GNG1106
Fundamentals of Engineering Computation

Instructor: **Hitham Jleed**

University of Ottawa

Fall 2023 ~

Pointer: Address-Typed Variable

Declaration

```
someType *x;
```

- `someType` can be any type, including structures.
- `x` is the name of declared variable.
- You are encouraged to think of "`someType *`" as the type of variable `x`.
- This line of declaration says the following:
  - `x` is declared to be a variable that takes an address as its value, and
  - the memory block with starting from address `x`, when referred to using `x`, should be interpreted as storing data of type `someType`.
- We may say that `x` is a "pointer to `someType`". E.g., if we declare "`int *x;`", we say `x` is a "pointer to int".

- The declaration of variables having a given type and the declaration of pointers to the same type can be combined in one statement.
- For example:

```
int myInt;
int *myPointer2int;
```

can be written as

```
int myInt, *myPointer2int;
```

- Like regular variables, when a pointer is declared, a block of memory is allocated for storing the value of the pointer.
  - Note: the value of a pointer is an address!
- If you declare a pointer `x`, you can use `sizeof(x)` to obtain the number of bytes that is required to store of the value of `x`.
  - On a 64-bit computer, it takes 8 bytes to store the value of a pointer.

Assignment of Pointers

- A pointer to a given type, say to type A, can be assigned an address of a memory block that has been designated for storing data of the same type, namely, type A. We then say that the pointer points to the memory block (or the variable).

Example

If we have declared "`int x, a[10], *ptr;`", the following are legal:

```
ptr=&x;
ptr=a;
ptr=a+3;
ptr=&a[5];
```

Example

If we have declared "`int x, a[10];`" and "`float *ptr;`", the following are illegal:

```
ptr=&x;
ptr=a;
ptr=a+3;
ptr=&a[5];
```

Declaring and Initializing a Pointer

- A pointer can be declared and assigned a value at the same time. For example, the following two ways are identical.

```
int x;
int *ptr=&x;
```

```
int x, *ptr=&x;
```

The NULL Pointer

- When a pointer is not assigned, it may have random values or have value NULL. This may depend on the computer system and the compiler.
- NULL is a symbolic constant defined in **stdio.h**. Its numerical value is 0.
- A pointer having value NULL means that it points to nowhere, i.e., that it does not point to any valid memory address.
- When we want to assure a pointer point to nowhere, we should explicitly assign value NULL to it.
- It is also encouraged to initialize pointers to NULL when they are declared.

Dereferencing a Pointer

- A pointer can be dereferenced using the operator “*”.
- Dereferencing a pointer works in exactly the same way as dereferencing an address.
- A dereferenced pointer is essentially a variable and can be used in exactly the same way as a variable.

```
int x=10, *ptr;
ptr=&x;
printf("%d\n", *ptr); // prints 10
*ptr = *ptr + 10;
printf("%d\n", *ptr); // prints 20
printf("%d\n", x); // prints 20
```

The Confusing Symbol *

- We have now learned three distinct uses of the symbol "*".
  - As the multiplication operator
  - In declaring a pointer (i.e., forming a "pointer type")
  - As the dereferencing operator
- When you see the symbol * in a code, ask yourself what it is meant there.

Dereferencing a Pointer That Points to an Array

- We have seen that a pointer can point to an array, or more precisely, points to the first element of an array.
- Example: in "`int a[3]={2, 4, 6}, *ptr=a;` "
- In this case, we can use pointer arithmetics (namely, address arithmetics) in combination with pointer dereferencing to access any element of the array.
- Following the above example line of code, the dereferenced addresses `*ptr`, `*(ptr+1)`, and `*(ptr+2)` will be precisely variables `a[0]`, `a[1]`, and `a[2]` respectively.
- It is also allowed to use the array notation to write the dereferenced addresses. That is, `*ptr`, `*(ptr+1)`, and `*(ptr+2)` can also be written respectively as `ptr[0]`, `ptr[1]`, and `ptr[2]`.

Allowed Operations on Pointers

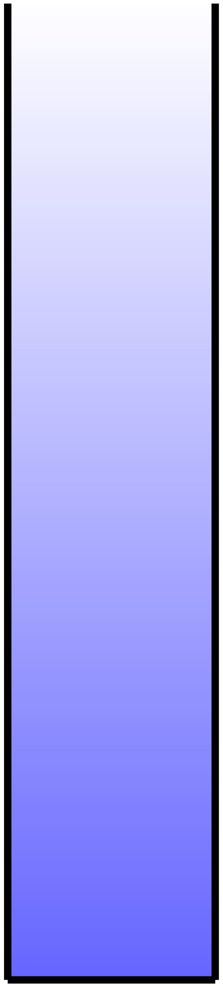
- Assignment
- Dereferencing
- Address arithmetics (addition/subtraction)
- Increment: `ptr++;` (the same as `ptr=ptr+1;`)
- Decrement: `ptr--;` (the same as `ptr=ptr-1;`)
- Comparison (with NULL or with other pointers)

Tracing A Code in Programming Model

Code Segment

```
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```

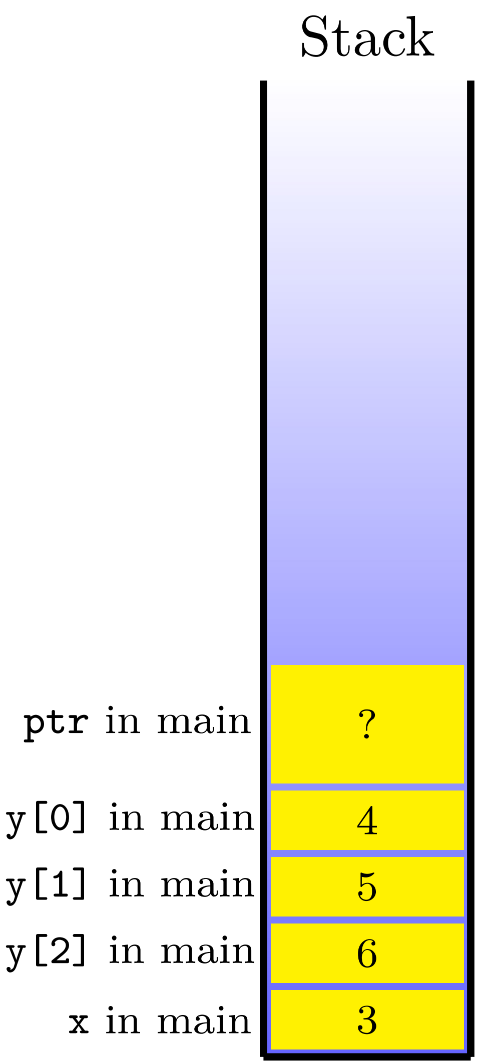
Stack



Tracing A Code in Programming Model: Step 1

Code Segment

```
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```



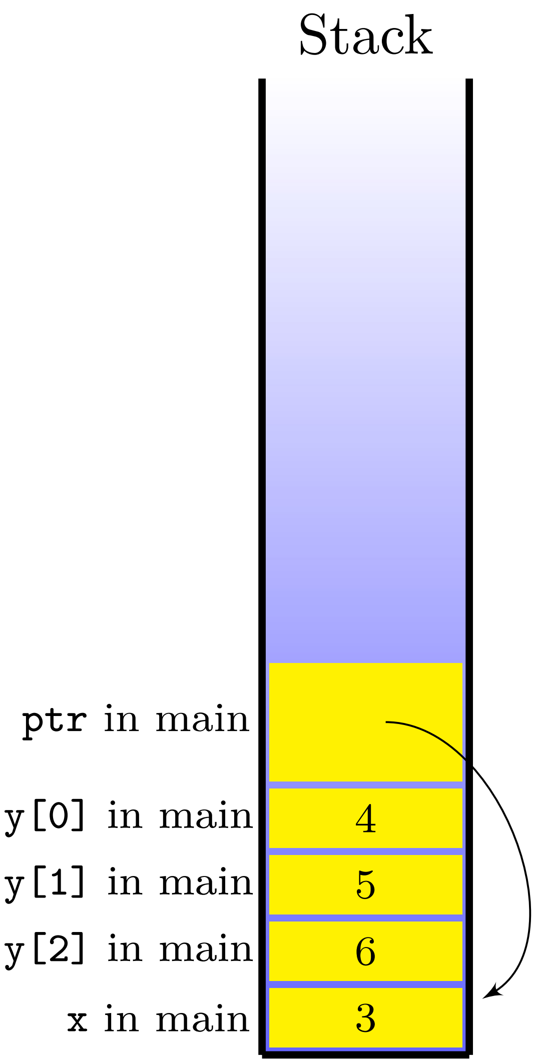
Stack

ptr in main	?
y[0] in main	4
y[1] in main	5
y[2] in main	6
x in main	3

Tracing A Code in Programming Model: Step 2

Code Segment

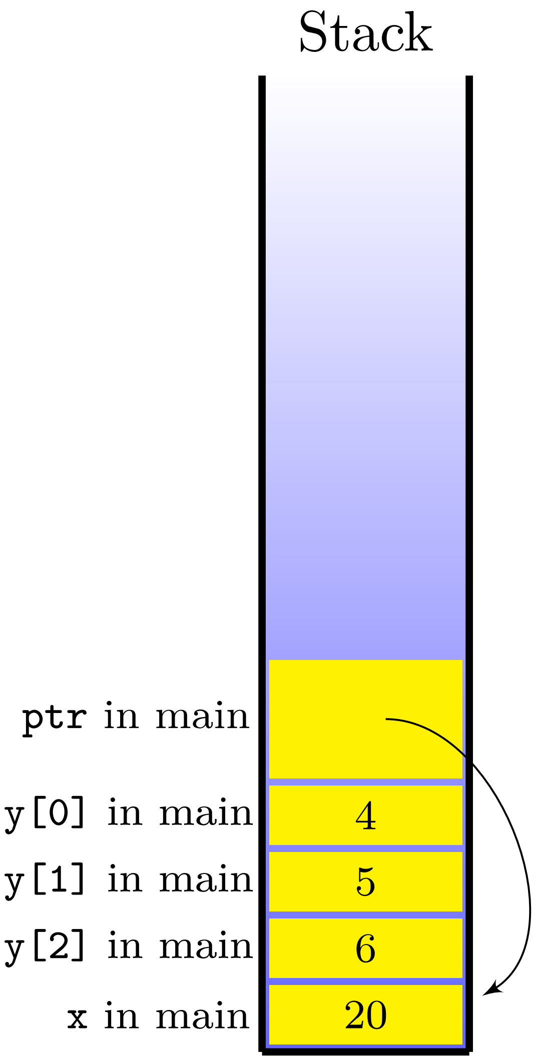
```
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```



Tracing A Code in Programming Model: Step 3

Code Segment

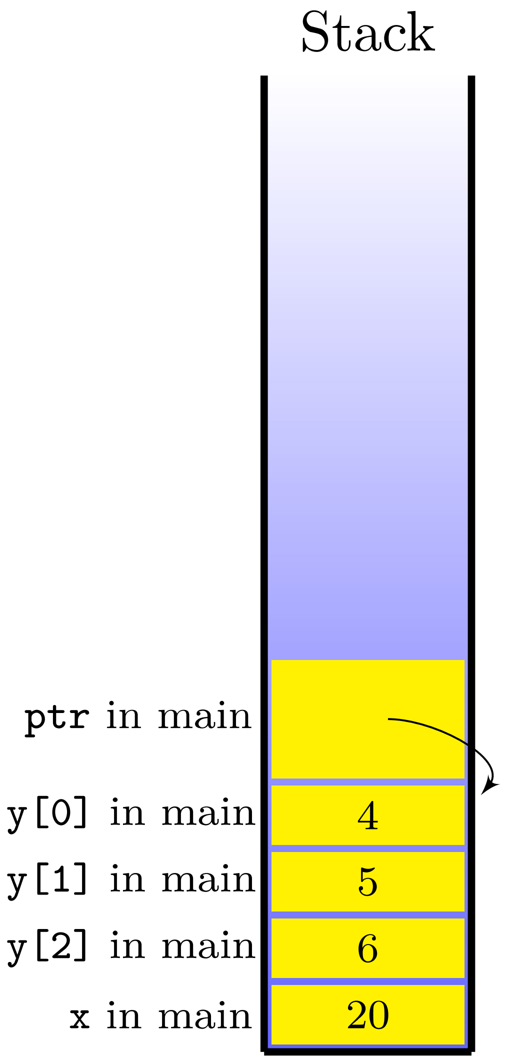
```
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```



Tracing A Code in Programming Model: Step 4

Code Segment

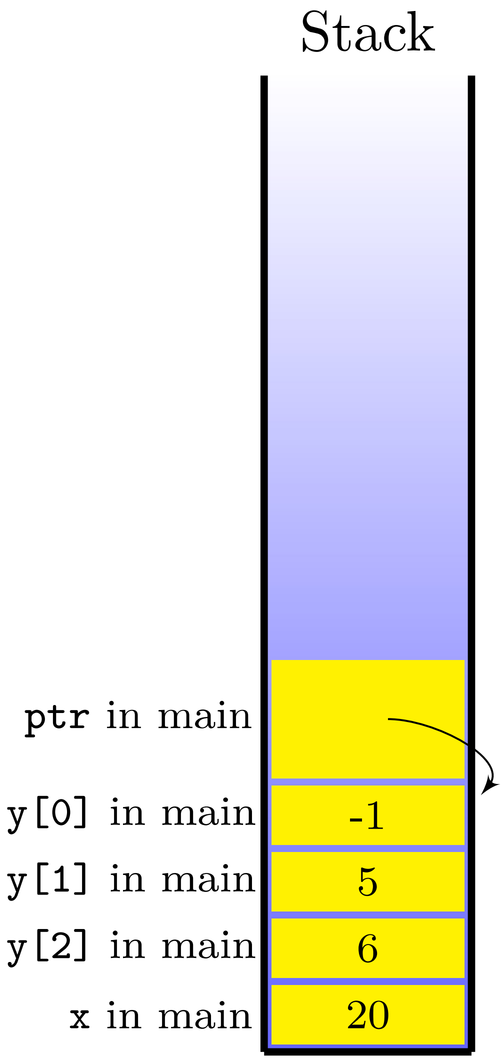
```
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```



Tracing A Code in Programming Model: Step 5

Code Segment

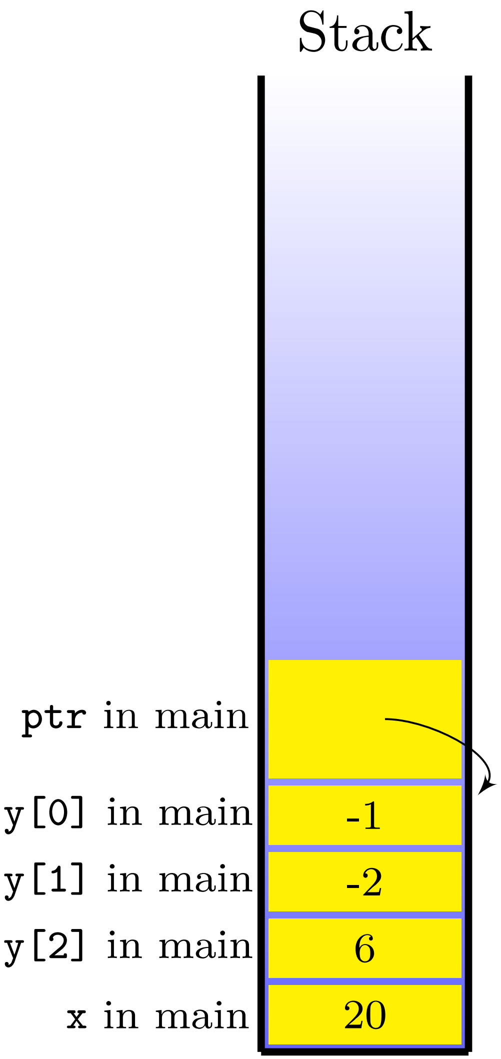
```
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```



Tracing A Code in Programming Model: Step 6

Code Segment

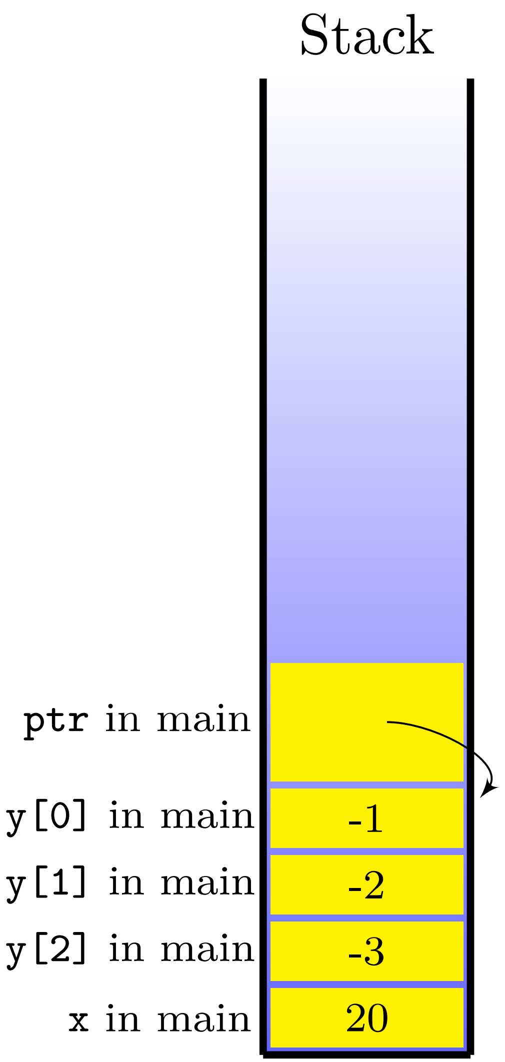
```
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```



Tracing A Code in Programming Model: Step 7

Code Segment

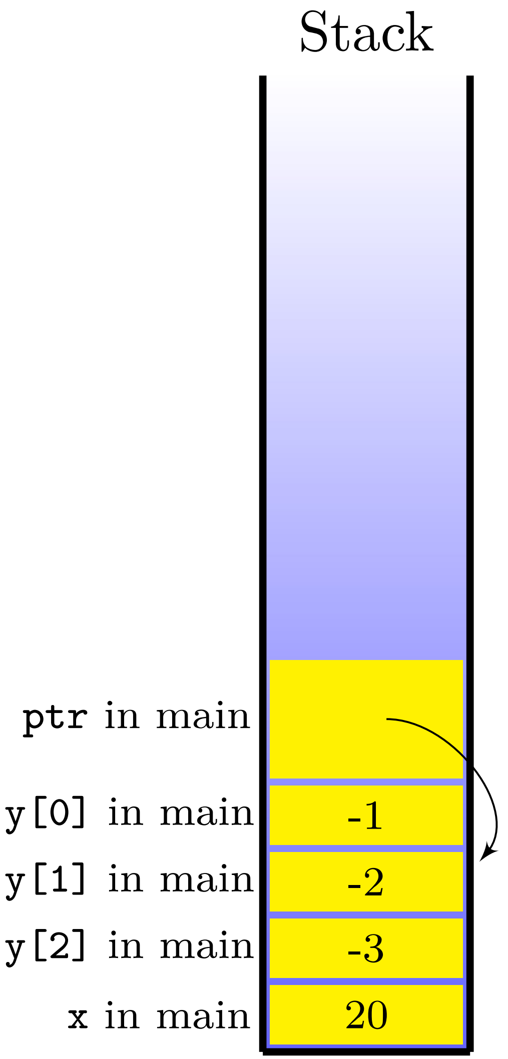
```
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```



Tracing A Code in Programming Model: Step 8

Code Segment

```
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```



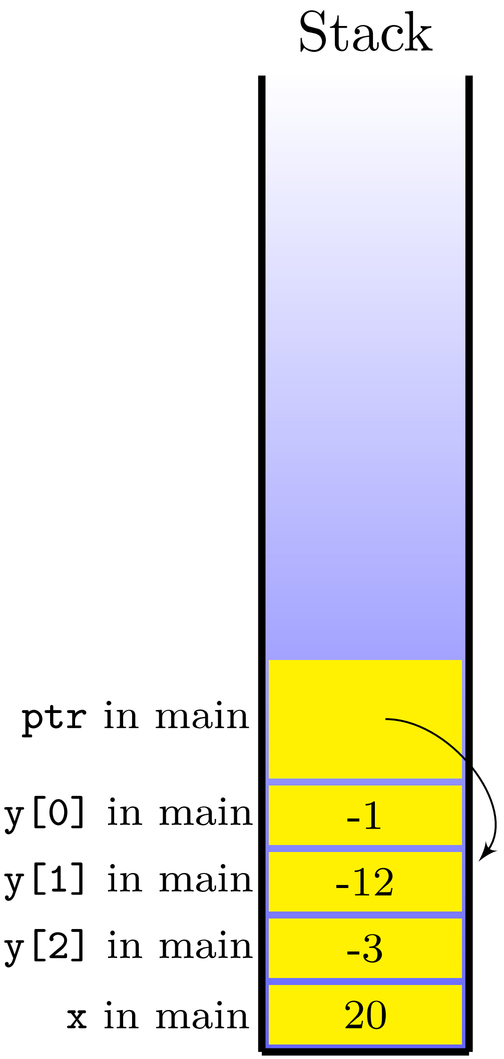
Stack

ptr in main	
y[0] in main	-1
y[1] in main	-2
y[2] in main	-3
x in main	20

Tracing A Code in Programming Model: Step 9

Code Segment

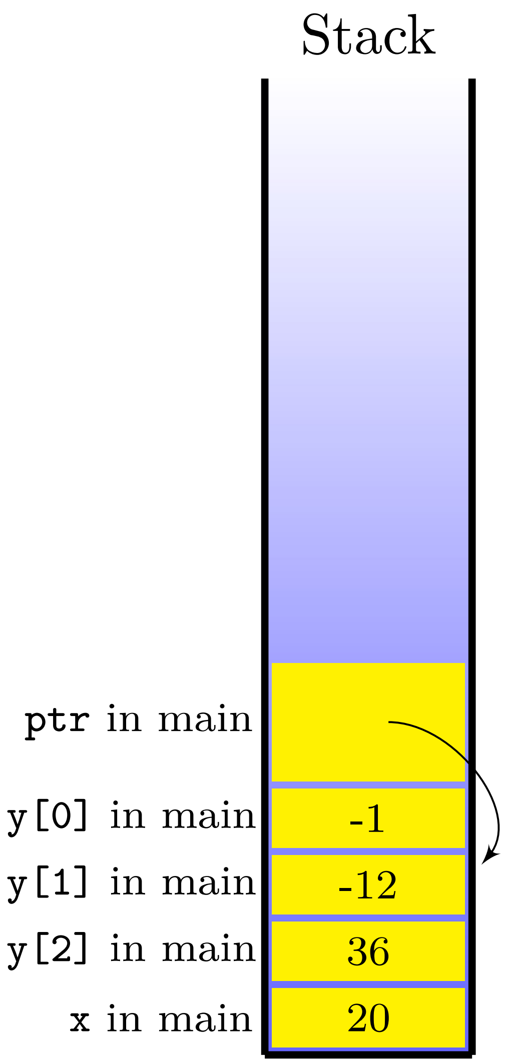
```
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```



Tracing A Code in Programming Model: Step 10

Code Segment

```
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```

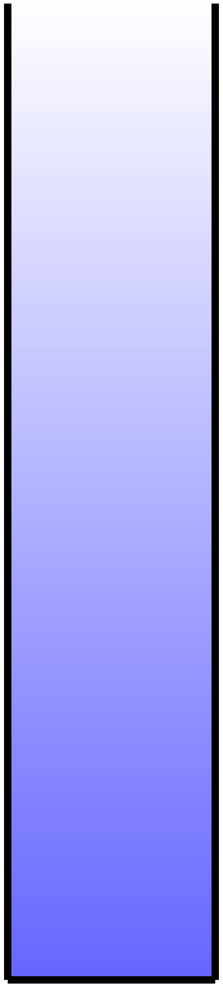


Tracing A Code in Programming Model: Step 11

Code Segment

```
#include <stdio.h>
int main()
{
   int x=3, y[3]={4, 5, 6};
   int *ptr;
   ptr=&x;
   *ptr=20;
   ptr=y;
   *ptr=-1;
   ptr[1]=-2;
   *(ptr+2)=-3;
   ptr++;
   ptr[0]=ptr[0]-10;
   *(ptr+1)=ptr[0]*ptr[1];
   return 0;
}
```

Stack



- A function can take pointers in its parameter list. E.g.

```
float myFunction(int *ptrA, float *ptrB)
{
  ...
}
```

- The prototype of such a function will then be

```
float myFunction(int *, float *)
```

- If a function has a pointer to type A as one of its parameter, when calling the function,
  - an address should be passed to the function as the value for the pointer, and
  - the memory at that address should have been designated (or interpreted) as storing data of type A.
- Passing an address value to a function is also referred to as "pass by reference", as opposed to passing a non-address value to a function, which is referred to as "pass by value".

- Like non-pointer variables, when a pointer is a parameter of a function, the pointer is a local variable of the function.
- When the function is called, memory is allocated for the pointer in the stack.
- When the function exits, the memories allocated for all of its local variables are released.

- Since the name of an array is an address, it can be legally passed to a function via a pointer.
- More precisely, suppose that a function's parameter includes a variable, say `X`, which is a pointer to type Y. Then the name of a Y-typed array can be passed to the function as the value of `X`. This is in fact one of the ways (and in fact the preferred way) of passing an array to a function.
- Of course, if a function is designed to analyze or process an array input, it needs to know the length of the array. So not only a pointer (serving to receive the array name, i.e. the address of the array) needs to be made as a parameter of the function, so does the length of the array.

- A function using a pointer to receive an array as input always has the following form

```
returnType myFunc(someType* ptrArray, int lengthArray)
{
    ...
}
```

- A function taking an array as input can also be designed to have an array as one of its parameters, which takes the following form:

```
returnType myFunc(someType ptrArray[], int
lengthArray)
{
    ...
}
```

  - Note the empty square brackets
  - Its prototype will be “`returnType myFunc(someType [], int );`”

Highlight

Passing an array into a function either via a pointer or via an array in the function's parameter list is "pass by reference". That is:

- The array passed to the function is not copied inside the function.
- Only the address of the passed array is copied inside the function.
- Any modification of the array content inside the function directly applies to the content of of the array in the calling function! This is very different from "pass by value".

- A function is also allowed to take a 2D array as an input parameter. But we will not introduce this. Instead, we will soon (hopefully) introduce a more elegant alternatively, namely, letting a function take a "pointer to a pointer" as its parameter.

Write a Program ...

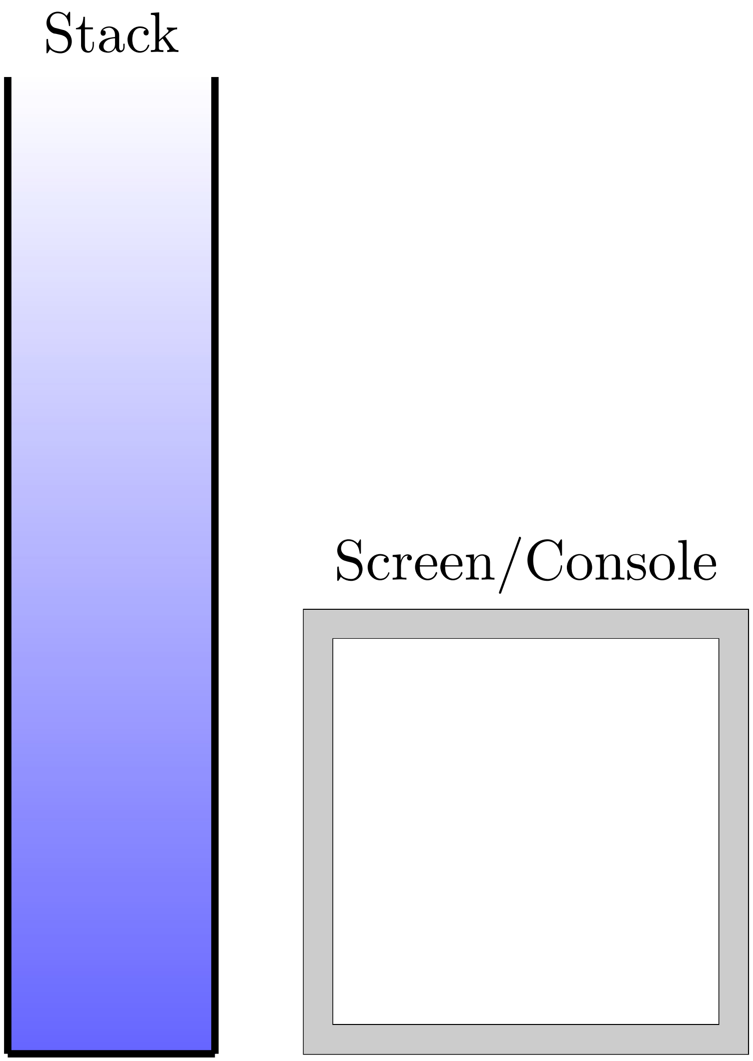
- Write a program that reads two numbers from keyboards into variables `a` and `b`, and then swap them (namely, make variable `a` contains the second entered value and variable `b` contain the first).
- Change the program so that swapping two numbers is implemented as a function.

Coding Demonstration

Tracing the Program with Swap Function

Code Segment

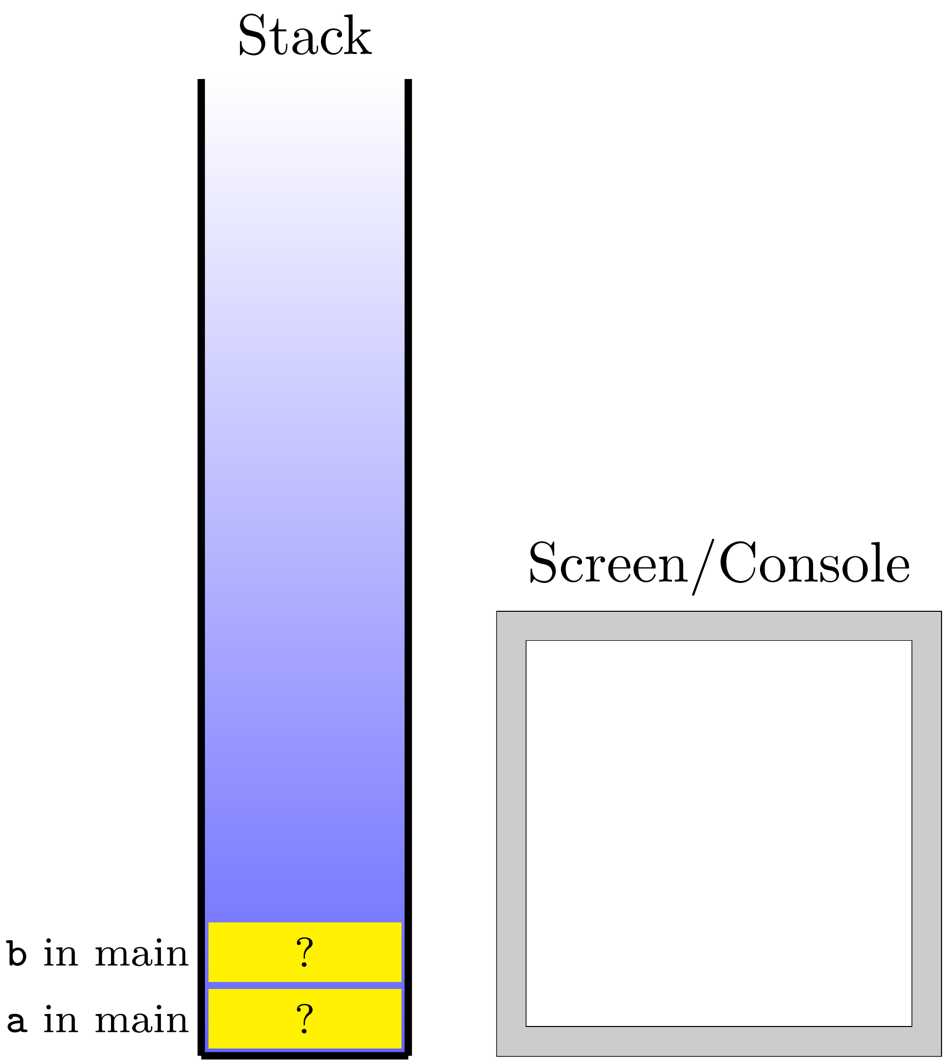
```
#include <stdio.h>
void swap(int *ptrA, int *ptrB)
{
   int c;
   c=*ptrA;
   *ptrA=*ptrB;
   *ptrB=c;
}
int main()
{
   int a, b;
   printf("enter a and b\n");
   scanf("%d%d", &a, &b);
   swap(&a, &b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```



Tracing the Program with Swap Function: Step 1

Code Segment

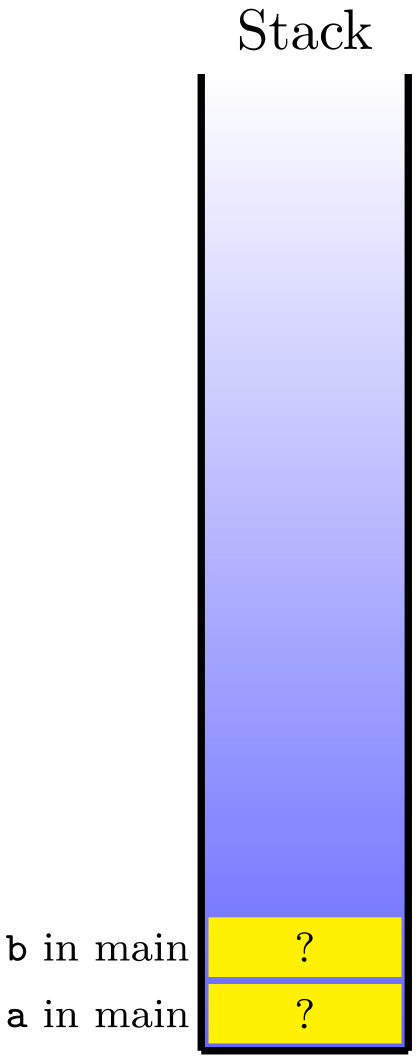
```
#include <stdio.h>
void swap(int *ptrA, int *ptrB)
{
   int c;
   c=*ptrA;
   *ptrA=*ptrB;
   *ptrB=c;
}
int main()
{
   int a, b;
   printf("enter a and b\n");
   scanf("%d%d", &a, &b);
   swap(&a, &b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```



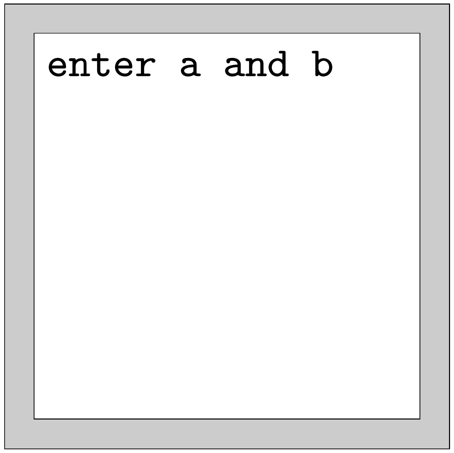
Tracing the Program with Swap Function: Step 2

Code Segment

```
#include <stdio.h>
void swap(int *ptrA, int *ptrB)
{
   int c;
   c=*ptrA;
   *ptrA=*ptrB;
   *ptrB=c;
}
int main()
{
   int a, b;
   printf("enter a and b\n");
   scanf("%d%d", &a, &b);
   swap(&a, &b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```



Screen/Console

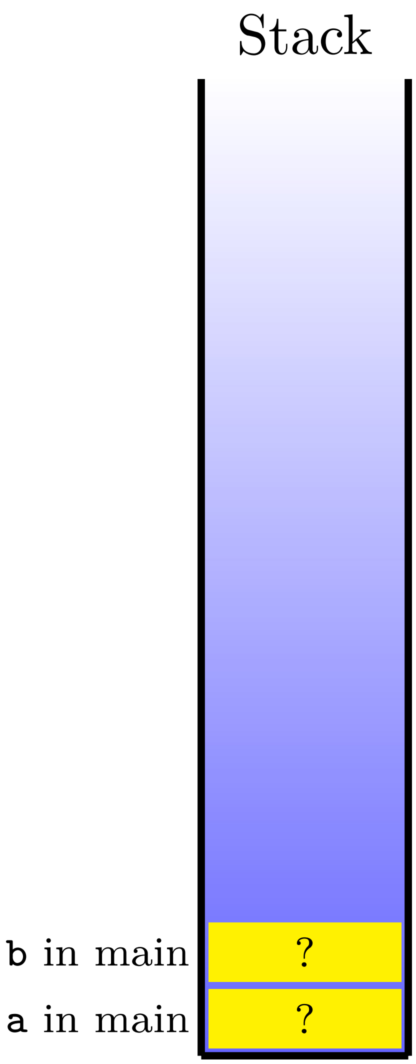


```
enter a and b
```

Tracing the Program with Swap Function: Step 3

Code Segment

```
#include <stdio.h>
void swap(int *ptrA, int *ptrB)
{
   int c;
   c=*ptrA;
   *ptrA=*ptrB;
   *ptrB=c;
}
int main()
{
   int a, b;
   printf("enter a and b\n");
   scanf("%d%d", &a, &b);
   swap(&a, &b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```



Screen/Console

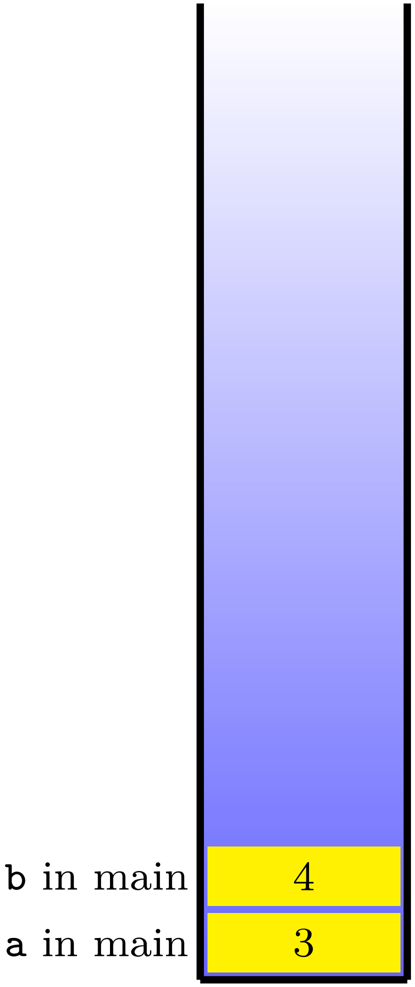
```
enter a and b
```

Tracing the Program with Swap Function: Step 4

Code Segment

```
#include <stdio.h>
void swap(int *ptrA, int *ptrB)
{
   int c;
   c=*ptrA;
   *ptrA=*ptrB;
   *ptrB=c;
}
int main()
{
   int a, b;
   printf("enter a and b\n");
   scanf("%d%d", &a, &b);
   swap(&a, &b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```

Stack



Screen/Console

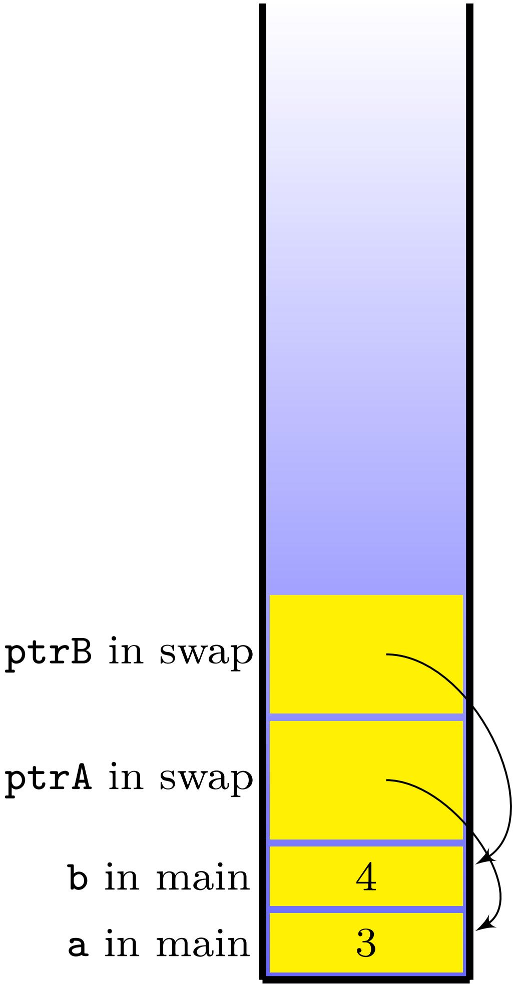
```
enter a and b
3
4
```

Tracing the Program with Swap Function: Step 5

Code Segment

```c
#include <stdio.h>
void swap(int *ptrA, int *ptrB)
{
   int c;
   c=*ptrA;
   *ptrA=*ptrB;
   *ptrB=c;
}
int main()
{
   int a, b;
   printf("enter a and b\n");
   scanf("%d%d", &a, &b);
   swap(&a, &b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```

Stack



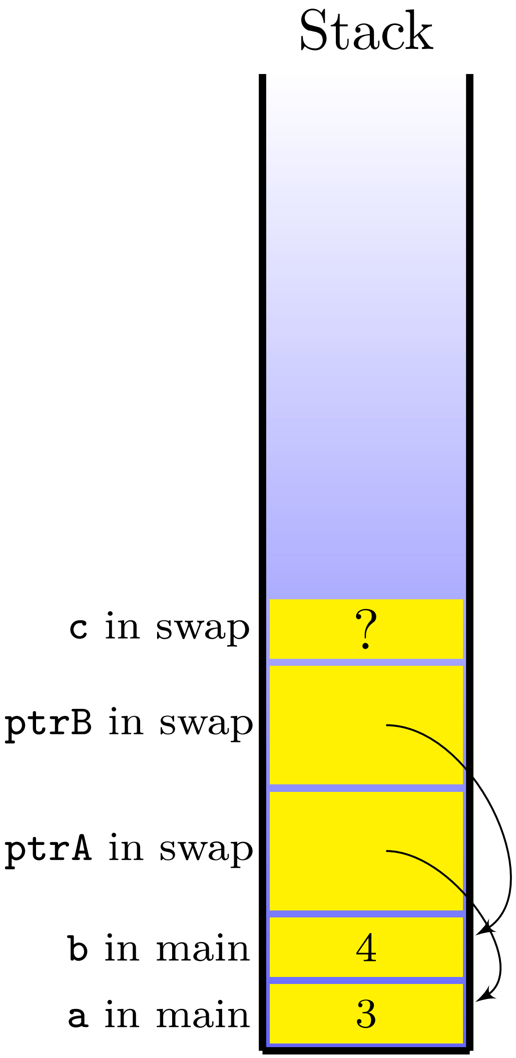
Screen/Console

```
enter a and b
3
4
```

Tracing the Program with Swap Function: Step 6

Code Segment

```
#include <stdio.h>
void swap(int *ptrA, int *ptrB)
{
   int c;
   c=*ptrA;
   *ptrA=*ptrB;
   *ptrB=c;
}
int main()
{
   int a, b;
   printf("enter a and b\n");
   scanf("%d%d", &a, &b);
   swap(&a, &b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```



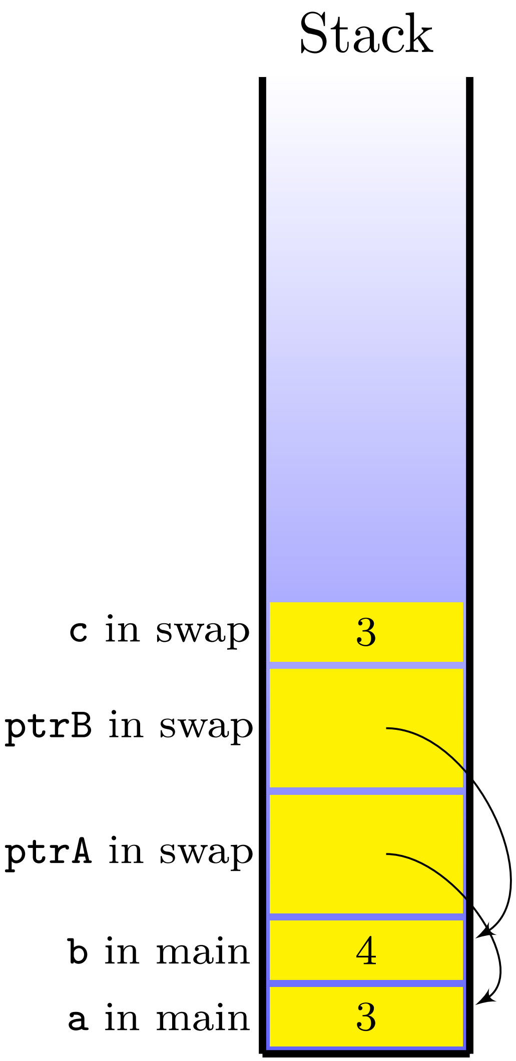
Screen/Console

```
enter a and b
3
4
```

Tracing the Program with Swap Function: Step 7

Code Segment

```
#include <stdio.h>
void swap(int *ptrA, int *ptrB)
{
   int c;
   c=*ptrA;
   *ptrA=*ptrB;
   *ptrB=c;
}
int main()
{
   int a, b;
   printf("enter a and b\n");
   scanf("%d%d", &a, &b);
   swap(&a, &b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```



Screen/Console

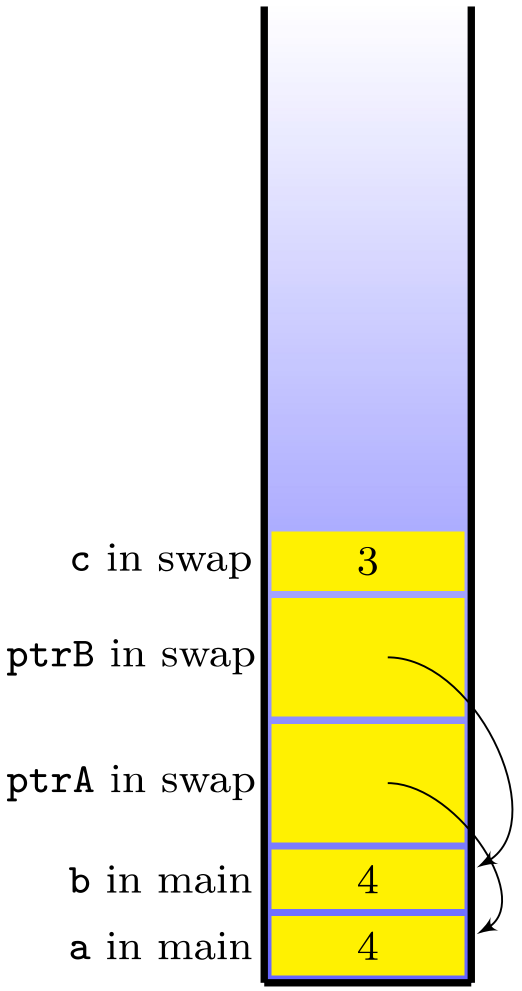
```
enter a and b
3
4
```

Tracing the Program with Swap Function: Step 8

Code Segment

```
#include <stdio.h>
void swap(int *ptrA, int *ptrB)
{
   int c;
   c=*ptrA;
   *ptrA=*ptrB;
   *ptrB=c;
}
int main()
{
   int a, b;
   printf("enter a and b\n");
   scanf("%d%d", &a, &b);
   swap(&a, &b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```

Stack



Screen/Console

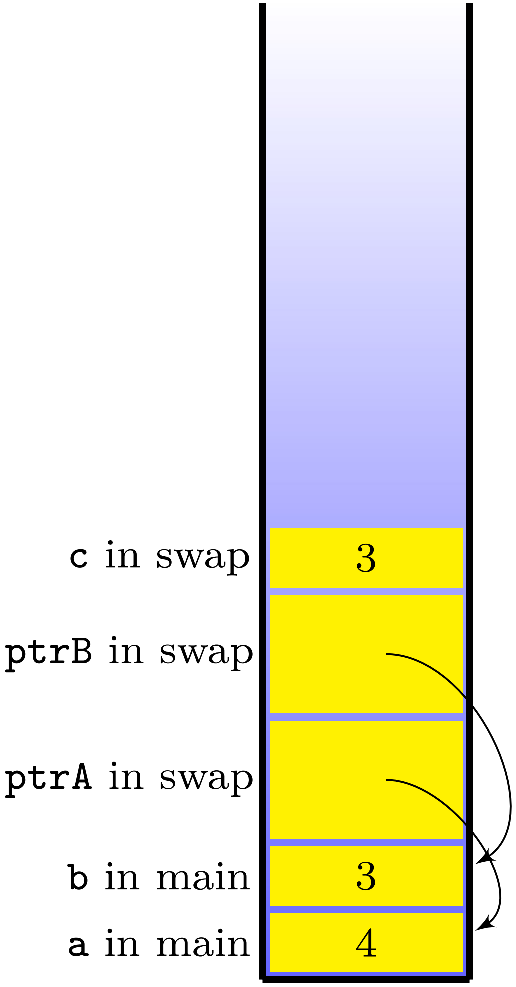
```
enter a and b
3
4
```

Tracing the Program with Swap Function: Step 9

Code Segment

```
#include <stdio.h>
void swap(int *ptrA, int *ptrB)
{
   int c;
   c=*ptrA;
   *ptrA=*ptrB;
   *ptrB=c;
}
int main()
{
   int a, b;
   printf("enter a and b\n");
   scanf("%d%d", &a, &b);
   swap(&a, &b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```

Stack



Screen/Console

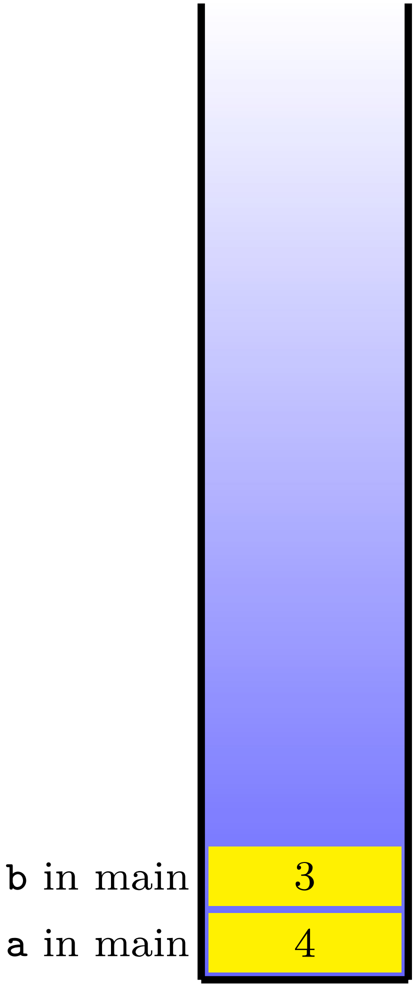
```
enter a and b
3
4
```

Tracing the Program with Swap Function: Step 10

Code Segment

```c
#include <stdio.h>
void swap(int *ptrA, int *ptrB)
{
   int c;
   c=*ptrA;
   *ptrA=*ptrB;
   *ptrB=c;
}
int main()
{
   int a, b;
   printf("enter a and b\n");
   scanf("%d%d", &a, &b);
   swap(&a, &b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```

Stack



Screen/Console

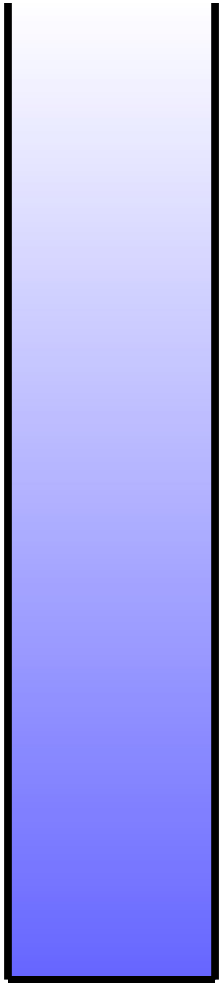
```
enter a and b
3
4
a=4, b=3
```

Tracing the Program with Swap Function: Step 11

Code Segment

```
#include <stdio.h>
void swap(int *ptrA, int *ptrB)
{
   int c;
   c=*ptrA;
   *ptrA=*ptrB;
   *ptrB=c;
}
int main()
{
   int a, b;
   printf("enter a and b\n");
   scanf("%d%d", &a, &b);
   swap(&a, &b);
   printf("a=%d, b=%d\n", a, b);
   return 0;
}
```

Stack



Screen/Console

```
enter a and b
3
4
a=4, b=3
```

In-Class Exercise: