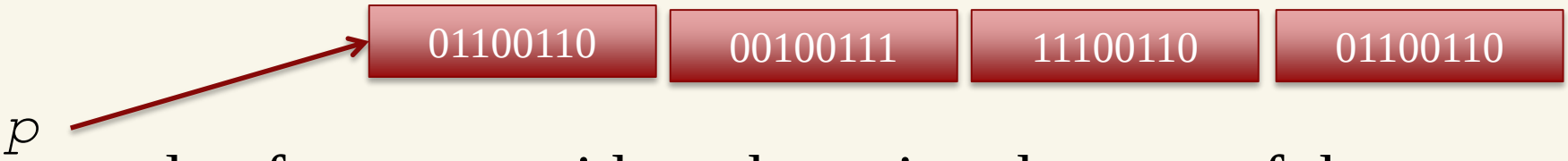
```
int main(){
    char* ch = "Hello world";
    printf("%s \n", ch); /* Hello world */
    printf("%c \n", ch[2]); /* l */
    printf("%c \n", *(ch+1)); /* e */
    return 0;
}
```

Pointer types

`int*` → points to an integer

`char*` → points to a char

Why don't we use a generic type for pointers?



Can we dereference p without knowing the type of the variable that it is pointing to?

int → 4 bytes

char → 1 byte

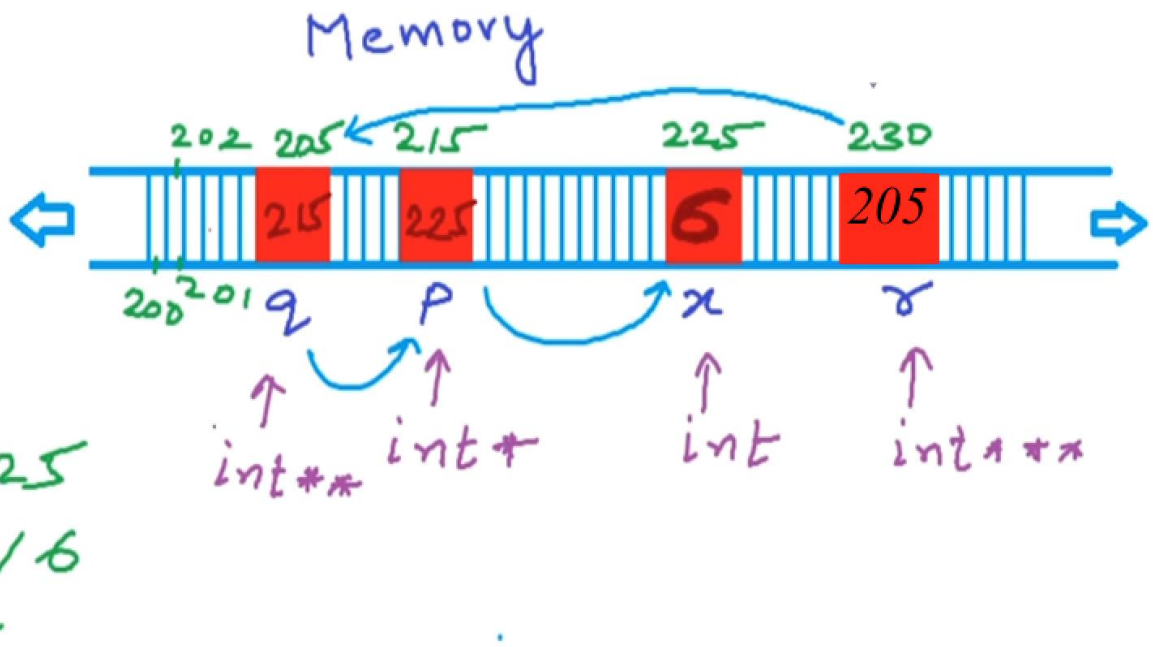
Generic pointers

- Generic pointers can be declared, but they need to be casted before they can be dereferenced.

```
int main()
{
    int x=10;
    char ch = 'A';
    void* gp;
    gp = &x;
    printf("\n integer value = %d", *(int*)gp); /* 10 */
    gp = &ch;
    printf("\n now points to %c \n", *(char*)gp);/* A */
    return 0;
}
```

Pointer to pointer

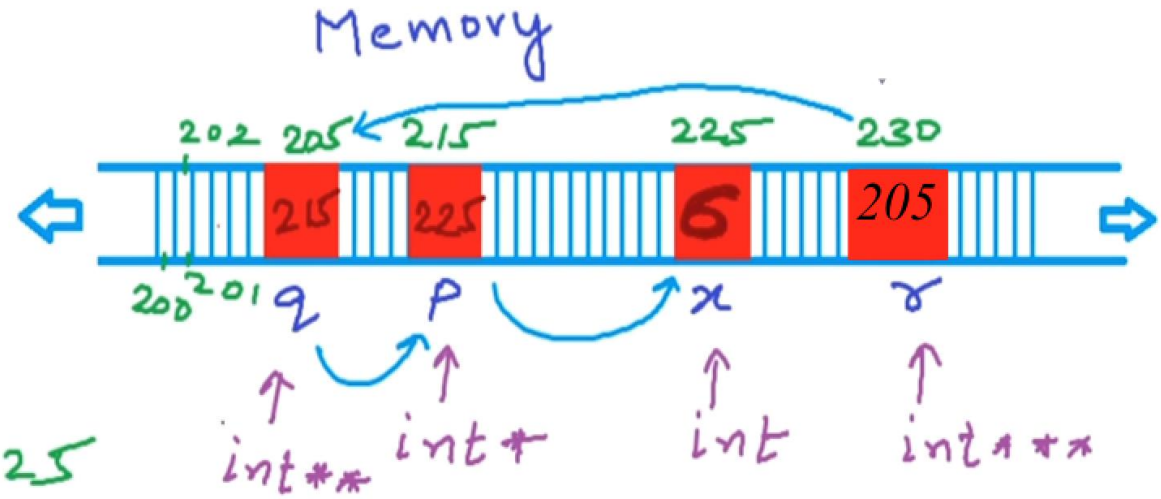
```c
#include<stdio.h>
int main()
{
    int x = 5;
    int* p = &x;
    *p = 6;
    int** q = &p;
    int*** r = &q;
    printf("%d\n",*p); // 6
    printf("%d\n",*q); // 225
    printf("%d\n",*(*q)); // 6
```



"You can keep adding levels of pointers until your brain explodes or the compiler melts - whichever happens sooner"

Clicker Question

```
#include<stdio.h>
int main()
{
    int x = 5;
    int* p = &x;
    *p = 6;
    int** q = &p;
    int*** r = &q;
    printf("%d\n",*p); // 6
    printf("%d\n",*q); // 225
    printf("%d\n",*(*q)); // 6
```



What would be printed to the screen after executing:

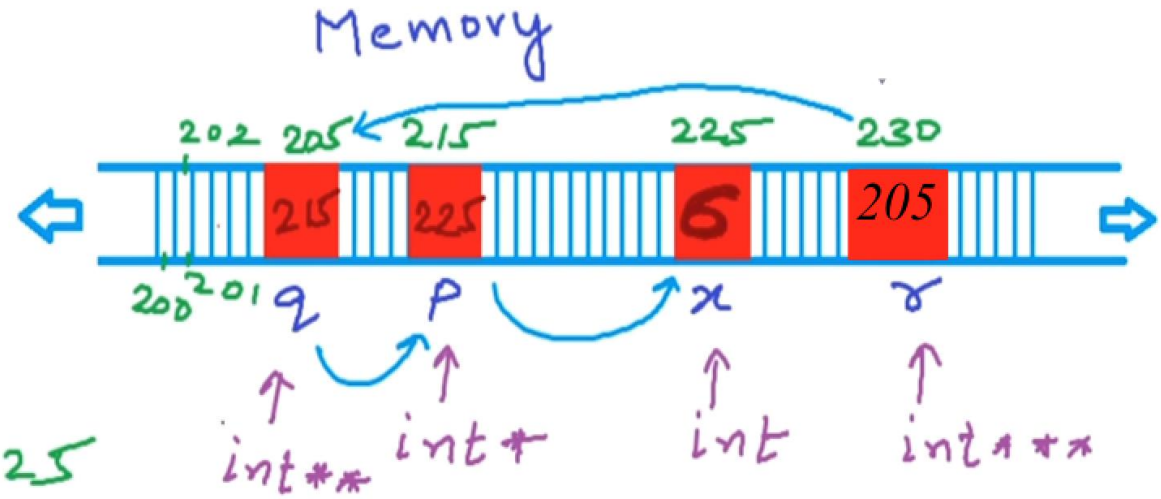
`printf("%d\n", *(*r))`?

A: 225 *B: 215*

C: 6 *D: None*

Clicker Question

```
#include<stdio.h>
int main()
{
    int x = 5;
    int* p = &x;
    *p = 6;
    int** q = &p;
    int*** r = &q;
    printf("%d\n",*p); // 6
    printf("%d\n",*q); // 225
    printf("%d\n",*(*q)); // 6
```



What would be printed to the screen after executing:

`printf("%d\n", *(*r))`?

A: 225 *B: 215*

C: 6 *D: None*

Parameter passing (call by value)

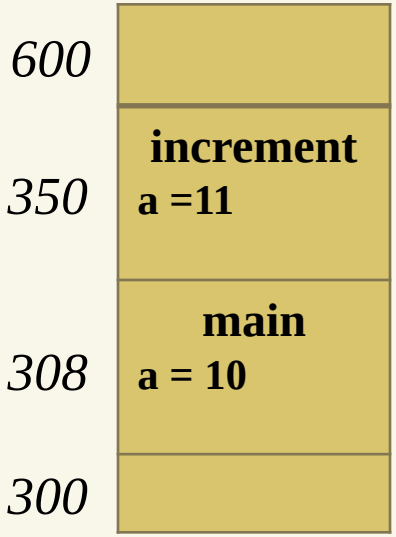
```
#include <stdio.h>
void increment(int a){
  a = a + 1;
}

int main(){
  int a;
  a = 10;
  increment(a);
  printf("a =  %d", a);
}
```

A: a=10

B: a=11

Parameter passing (call by value)

```
#include <stdio.h>
void increment(int a){
  a = a + 1;
}

int main(){
  int a;
  a = 10;
  increment(a);
  printf("a =  %d", a);
}
```

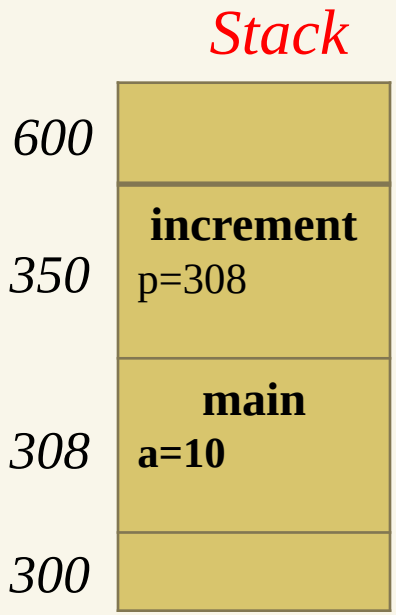


- a from main is mapped to a in increment: a → a
- If the argument in increment was called x then a → x
- mapped/copied into another variable→ **call by value**

Parameter passing (call by reference)

```
void increment(int* p){
    printf("address of p in function= %p \n", &p); // 350
    printf("value of p in function= %p \n", p); // 308
    printf("value of *p = %d \n", *p); // 10
    *p = *p +1;
}
```

```
int main(){
  int a=10;
  printf("address of a in main = %p \n",
  &a); // 308
  increment(&a);
  printf("value of a in main = %d \n",
  a); // 11
}
```



Another example Adding- call by reference

```
int add(int* num1, int* num2){

  return *num1 + *num2;
}

int main(){
  int a=2;
  int b=4;
  int c = add(&a,&b);
  printf("%d",c);
}
```

- Can the following program be modified so it uses a pointer to return the answer?

Be very careful when using pointers

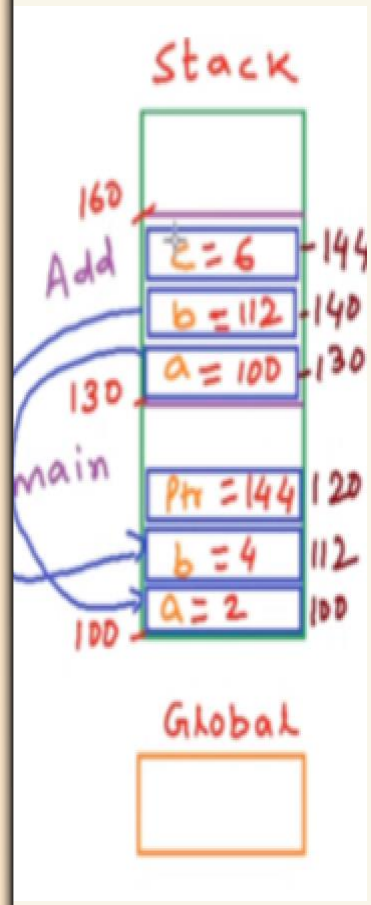
- If this is your answer, then you are on the right track,
  - but ……. This is not going to work!

```
int* add(int* num1, int* num2){

  int ans =  *num1 + *num2;
  return &ans;
}

int main(){
  int a=2;
  int b=4;
  int* c = add(&a,&b);
  printf("%d",*c);
}
```

See the pointer_adding.c example

Be very careful when using pointers

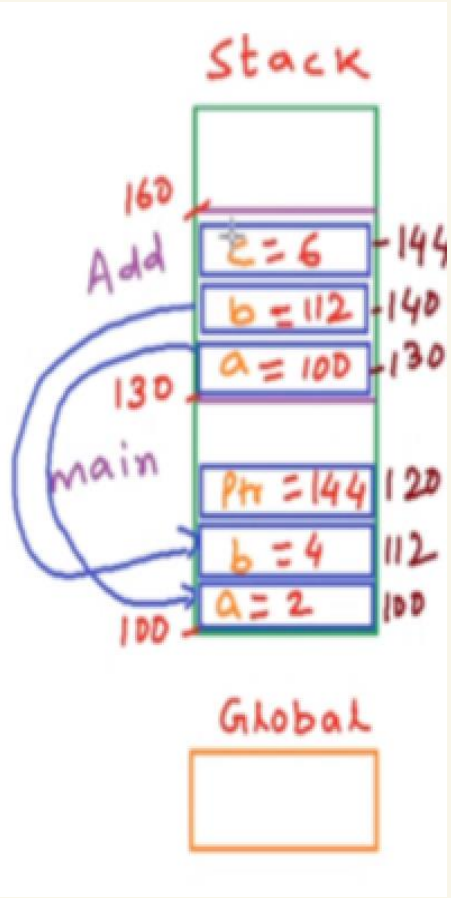
```
void printHelloWorld(){
  printf("Hello World! \n");
}

int* add (int* a, int* b){
  int c = *a + *b ;
  return &c;
}

int main(){
  int a = 2;
  int b = 4;
  int* ptr = add(&a, &b);
  printHelloWorld();
  printf("Sum = %d", *ptr);
}
```

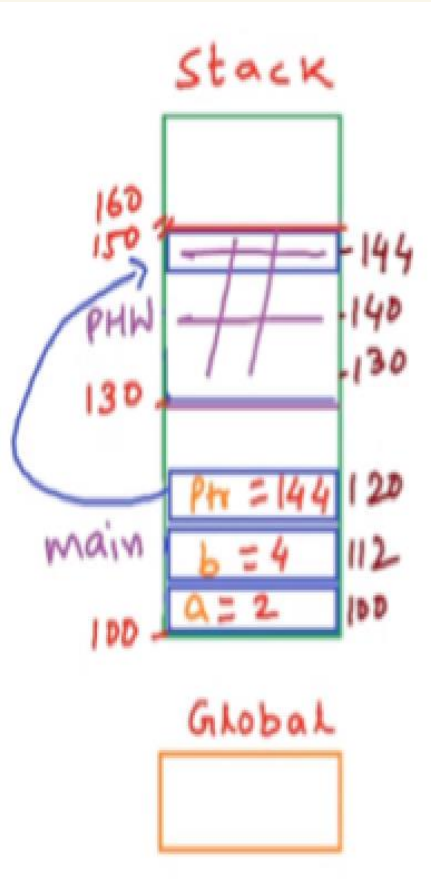


What actually happens is that section of memory is marked as OK to overwrite, but it's still there in memory, at least until its overwritten.

Be very careful when using pointers

```
void printHelloWorld(){
  printf("Hello World! \n");
}

int* add (int* a, int* b){
  int c = *a + *b ;
  return &c;
}

int main(){
  int a = 2;
  int b = 4;
  int* ptr = add(&a, &b);
  printHelloWorld();
  printf("Sum = %d", *ptr);
}
```





Modifying the Add function so it uses a pointer to return the answer

```
void addptr(int* num1, int* num2, int* result){
  *result =  *num1 + *num2;
}

int main(){
  int a=2;
  int b=4;
  int c;
  addptr(&a,&b,&c);
  printf("%d",c);
  return 0;
}
```

Example of the Usage of Pointers

Write code for the `calculate_triangle_area` function

```
int main(){
    double  base, height, area;

    base = 10.0;
    height = 5.0;
    calculate_triangle_area(&base, &height, &area);
    printf("The area of the triangle is: %0.1f", area);

    return 0;
}
```

Example of the Usage of Pointers

```
int main(){
    double  base, height, area;

    base = 10.0;
    height = 5.0;
    calculate_triangle_area(&base, &height, &area);
    printf("The area of the triangle is: %0.1f", area);

    return 0;
}
```

```
void calculate_triangle_area(double* b, double* h, double* a)
{
   *a = 0.5 * *b * *h;
}
```

See the pointers_triangle_area.c example

Addresses, &, and Call-by-Reference

- Arrays are always assumed to be pass-by-reference

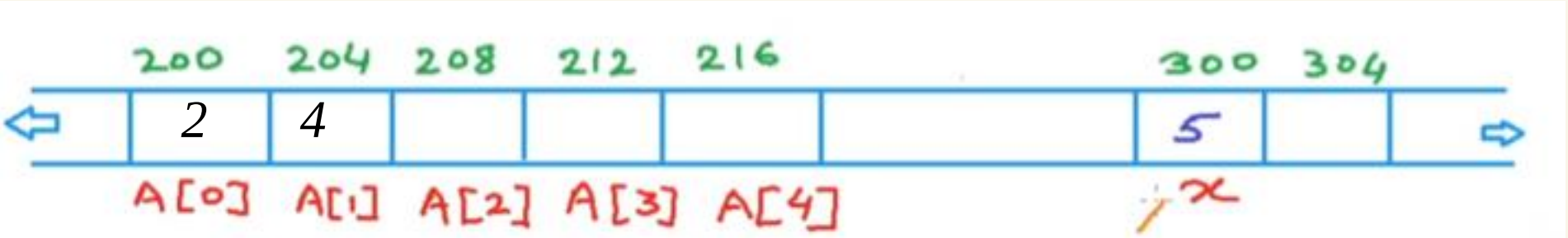
```
double getMaximum(double data[], int size); /* prototype */
double getMaximum(double * data, int size); /* equivalent */
answer = getMaximum(myArray, length);       /* function call */
```

- Note that we don't need to provide "&" when specifying the address of the whole array;
- However, if we want to specify the *address* of an individual element (cell) in the array, then we would do so (e.g., `&data[4]`).

Pointers arithmetic

```
int A[5];
int* q = &A[0];
printf("%p \n",q); // 200
A[0] = 2;
A[1] = 4;
printf("%d \n",*q); // 2 printf("%d
\n",*(q+1)); // 4
```

```
int x=5;
int* p = &x;
printf("%p \n",p); // 300
printf("%d \n",*p); // 5
printf("%d \n",*(p+1));//garbage
```



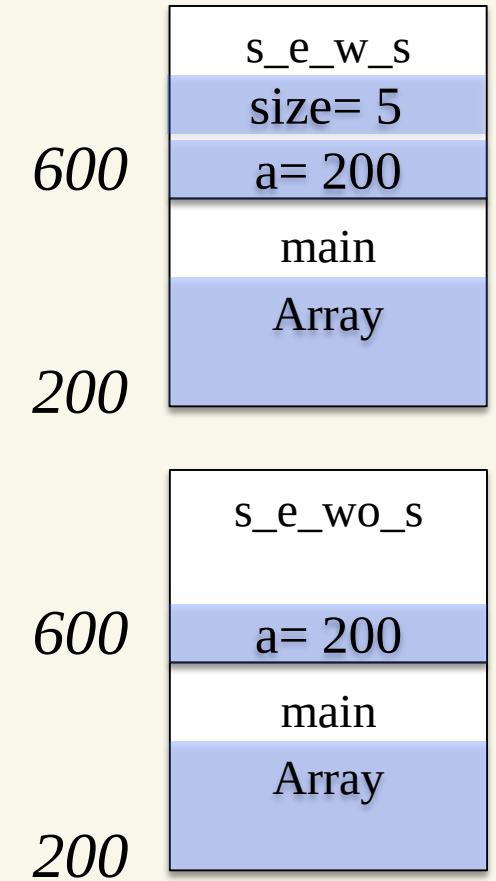
Arrays as function arguments

See example array_function.c

In this example we're going to look at two functions that find the sum of the values inside of an array.

```
int sum_elements_with_size(int a[], int size);
```

```
int sum_elements_without_size(int a[]);
```



The second function cannot find the correct sum because it cannot determine how big the array is.

Dynamic Memory Allocation

- Sometimes we don't know *how much* memory we need at compile time.
- Every user may use the program differently. For example, suppose you allocate an array capable of holding 1000 integers, and you hardcode the value "1000".
  - What if the user plans to store more than 1000 integers?

    Change program and recompile
  - What if the user only needs 5 integers, and not 1000?

    Wasting memory
- If you hardcode the value 1000 in the body of the program, this is not good. Why not?

  Hard to find and change
- If you hardcode it as a symbolic constant, this is better; but, there's still a problem: Cannot change it without changing code

Memory management in C

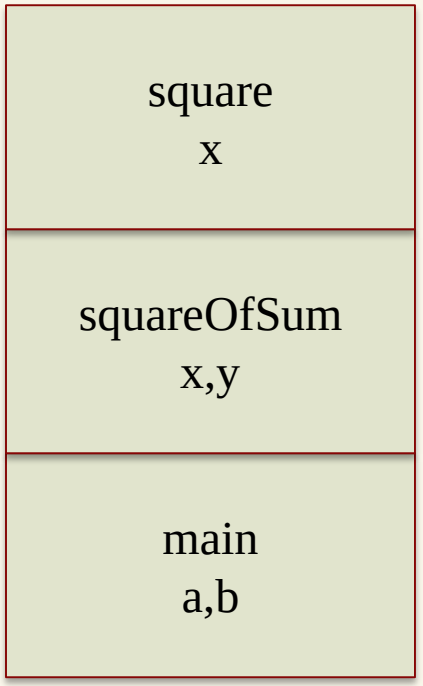
- Frame/Stack segment
  - Area of memory that temporary holds arguments and variables
  - There is no need to manage the memory yourself, variables are allocated and freed automatically
  - Stack has size limits
  - Stack variables only exist while the function that created them, is running

```
void MyFunction( ){
        int i; // created on stack
}// variable goes out of scope and is now deleted.
```

Stack Example

```
int square (int x){
    return x*x;
}

int squareOfSum(int x, int y){
    return square(x+y);

}
int main() {
    int a = 4;
    int b = 8;
    int total = squareOfSum(a, b);
     printf("%d", total);
}
```

Stack



Having large data structures on the stack is a problem and leads to Stack overflow

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"):

- Heap objects must **explicitly be deleted** by the programmer.

Dynamic Memory Allocation

Function `malloc` returns a *pointer* to a memory block of at least `size` bytes:

```
ptr = (cast-type*)malloc(byte-size);
```

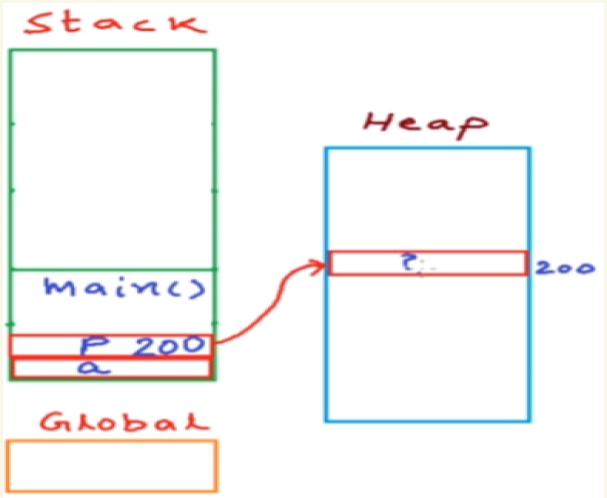
Function `free` returns the memory block (previously allocated with `malloc`) and pointed to by `ptr` to the memory heap:

```
free(ptr);
```

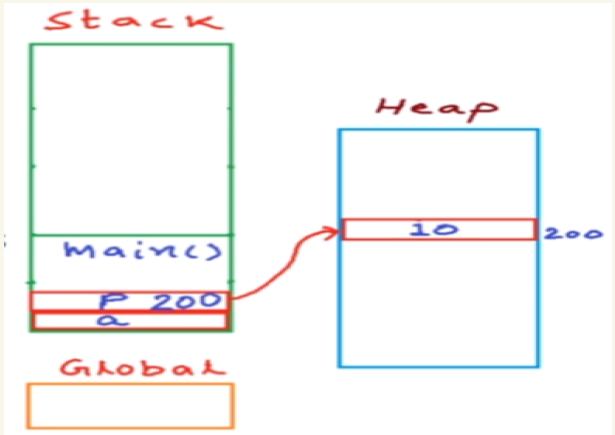
Yes, the computer remembers how many bytes need to be freed, provided you give it the correct address (`ptr`).

Heap example

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

int main(){
    int a; /* goes on stack */
    int* p = (int*) malloc(sizeof(int));
}
```




```
int main(){
    int a; /* goes on stack */
    int* p = (int*) malloc(sizeof(int));
    *p =10;
}
```



- What if there is no free memory left on the heap?

  malloc will return a null pointer.

More examples

1. Suppose we want to allocate space for *exactly* 10 integers in an array:

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

int main(){
    int* i;
    i = (int*) malloc(10 * sizeof(int));
    if (i == NULL){
        printf("Error: can't get memory ...\n");
        exit(1);   /* terminate processing */
    }

    i[0] = 3; // *(i+0)=3;
    i[1] = 16; // *(i+1) = 16;
    printf("%d",*i);
    /*  perform some actions */
}
```

More examples

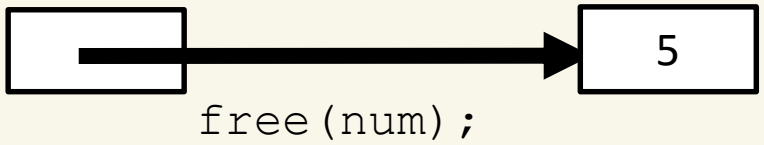
- Suppose we want to allocate space for *a variable number* of employees ‘ hourly wages in an array:

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

int main(){
    int      employees, index;
    double*    wages;
    printf(" Total number of employees? ");
    scanf("%d", &employees);  /* user enters # of employees */

    wages = (double*) malloc(employees * sizeof(double));
    if (!wages){  /* same as “if (wages == Null)” */
        printf("error, no more memory available \n");
      }

    printf(" everything’s OK \n");
    /*  perform some actions */
}
```

See the dma_examples.c for another example

Dangling pointer

When we're done with the object we **free** it, which reclaims the memory




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

int main(){
 int* i = (int*)malloc(sizeof(int));
 *i = 5;
 free(i);
 printf("%d",*i); // 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.

Dangling pointer and NULL pointer

- Thus, whenever you call `free`, you must set the pointer either to a new value, if you're reusing the pointer, or to **NULL**.
- Think of NULL as the "absence of an object"-- its nothing, literally!

- Suppose we want to free the storage allocated in the previous two examples:

```
/* int* i;                              */
/* i = (int*) malloc(10 * sizeof(int)); */
...
free(i);     /* i becomes a dangling pointer      */
i = NULL;    /* i is no longer a dangling pointer */

/* double* wages;                                    */
/* wages = (double*) malloc(employees * sizeof(double)); */
...
free(wages);
wages = NULL;
```

Segmentation Faults

- Don't try to dereference a dangling pointer
  - This will usually crash the program, because you're trying to dereference memory that doesn't belong to you (e.g., the memory belongs to someone else, or it belongs to the operating system and is inaccessible to your program)

- Also, if you've already freed memory, don't try to re-access it.
- Also, don't go out-of-bounds on an array or other data structure.

malloc vs. calloc

- Contrast memory allocation (malloc) and cleared allocation (calloc):

```
double* x = (double *) malloc(number_to_get * sizeof(double));
double* y = (double *) calloc(number_to_get, sizeof(double));
…
```

malloc: The contents of the memory you acquire from the heap are deemed to be uninitialized. Think of the locations as containing "garbage values".

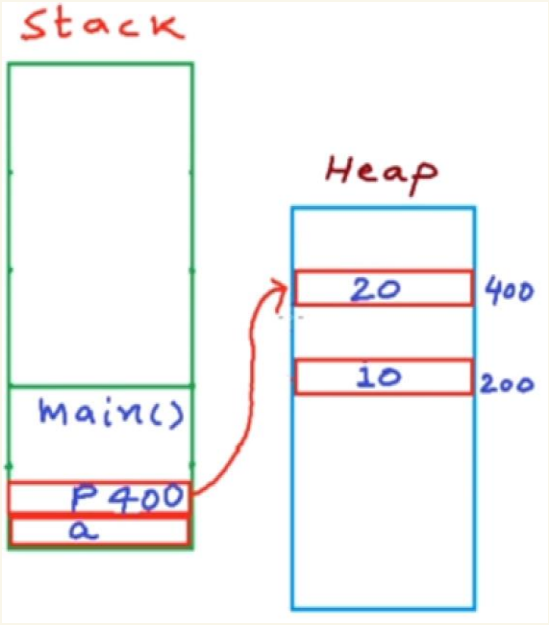
calloc: The memory you acquire is set (cleared) to binary zeros.

Often, you don't need calloc because the programmer is going to explicitly populate the memory locations with appropriate data.

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; /* goes on stack */
    int* p = (int*) malloc(sizeof(int));
    *p =10;
    p = (int*) malloc(sizeof(int));
    *p = 20;
}
```



Memory Leaks

- More examples

```
while (1) {   /* endless loop */
   getMemory = (char *) malloc(40960);
   if (getMemory == NULL) {
     printf("Error: no more memory available\n");
     system("pause");
     return -1;
   }
   printf("got memory at location %p\n", getMemory);
}
```

See the memory_leak.c example

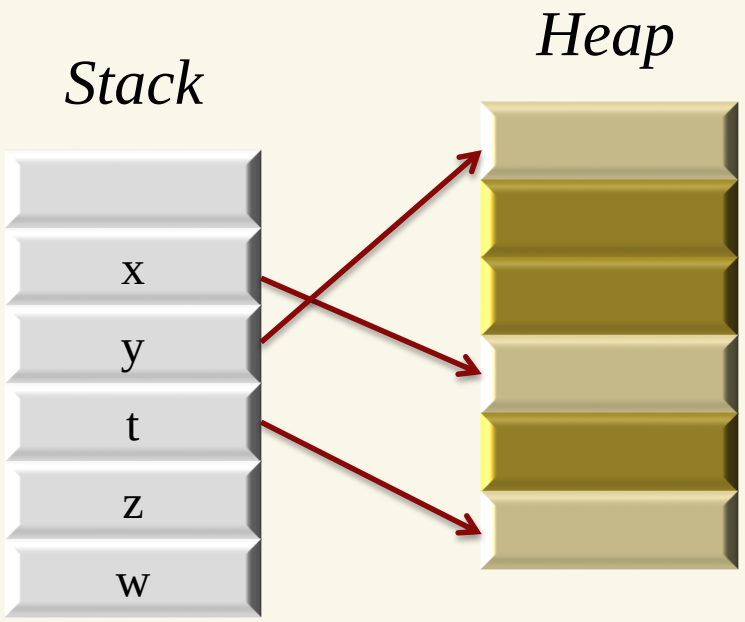
Example

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

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x= %d     *y= %d      z=%d      w=%d", *x, *y, z, w);
```

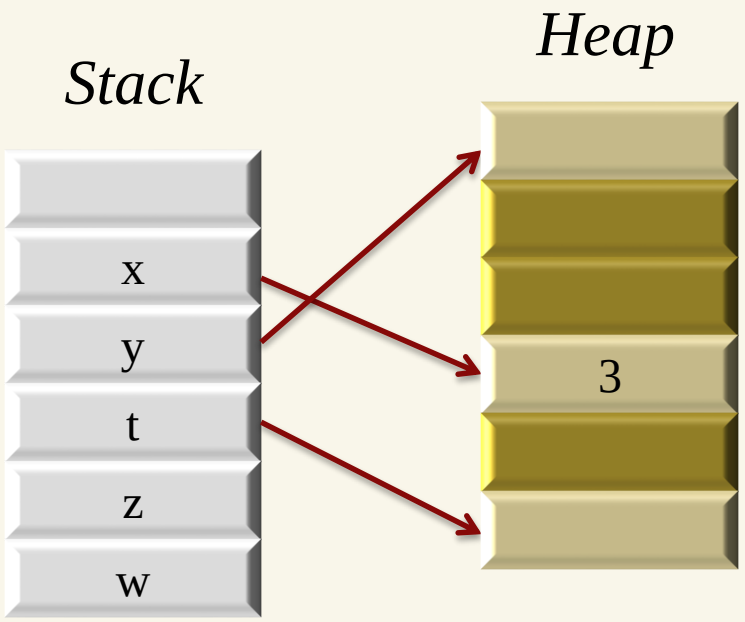
Example

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x= %d     *y= %d     z=%d     w=%d", *x, *y, z, w);
```



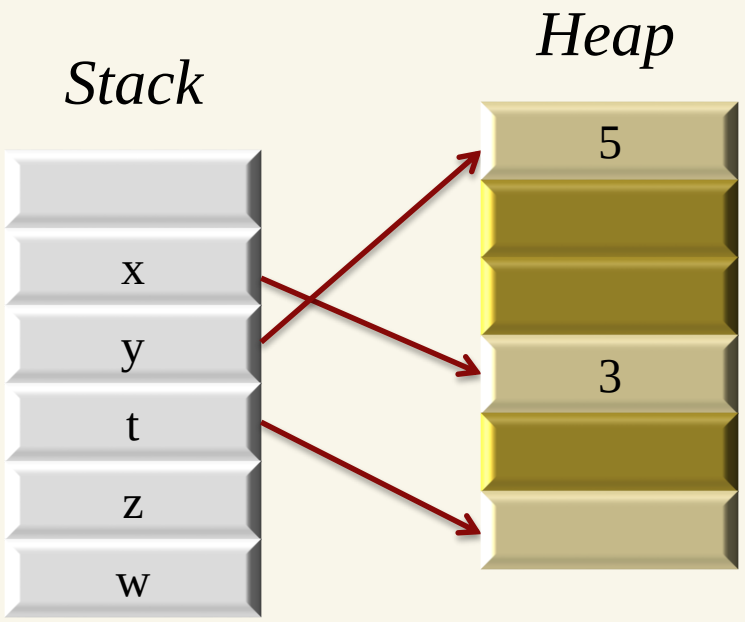
Example

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x= %d     *y= %d     z=%d     w=%d", *x, *y, z, w);
```



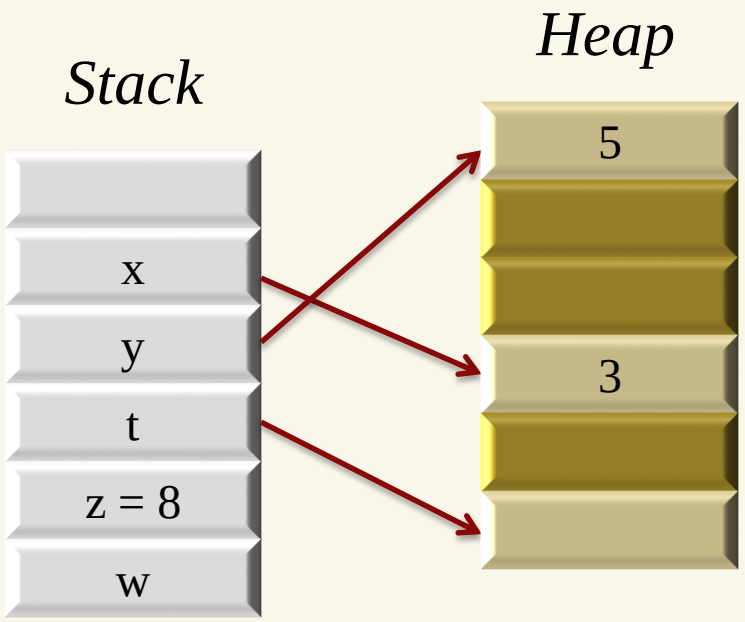
Example

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x= %d     *y= %d     z=%d     w=%d", *x, *y, z, w);
```



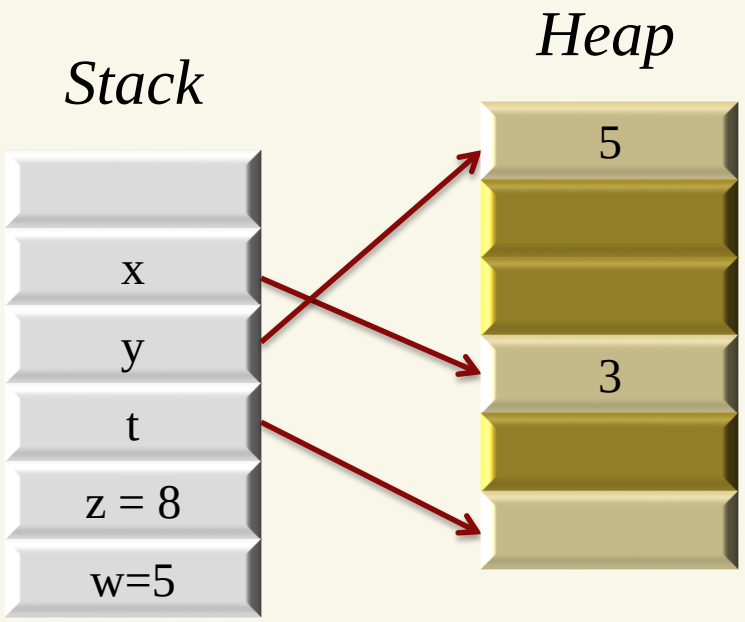
Example

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x= %d     *y= %d     z=%d     w=%d", *x, *y, z, w);
```



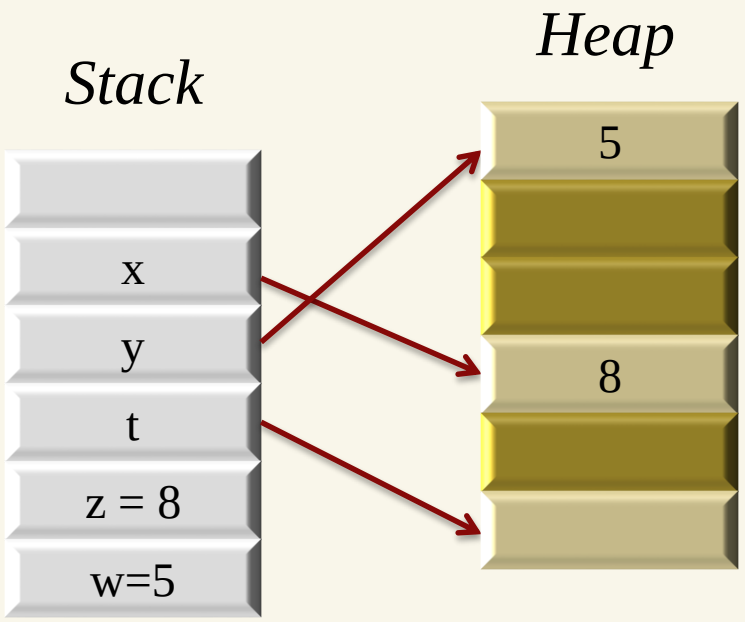
Example

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x= %d     *y= %d     z=%d     w=%d", *x, *y, z, w);
```



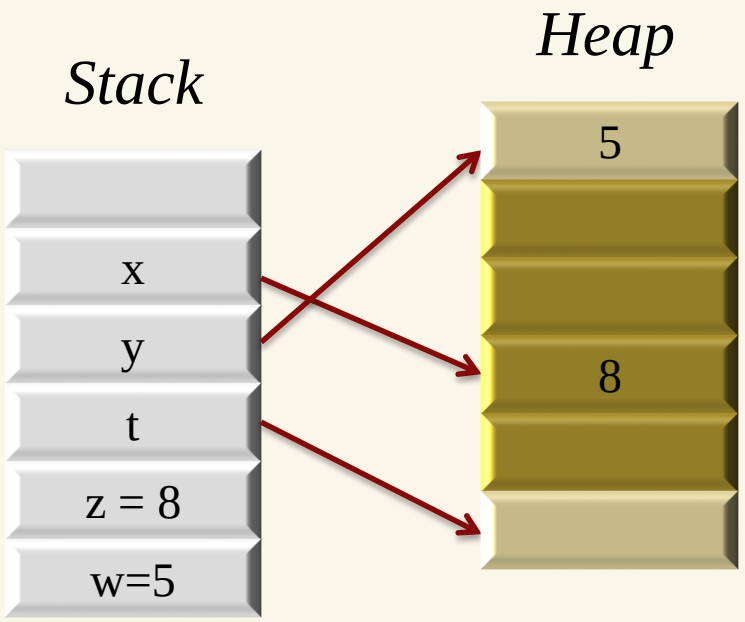
Example

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x= %d     *y= %d     z=%d     w=%d", *x, *y, z, w);
```



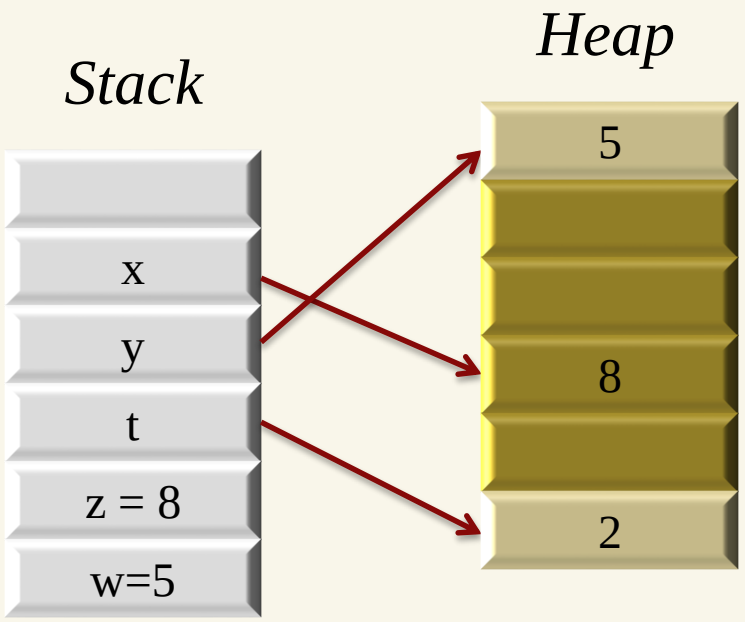
Example

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x= %d     *y= %d     z=%d     w=%d", *x, *y, z, w);
```



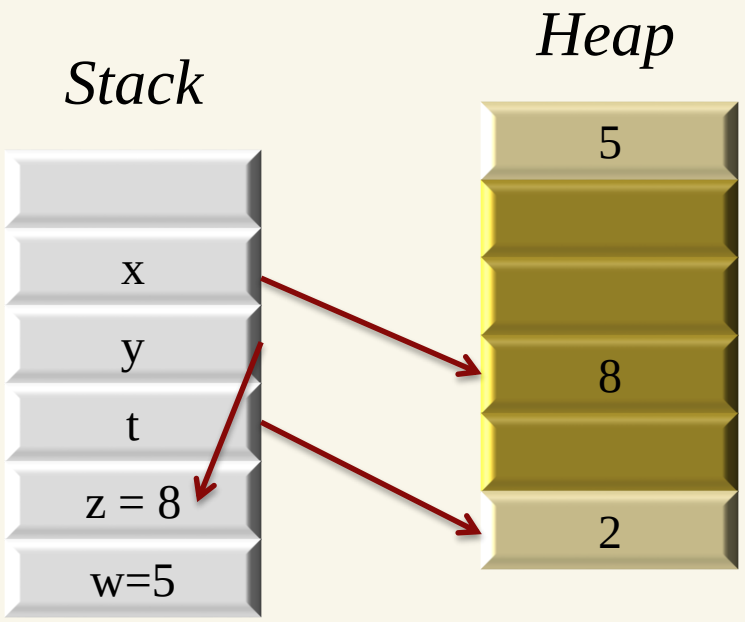
Example

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x= %d     *y= %d     z=%d     w=%d", *x, *y, z, w);
```



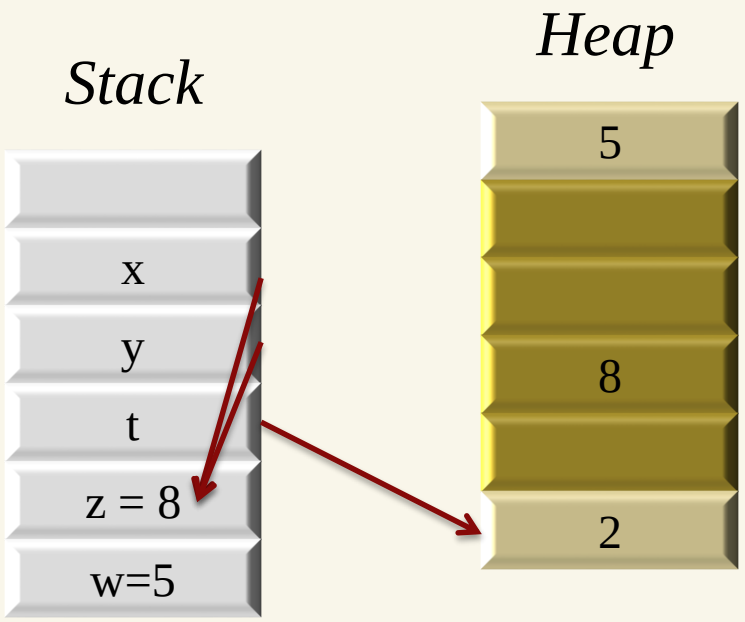
Example

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x= %d     *y= %d     z=%d     w=%d", *x, *y, z, w);
```



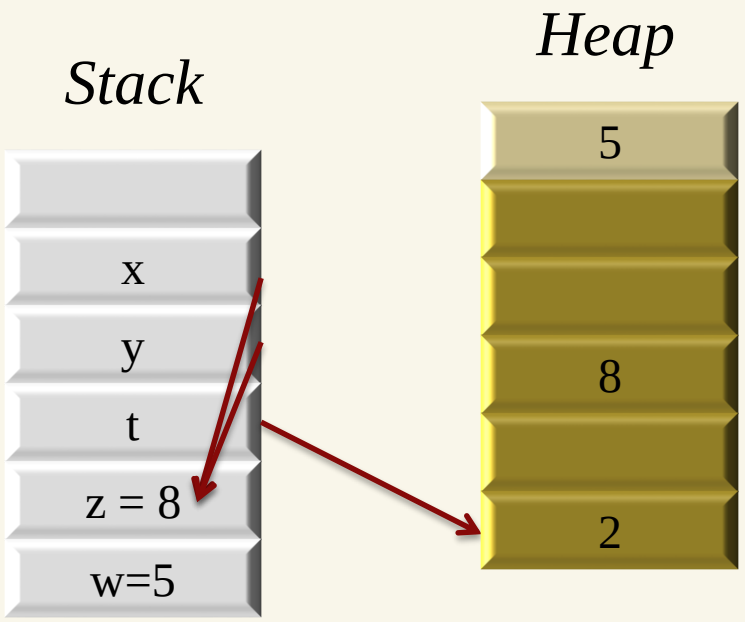
Example

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x= %d     *y= %d      z=%d      w=%d", *x, *y, z, w);
```



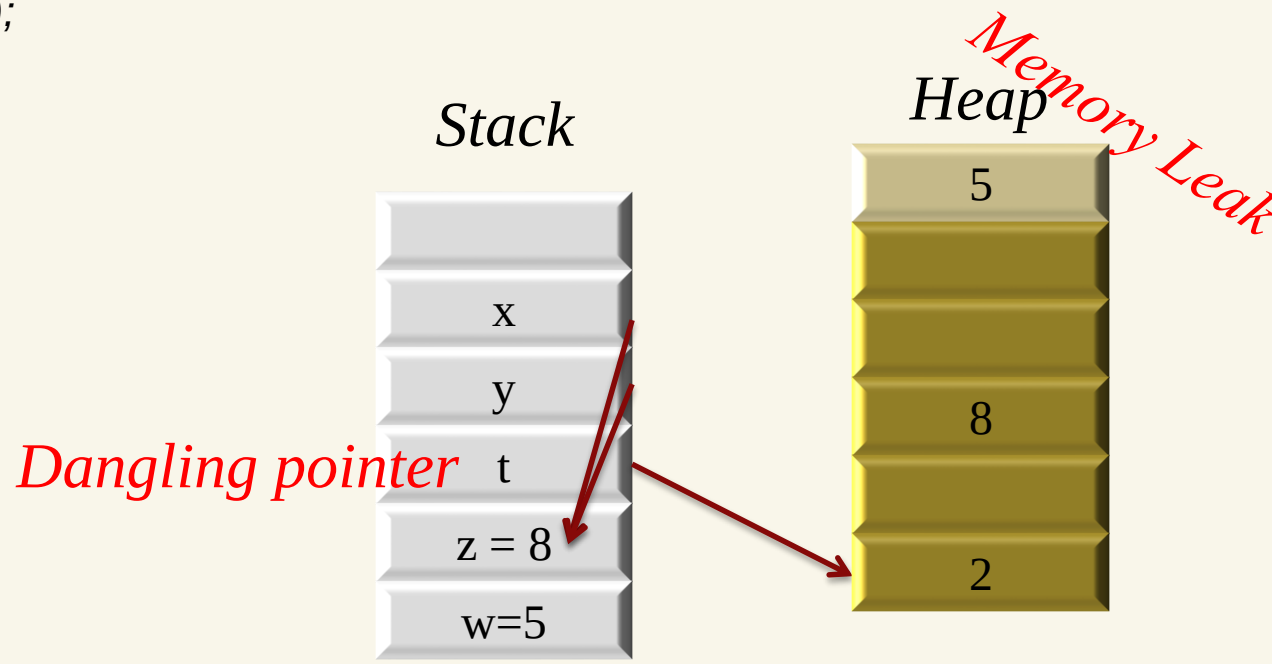
Example

```
int w;
int z;
int* t = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* x = (int*) malloc(sizeof(int));

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x= %d     *y= %d      z=%d      w=%d", *x, *y, z, w);
```



Example

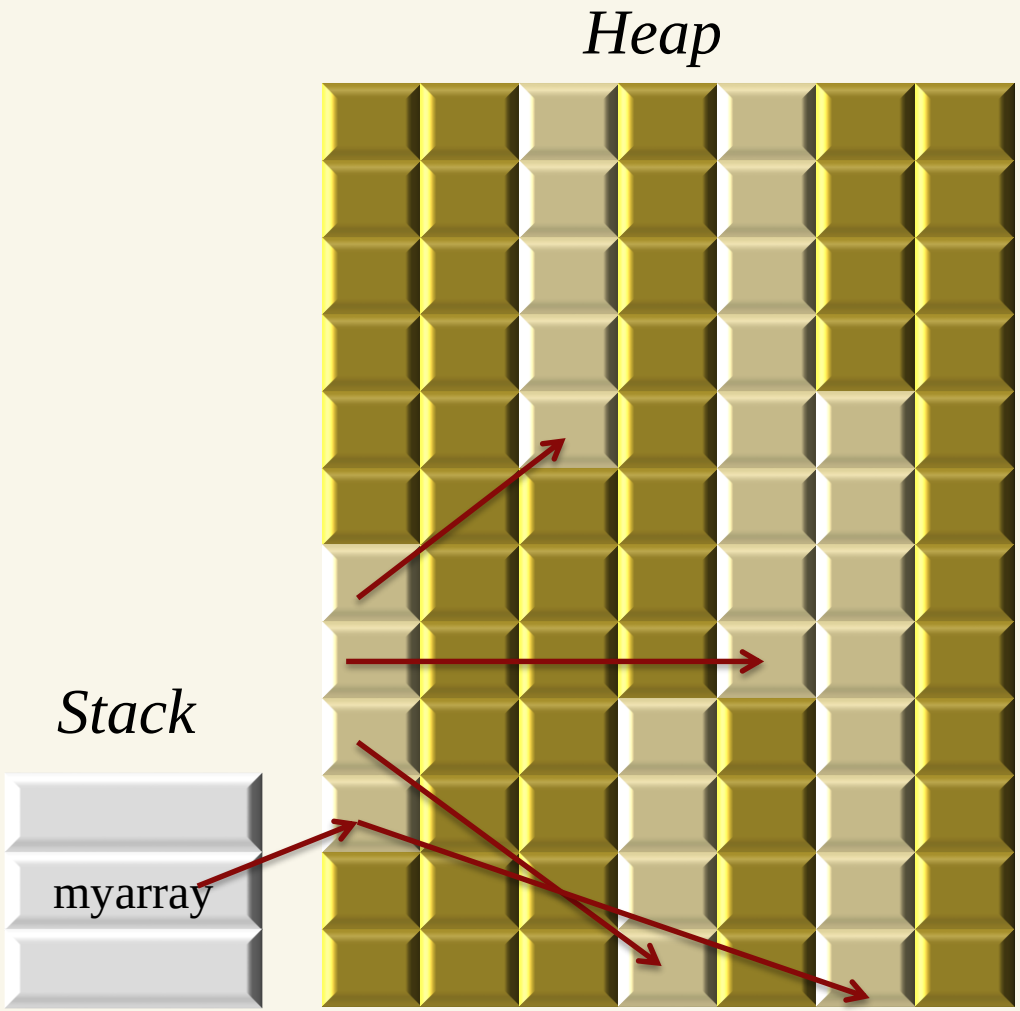
```
int* x = (int*) malloc(sizeof(int));
int* y = (int*) malloc(sizeof(int));
int* t = (int*) malloc(sizeof(int));
int z;
int w;

*x = 3;
*y = 5;
z = *x + *y;
w = *y;
*x = z;
free(x);
*t = 2;
y = &z;
x = y;
free(t);
printf("*x= %d     *y= %d     z=%d     w=%d", *x, *y, z, w);
```

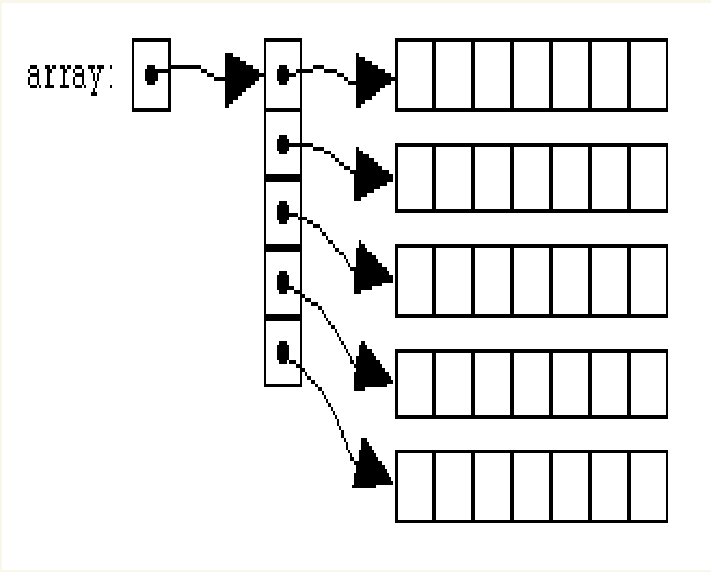


x= 8   *y= 8   z=8   w=5

Dynamic Allocation of a 2-D Array

See dma_2d.c please





Stack vs Heap Pros and Cons

- Stack
  - very fast access
  - don't have to explicitly de-allocate variables
  - space is managed efficiently by CPU, memory will not become fragmented
  - local variables only
  - limit on stack size (OS-dependent)
  - variables cannot be resized

- *Heap*
  - variables can be accessed globally
  - no limit on memory size
  - (relatively) slower access
  - no guaranteed efficient use of space
  - you must manage memory (you're in charge of allocating and freeing variables)
  - variables can be resized

Learning Goals revisited

- Describe the purpose of a pointer data type.
- Describe the relationship between addresses and pointers.
- Explain the difference in parameter passing for call-by-value versus call-by-reference.
- Explain the purpose of dynamic memory allocation. Give examples of where dynamic memory allocation is particularly useful, and examples of where it is not (i.e., where static allocation (at compile time) is better).
- Gain experience with pointers in C and describe their tradeoffs and risks (e.g., dangling pointers, memory leaks).
- Demonstrate how dynamic memory management is handled in C (e.g., allocation and deallocation from the memory heap).