

```
#include <stdio.h>
int main(void)
{
    int count;

    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```

WENT 10.3



CSE 30

1

Lecture 3 (Part1): C Programming

Functions:

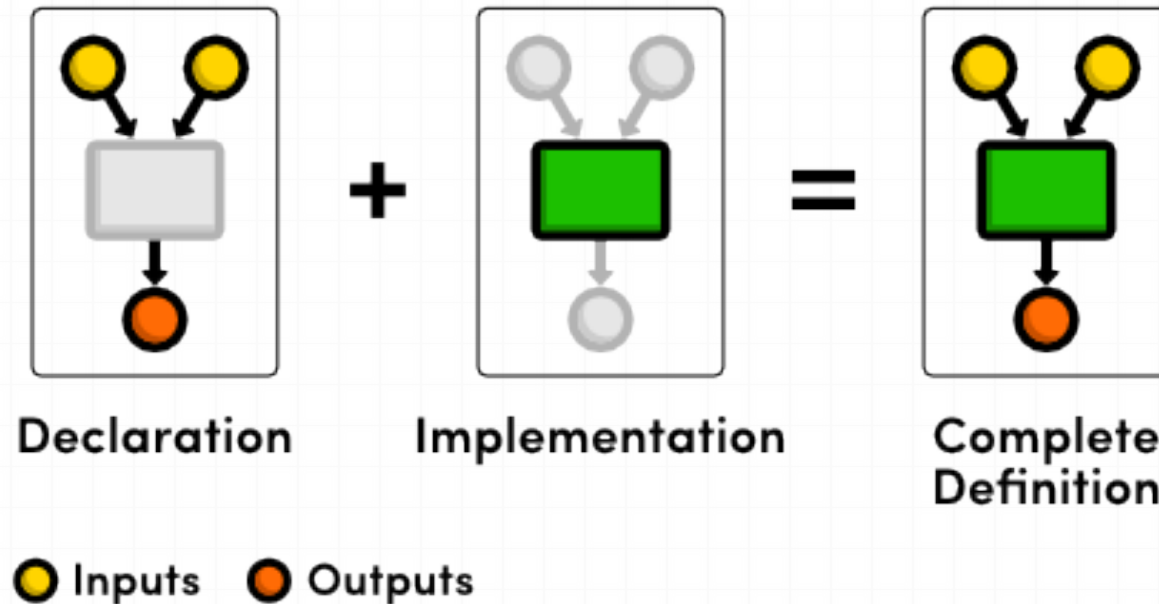
***Breaking large computing tasks to
smaller ones***

C Programming

- Procedural thought process

```
main () /* High level Outline */
{
    . . .
    get_input(arg1) /*Comment: Step 1 */
    perform_step_2(arg2);
    perform_step_3();
    store_result(); /* Print output or store in a file */
}
```

Overview of Functions



Functions make code easy to

- Maintain
- Debug
- Reuse

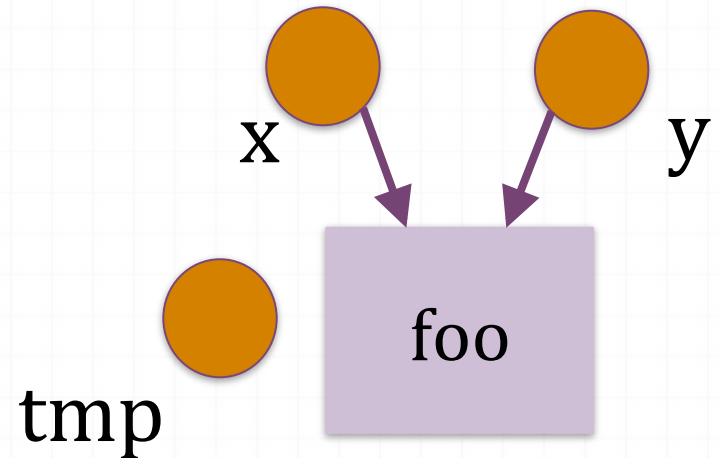
Functions Overview

```
void foo(int x, int y); //Declaration
```

Functions Overview

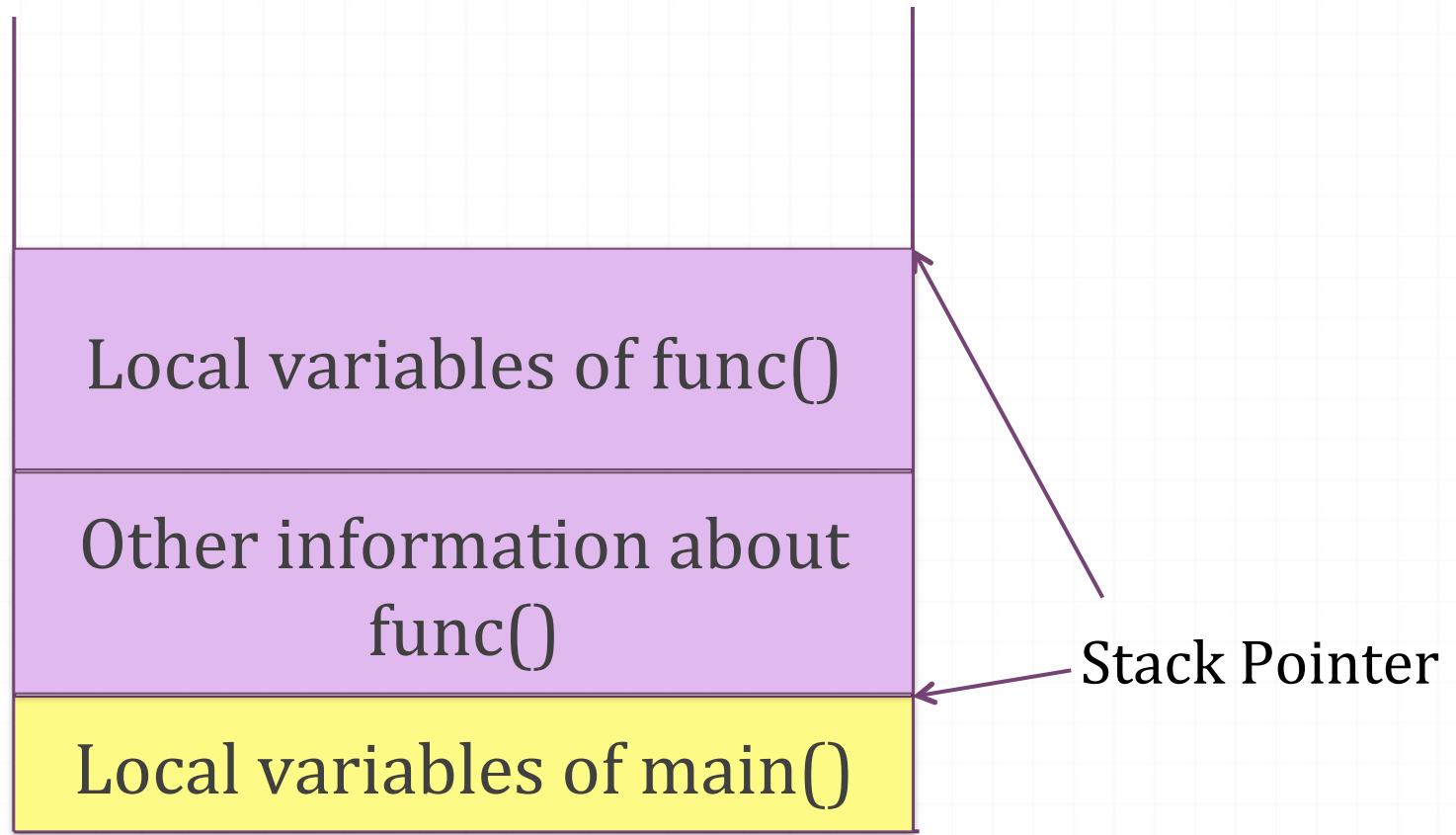
//Definition

```
void foo(int x, int y) {  
    int tmp;  
    tmp= x;  
    x= y;  
    y= tmp; }  
  
What does foo do?
```



Stack Allocation: Function local variables and parameters

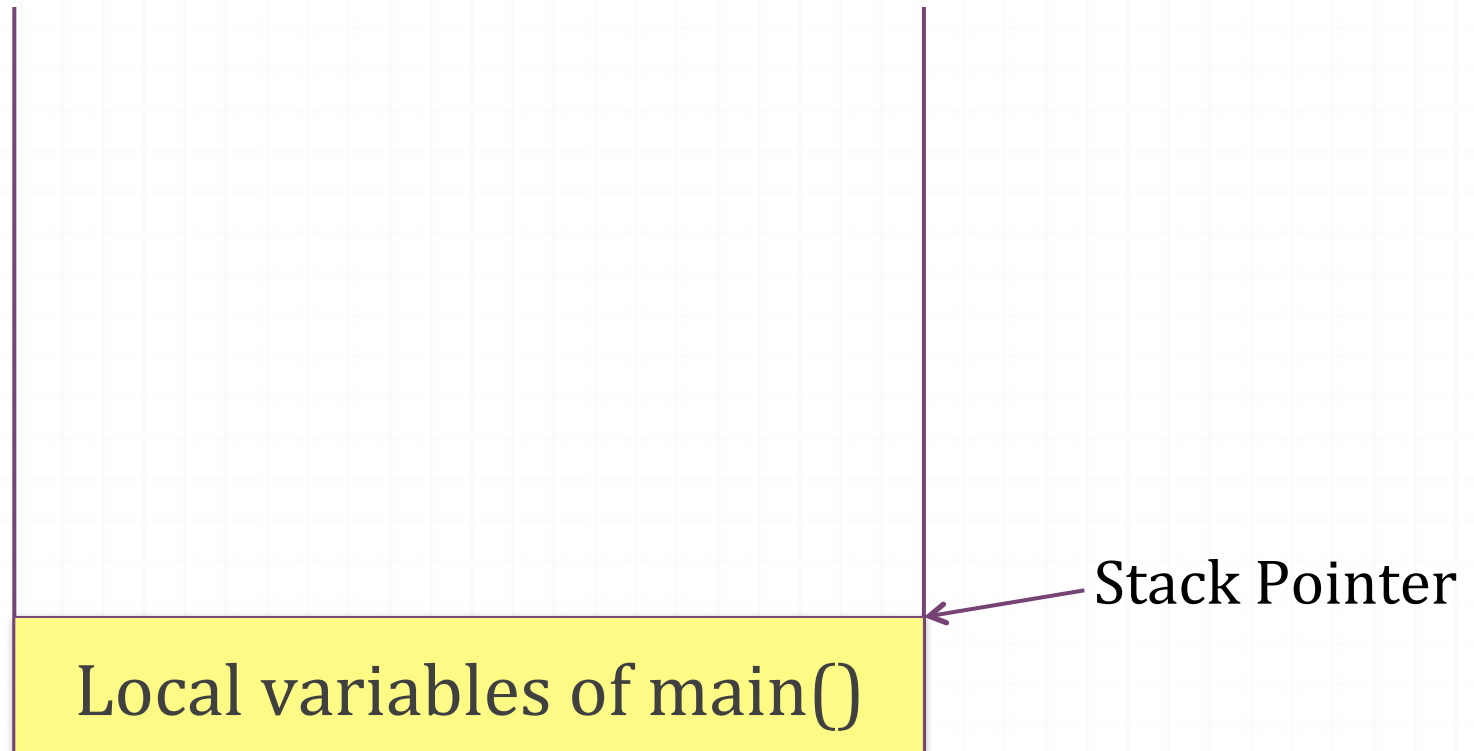
- When program execution starts



What if main calls the function func()?

Stack Allocation: Function local variables and parameters

- Variables whose lifetime is the execution time of function are managed using the stack structure



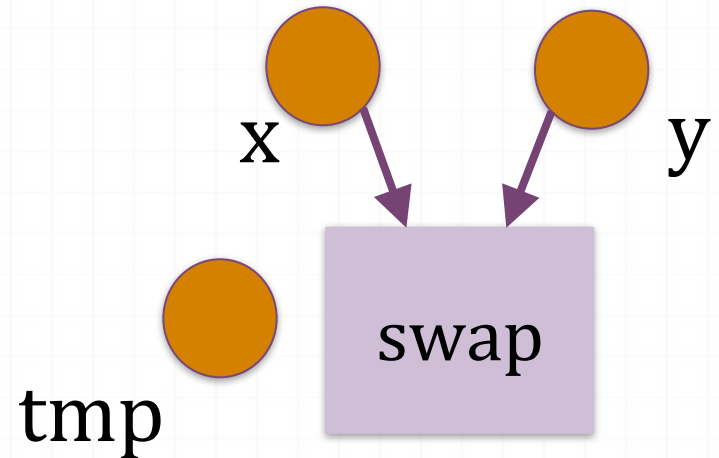
Variables declared within a function

- A. can be used by other functions
- B. can only be used within the function
- C. continue to exist in memory as long as the program executes

Consider the following function

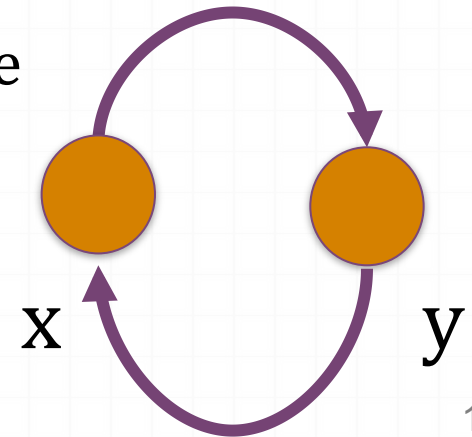
```
void swap(int x, int y); //Declaration
```

```
void swap(int x, int y) {  
    int tmp;  
    tmp= x;  
    x= y;  
    y= tmp; }  
}
```



The swap function intends to swap the value of its inputs. Is the logic correct?

- A. Yes
- B. No
- C. Depends



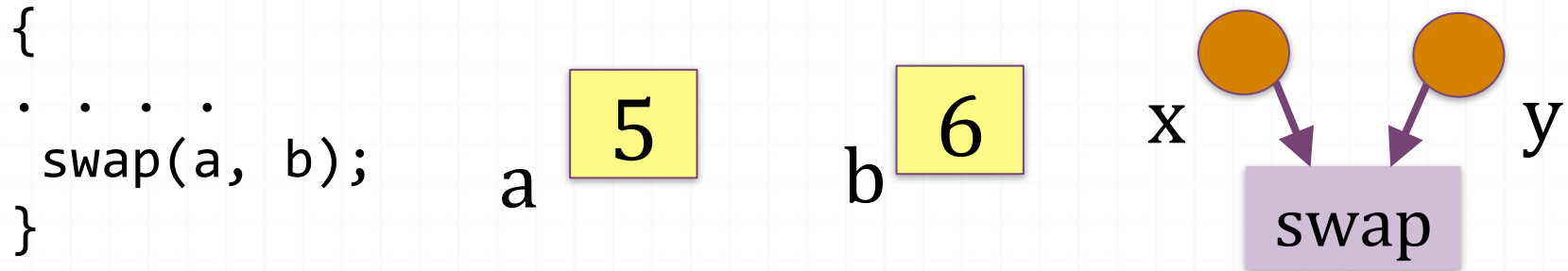
Q: Are the value of variables 'a' and 'b' interchanged after swap is called?

```
main() {  
  . . .  
  swap(a, b);  
  . . . }  
a 5 b 6
```

A. Yes, because

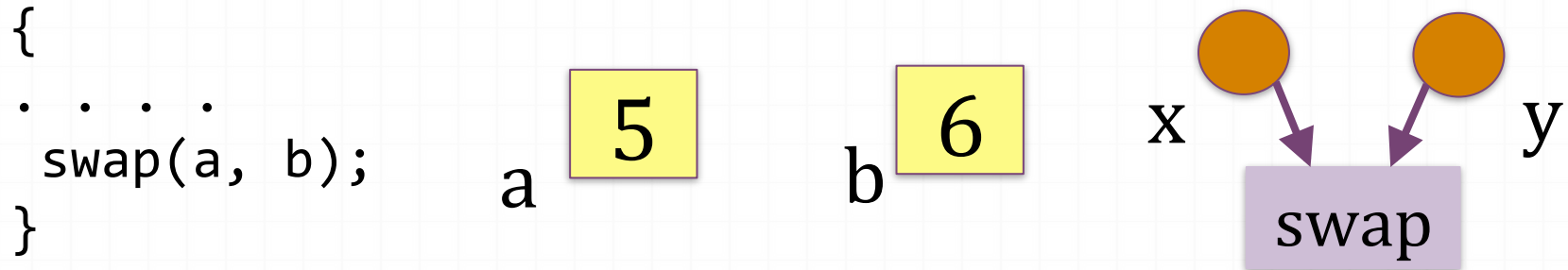
B. No, because.....

Q: Which of the following changes are required to interchange the values in 'a' and 'b' when swap is called?



- A. In swap, return the values of 'x' and 'y' to the main function after swapping them
- B. Declare 'a' and 'b' as global variables, so that they become accessible to the swap routine
- C. Pass the address of 'a' and 'b' to swap instead of their value
- D. Move the implementation in swap to the main function

Q: Which of the following changes are required to interchange the values in 'a' and 'b' when swap is called?



- A. In swap, return the values of 'x' and 'y' to the main function after swapping them
- B. Declare 'a' and 'b' as global variables, so that they become accessible to the swap routine
- C. Pass the address of 'a' and 'b' to swap instead of their value
- D. Move the implementation in swap to the main function

Functions: Call by reference

```
void swap(int *x, int *y) {  
    . . .  
}
```

Q: What should the modified swap function do?

- A. Swap the addresses in 'x' and 'y'
- B. Swap the values pointed to by 'x' and 'y'
- C. Both the above operations are equivalent

Functions: Call by reference

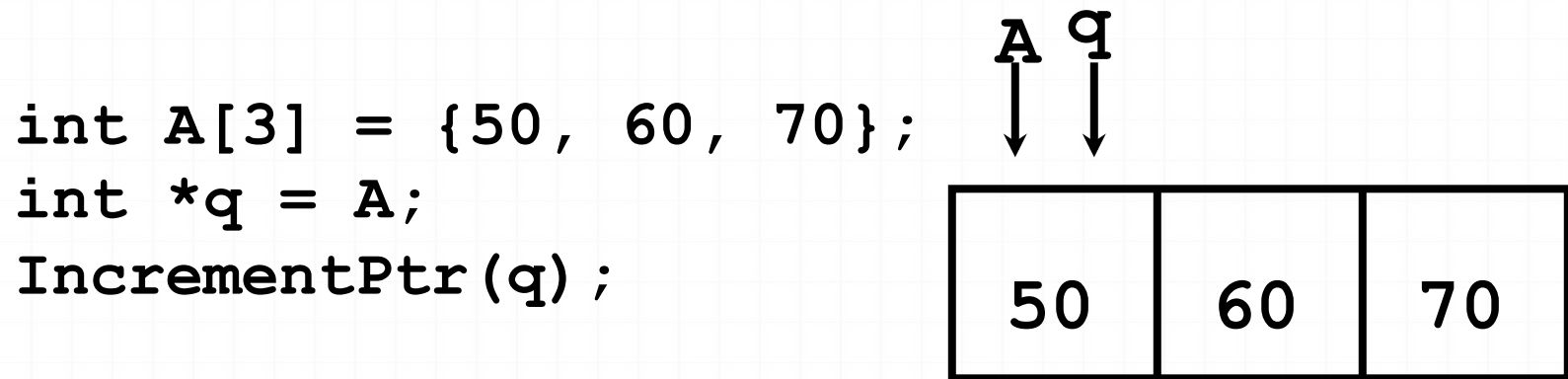
```
void swap(int *x, int *y) {  
    . . .  
}
```

Q: What should the modified swap function do?

- A. Swap the addresses in 'x' and 'y'
- B. Swap the values pointed to by 'x' and 'y'
- C. Both the above operations are equivalent

```
void IncrementPtr(int *p) {  
    p = p + 1; }  
}
```

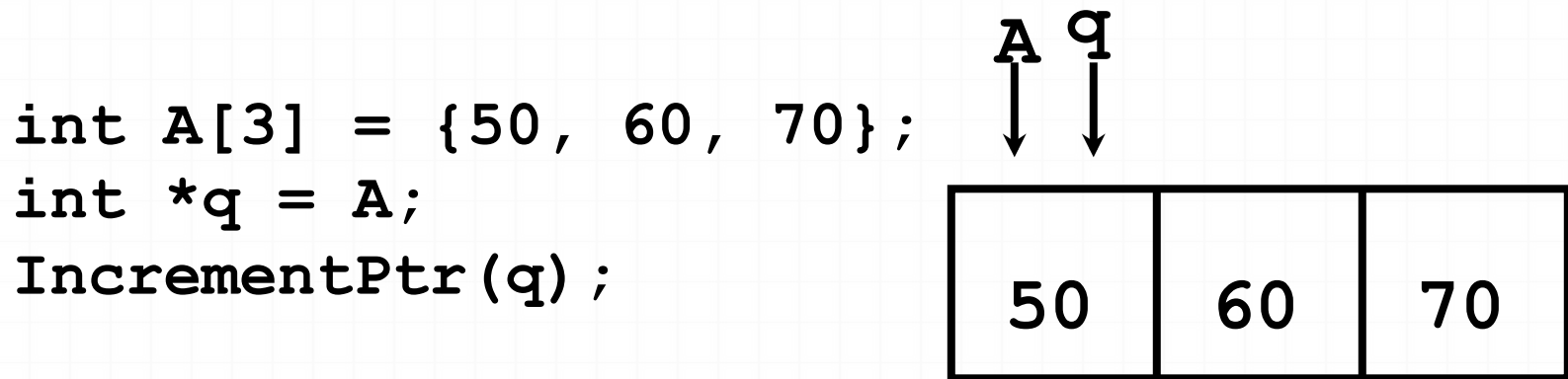
Q: What happens when **IncrementPtr(q)** is called in the following code:



- A. The pointer 'q' points to the next element in the array with value 60
- B. The pointer 'q' points to the first element in the array with value 50

```
void IncrementPtr(int *p) {  
    p = p + 1; }  
}
```

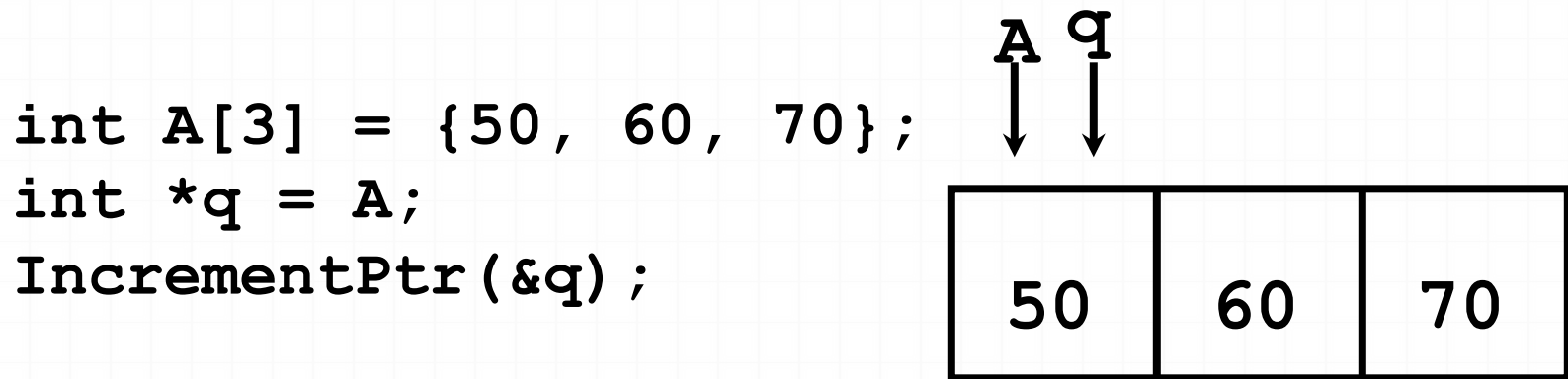
Q: What happens when **IncrementPtr(q)** is called in the following code:



- A. The pointer 'q' points to the next element in the array with value 60
- B. The pointer 'q' points to the first element in the array with value 50

```
void IncrementPtr(int **p) {  
    p = p + 1; }  
}
```

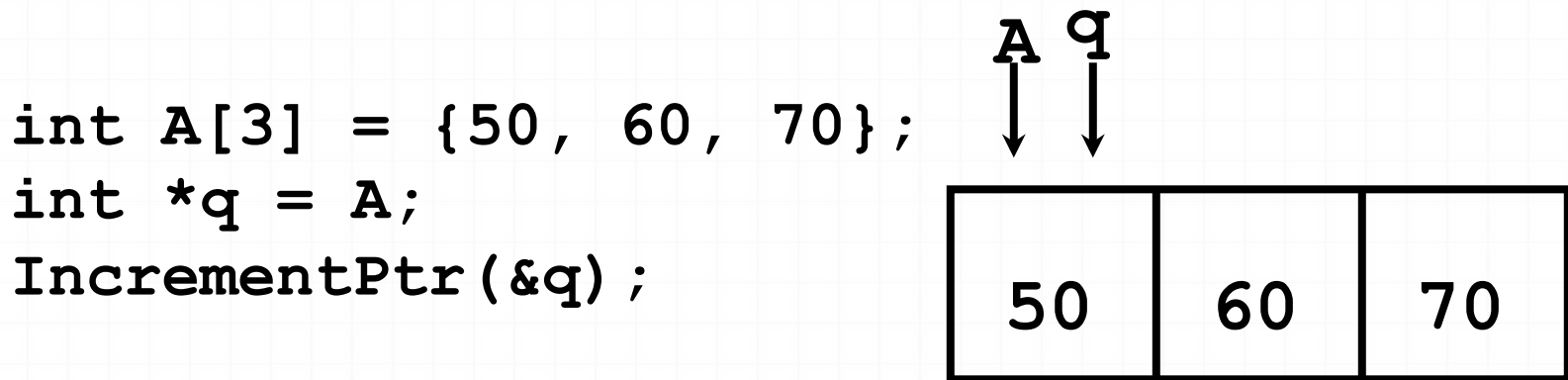
Q: How should we implement `IncrementPtr()`, so that 'q' moves by one element when the following code executes?



- A.** `p = p + 1;` //The current one is correct
- B.** `&p = &p + 1;`
- C.** `*p = *p + 1;`
- D.** `*p++;`
- E.** `p = &p+1;`

```
void IncrementPtr(int **p) {  
    p = p + 1; }  
}
```

Q: How should we implement `IncrementPtr()`, so that 'q' moves by one element when the following code executes?



- A. `p = p + 1;` //The current one is correct
- B. `&p = &p + 1;`
- C. `*p = *p + 1;`
- D. `*p++;`
- E. `p = &p+1;`



Derived data types: Structures

C structures : Overview

- A `struct` is a data structure composed of simpler data types.
 - Like a class in Java/C++ but without methods or inheritance.

```
struct point {  
    int x;  
    int y;  
}  
void PrintPoint(struct point p)  
{  
    printf( “ (%d,%d) ” , p.x, p.y) ;  
}
```

Pointers to structures

- The C arrow operator ($\rightarrow$) dereferences and extracts a structure field with a single operator.
- The following are equivalent:

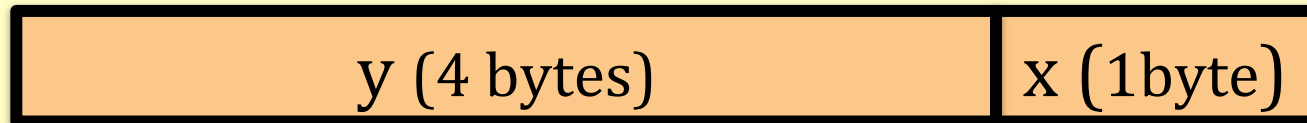
```
struct point *p;
```

```
printf( "x is %d\n" , (*p).x ) ;  
printf( "x is %d\n" , p->x ) ;
```

Representation in memory

```
struct p {  
    int y;  
    char x;  
};  
struct p sp;
```

sp



0x100

0x104

0x105

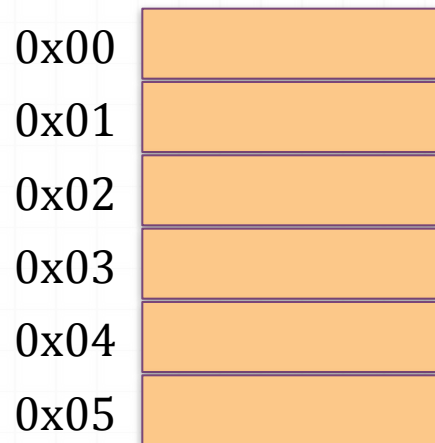
Data Alignment

```
struct p {  
    char x;  
    int y;  
};  
Struct p sp;
```

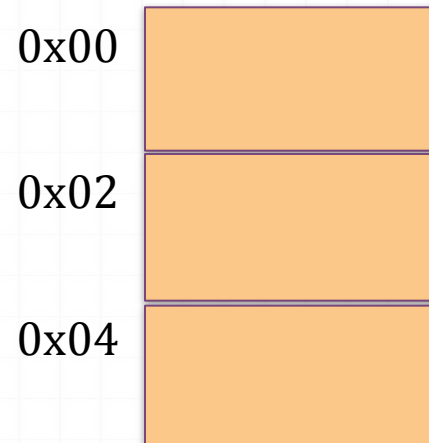
Q: How is the above struct laid out in memory?

Alignment Fundamentals

- Processors do not always access memory in byte sized chunks, instead in 2, 4, 8, even 16 or 32 byte chunks
- Boundaries at which data objects are stored affects the behavior of read/write operations into memory



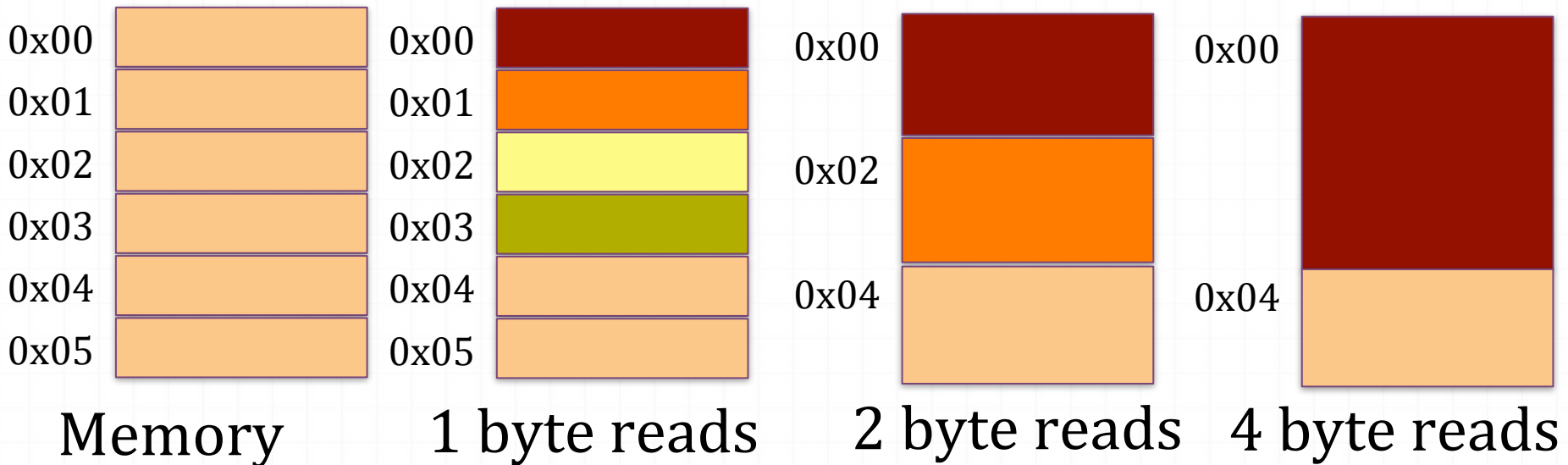
Programmer's view
of memory



Processor's view of
memory

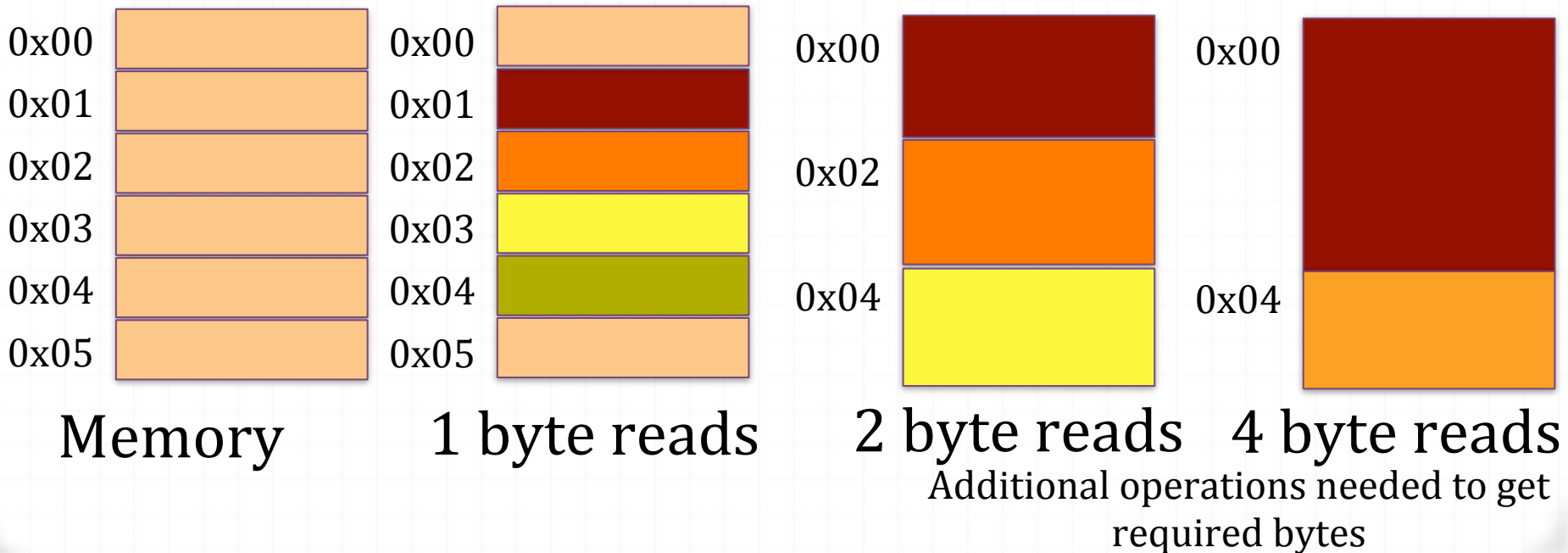
Alignment Fundamentals

- Consider the task of reading 4 bytes from memory starting at 0x00 for processors that read 1, 2, 4 bytes at a time.
- How many memory accesses are needed in each case?



Alignment Fundamentals

- Now consider the task of reading 4 bytes from memory starting at 0x01 for processors that read 1, 2, 4 bytes at a time.
- How many memory accesses are needed in each case?
 - Some processors just would not allow this scenario because it is extra work for the h/w and it affects the performance



Alignment requirements

- Data objects should start at memory addresses that are divisible by the size of the data object.
 - short (2 byte) at address divisible by 2 0b_____0
 - int (4 byte) at address divisible by 4 0b_____00
 - double (8 byte) at address divisible by 8 0b_____000

Alignment requirements

Q: How is 'sp' laid out in memory?

Compiler takes care of zero padding

```
struct p {  
    char x;  
    int y;  
};  
struct p sp;
```

Alignment requirements

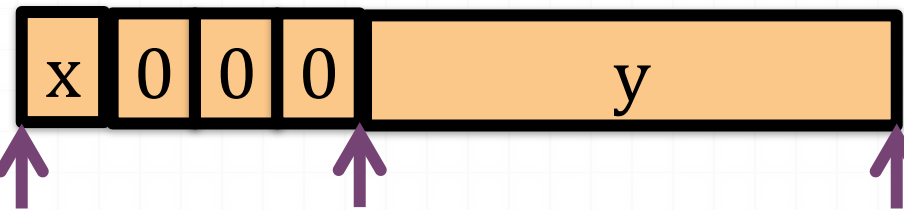
Q: How is 'sp' laid out in memory?

Compiler takes care of zero padding

```
struct p {  
    char x;  
    int y;  
};
```

```
struct p sp; 0x100          0x104          0x108
```

sp



How big are structs?

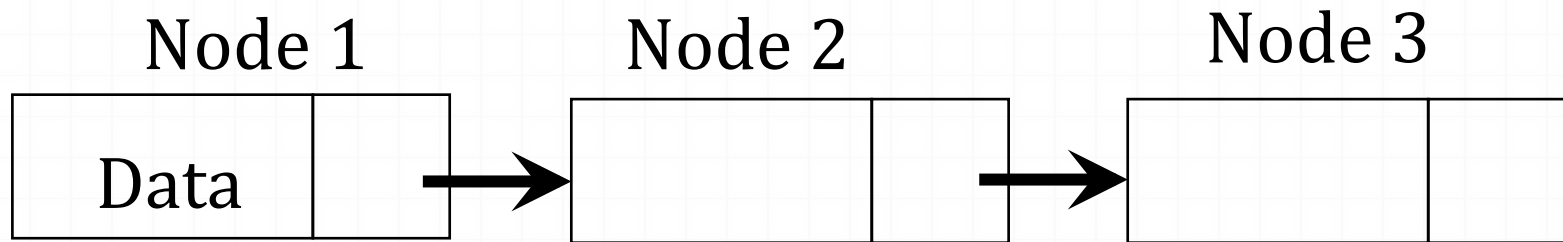
- Recall C operator `sizeof()` which gives size in bytes (of type or variable)

Q: How big is `sizeof(struct p)`?

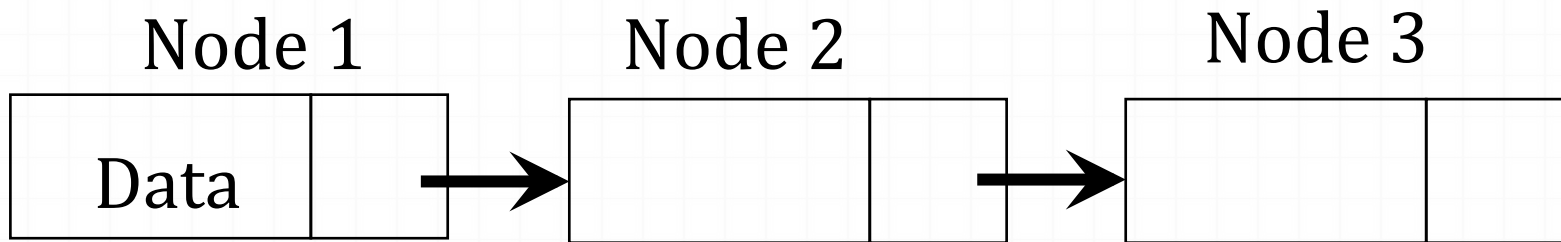
```
struct p {  
    char x;  
    int y;  
};
```

- A. 5 bytes
- B. 8 bytes
- C. 12 bytes

Linked Lists



Linked Lists



A generic linked list has:

- Multiple data objects (structs) also called nodes
- Each node linked to the next node in the list i.e. it knows the address of the next node
- Nodes located at different memory locations (unlike arrays where they are contiguous)

Advantages compared to arrays

- Multiple data members (of different data types) can be stored in each node
- Nodes can be easily inserted or removed from the list without modifying the whole list

Let's look at an example of using structures, pointers, `malloc()`, and `free()` to implement a **linked list of strings**.

```
typedef struct Node node;
```

```
struct Node {  
    char *value;  
    _____ next;  
};
```

node



Q:What is the data type of the variable '**next**'?

- A. struct Node
- B. Node
- C. node
- D. node *

Let's look at an example of using structures, pointers, `malloc()`, and `free()` to implement a **linked list of strings**.

```
typedef struct Node node;
```

```
struct Node {  
    char *value;  
    _____ next;  
};
```

node



Q:What is the data type of the variable '**next**'?

- A. struct Node
- B. Node
- C. node
- D. node *

Adding a node to the list

```
node *list_add(node* head, char *string)
```

```
{  
    //Step 1:Create a new node
```

new_node

```
    _____;  
    return new_node;
```

```
}
```

value=?

next=?

Q: How should we declare and initialize **new_node**?

A. `node *new_node=(node*) malloc(sizeof(node)) ;`

B. `node new_node;`

C. `node *new_node=head;`

D. `node *new_node=(node *)malloc(sizeof(head)) ;`

Adding a node to the list

```
node *list_add(node* head, char *string)
```

```
{  
    //Step 1:Create a new node
```

new_node

```
    _____;  
    return new_node;  
}
```

value=?

next=?

Q: How should we declare and initialize **new_node**?

A. `node *new_node=(node*) malloc(sizeof(node)) ;`

B. `node new_node;`

C. `node *new_node=head;`

D. `node *new_node=(node *)malloc(sizeof(head)) ;`

Q: What is wrong with step 2?

```
node *list_add(node* head, char *string)
{
    node *new_node=(node*) malloc(sizeof(node)) ;
}
```

Q: What is wrong with step 2?

```
node *list_add(node* head, char *string)
{
    node *new_node=(node*) malloc(sizeof(node)) ;
    //Step 2: Fill in its value
    strcpy(new_node->value, string);
    return new_node;
}
```

- A. Step 2 is correct.
- B. We should use the operator '.' instead of '->'
- C. Memory is not allocated for 'value'

Q: What is wrong with step 2?

```
node *list_add(node* head, char *string)
{
    node *new_node=(node*) malloc(sizeof(node)) ;
    //Step 2: Fill in its value
    strcpy(new_node->value, string);
    return new_node;
}
```

A. Step 2 is correct.

B. We should use the operator `.` instead of `->`

C. Memory is not allocated for `value`

Draw pointer diagrams

```
node *list_add(node* head, char *string)
{
    node *new_node=(node*) malloc(sizeof(node));
    new_node->value = (char*) malloc(strlen(string)+1);
    strcpy(new_node->value, string);

}
```

What should Step 3 be?

```
node *list_add(node* head, char *string){  
    node *new_node=(node*) malloc(sizeof(node));  
    new_node->value = (char*) malloc(strlen(string)+1);  
    strcpy(new_node->value, string);  
    //Step 3:  
}
```

A. `new_node->next =head;`

B. `next=head;`

C. `head=new_node;`

D. `new_node->next =*head;`

What should Step 3 be?

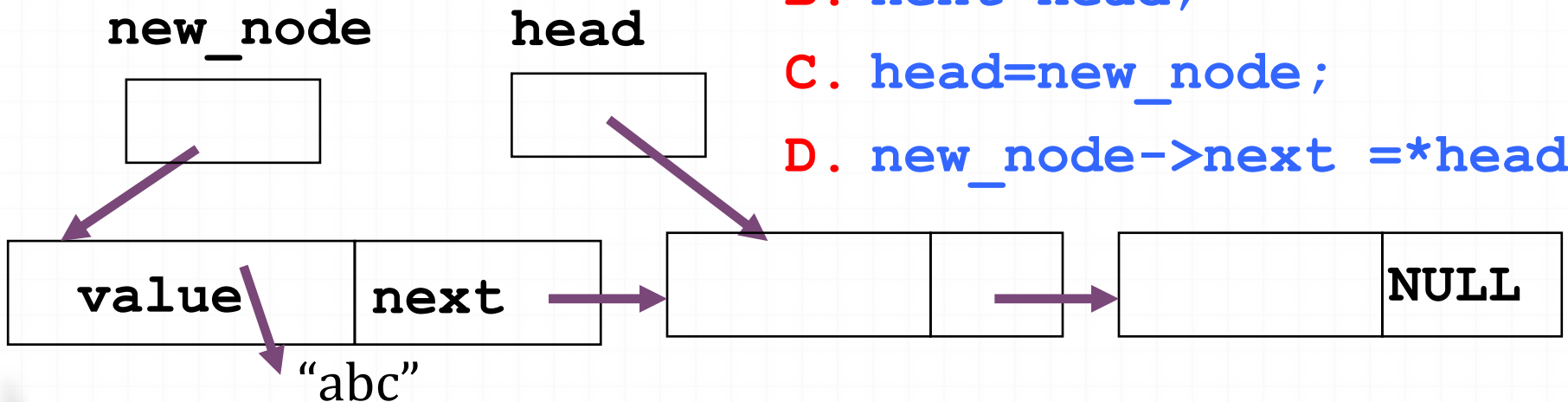
```
node *list_add(node* head, char *string){  
    //Step 1: Create a new node  
    node *new_node=(node*) malloc(sizeof(node));  
    //Step 2: Fill in its value  
    new_node->value = (char*) malloc(strlen(string)+1);  
    strcpy(new_node->value, string);  
    //Step 3: Link new_node to the head of the list  
    new_node->next =head;  
    return new_node; }
```

A. `new_node->next =head;`

B. `next=head;`

C. `head=new_node;`

D. `new_node->next =*head;`



Dangling pointers and memory leaks

- Dangling reference : Pointer points to a memory location that no longer exists
- Memory leaks (tardy free) Memory in heap that can no longer be accessed

Q: Which of the following functions returns a pointer to an object that may not exist in memory?

```
int * f1(int num) {  
    int *mem1 =(int *)malloc(num*sizeof(int)) ;  
    return(mem1) ;}
```

```
int * f2(int num) {  
    int mem2[num] ;  
    return(mem2) ;}
```

- A.** f1
- B.** f2
- C.** Both

Q: Which of the following functions returns a pointer to an object that may not exist in memory?

```
int * f1(int num) {  
    int *mem1 =(int *)malloc(num*sizeof(int)) ;  
    return(mem1) ;}
```

```
int * f2(int num) {  
    int mem2[num] ;  
    return(mem2) ;}
```

A. f1

B. f2 because mem2 is a local variable
created in the stack

C. Both

Which of the following is an example of a dangling pointer?

A.

```
void foo(int bytes) {  
    char *ch =(char *) malloc(bytes);  
    ....  
    free (ch);  
    ....}
```

B.

```
int * foo(int bytes) {  
    int i=14;  
    ....  
    return (&i);  
}
```

C.

```
char* foo(int bytes) {  
    char *ch =(char *) malloc(bytes);  
    ....  
    return (ch);  
}
```

D. A combination of the above

Which of the following is an example of a dangling pointer?

A.

```
void foo(int bytes) {  
    char *ch =(char *) malloc(bytes);  
    ....  
    free (ch);  
    ....}
```

B.

```
int * foo(int bytes) {  
    int i=14;  
    ....  
    return (&i);  
}
```

C.

```
char* foo(int bytes) {  
    char *ch =(char *) malloc(bytes);  
    ....  
    return (ch);  
}
```

D. A combination of the above

Memory Leak

- Is the following an example for a memory leak?

```
void foo (int bytes)
{
    int *p = (int *) malloc(bytes);
}
```

The C runtime environment

What does the executable contain?

- Instructions or “code”
- Data

What is data?

- Any information that instructions operate on
(objects in memory)

C Runtime Environment

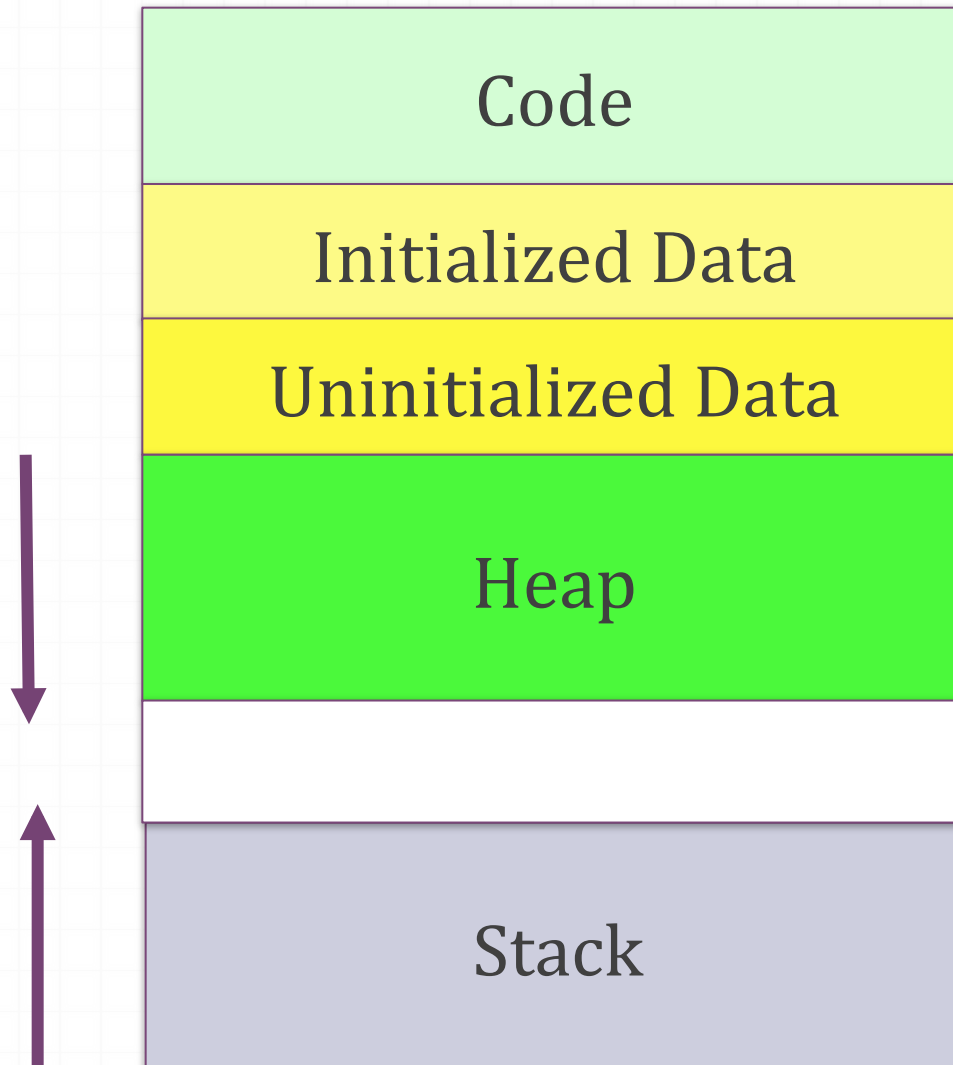
- “Code” (instructions in machine language)
- “Data” (initialized and uninitialized - static allocated)

Both code and data don't change in size

- “Heap” (for dynamically allocated data)
- “Stack” (for function local variables)

Heap and stack change in size as the program executes

C Runtime Environment



Data lifetime

- Lifetime: The interval between time of creation and end of existence of data

Possible lifetimes of data

1. Execution time of program
2. Time between explicit creation and explicit deletion
3. Execution time of a function (time between function call and function return)

Declarations and definitions

Scope and lifetime are often implicit but sometimes we have to use specific keywords:

- `static int a=0; /*Defines lifetime*/`
- `extern int a; /*Extends scope to multiple files*/`

Different types of data

What factor differentiates the following three 'categories' of data in a C program:

- Global and static variables
- Local variables
- Dynamic variables

- A. Scope
- B. Lifetime
- C. Representation

Different types of data

What factor differentiates the following three 'categories' of data in a C program:

- Global and static variables
- Local variables
- Dynamic variables

- A. Scope
- B. Lifetime
- C. Representation