

HIGH-LEVEL PROGRAMMING I

Intro to Pointers

by Prasanna Ghali

Addresses and Pointers (1 / 4)

2

- Main memory divided into bytes with each byte storing 8 bits of information



- Each byte has unique address

Addresses and Pointers (2/4)

3

- If there are n bytes of memory, addresses are numbers that range from 0 to $n-1$

Address	Contents
0	10101010
1	01001001
2	00011110
3	10000101
4	01110000
	⋮
$n-1$	10101100

Arbitrary contents of each byte in binary

Addresses and Pointers (3/4)

4

- When C program is executed, variables are assigned to memory locations
- Address of 1st byte of assigned storage is variable's address

100 is address of variable **x**.
Address 100 is used here for illustrative purposes.

Actual addresses of variables is not normally known nor should be of interest to programmers.

Address

Contents

100

101

102

103

⋮

⋮

x is variable
of type **int**

Addresses and Pointers (4/4)

5

- Addresses can be stored in special variables called *pointer* variables
- When address of variable *i* is stored in pointer variable *pi*, we say “*pi* points to *i*”

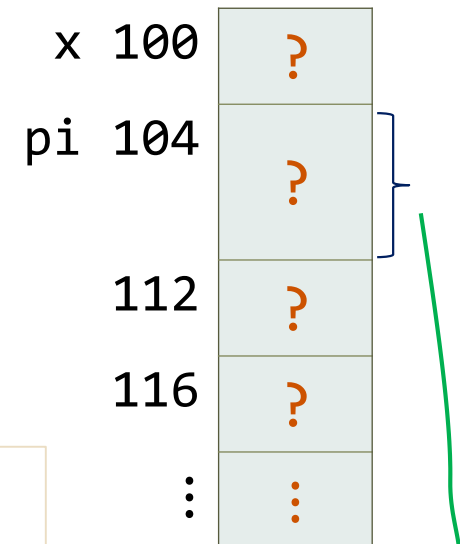


Declaring pointers (1 / 3)

6

- When pointer variable is declared, its name must be preceded by asterisk

```
int x;  
/*  
pi is pointer variable capable of  
pointing to objects of type int  
*/  
int *pi;
```



`sizeof(pi)` is 8 bytes on 64-bit machines and 4 bytes on 32-bit machines

Declaring pointers (2/3)

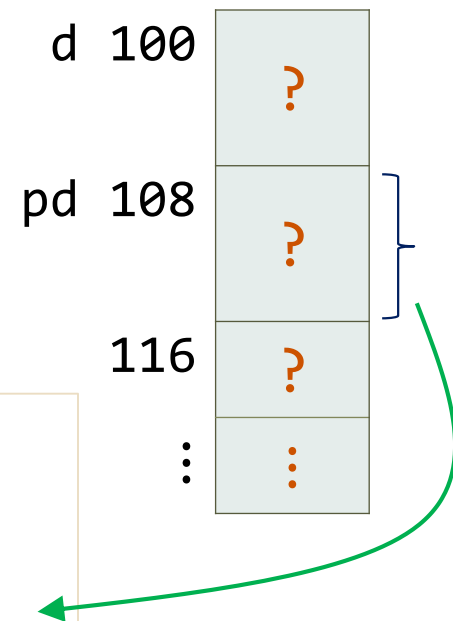
7

- Every pointer requires **8** bytes of storage on 64-bit machines

```
double d;  
  
// pd is pointer variable capable of  
// pointing to objects of type double  
double *pd;
```

`sizeof(pd)` is **8** bytes.

All pointer variables require **8** bytes of storage.
Only pointer type in declaration will determine how pointer is used.



Declaring pointers (3/3)

8

- Use of whitespace around asterisk is irrelevant

```
int* pw;  
int *px;  
int * py;  
int*pz;
```

These 4 declarations are equivalent

Read pointer declarations from right-to-left: pw (or px, py, pz) is *pointer to* int

- Pointer variables can appear in declarations with other variables

```
int x = 1, *pi, y = 2;
```


Address and Indirection operators

9

- How to make pointer variable point to specific variable?
- How to read from or write to memory pointed to?

Address operator (1 / 3)

10

- Memory storage address of variable (and function) can be referenced using address operator &

```
int value;  
scanf("%d", &value);
```

```
int x = 1, y = 2;  
  
printf("x = %d; address of x = %p\n", x, &x);  
printf("y = %d; address of y = %p\n", y, &y);
```



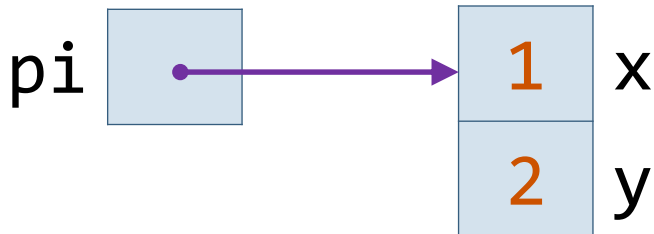
format specifier to print address
contained by pointer variable

Address operator (2/3)

11

- Address of operand returned by address operator can be assigned to pointer variable of appropriate type

```
int x = 1, y = 2, *pi = &x;
```



x	100	1
y	104	2
pi	108	100
	116	?
	⋮	⋮

Address operator (3/3)

12

- C/C++ are typed languages; pointers can only point to objects of *declared* type

```
// Error!!  
// pi can only point to int variables  
double d = 1.1;  
int x = 1, y = 2, *pi = &d;
```

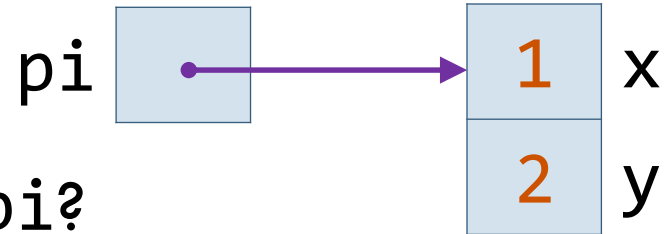
There is nothing inherent to pointer's value that suggests what type of data it is referencing or whether its contents are valid. Only pointer type in declaration will determine how pointer is used. Compiler steps in to complain when pointer is used incorrectly.

Indirection operator (1/3)

13

- So far, we've used & operator to make "pi point to x"

```
int x = 1, y = 2, *pi = &x;
```



- How do we reference x using pi?
- Indirection operator * references object that pointer points to

pi has type int *

- As long as pi points to x, *pi is *alias* for x

- *pi *means* x

*pi has type int

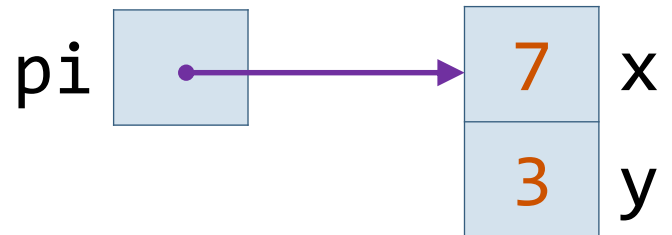
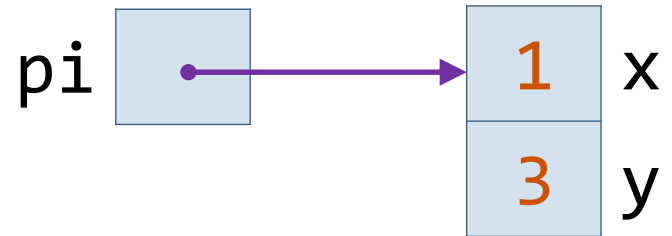
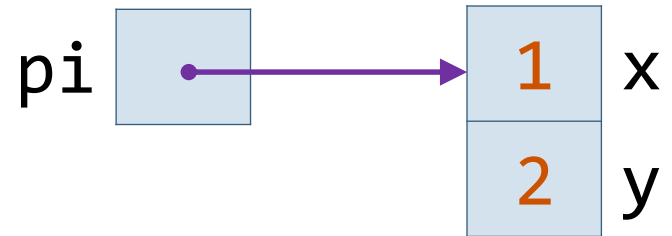
- Anywhere x can be used, *pi can be used!!!
- Changing value of *pi means changing value of x

Indirection operator (2/3)

14

- Since `pi` points to `x`, `*pi` is alias for `x`

```
int x = 1, y = 2, *pi = &x;  
y = *pi + 2;  
*pi = y + 4;
```



Indirection operator (3/3)

15

- Beware: reading from and writing through uninitialized pointer is meaningless!!!

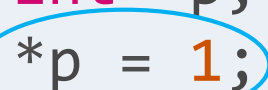
```
int *p; // uninitialized  
printf("*p: %d", *p); // WRONG
```



Applying indirection operator to uninitialized pointer variable causes undefined behavior

Dereferencing uninitialized pointer is catastrophic!!!

```
int *p; // uninitialized  
*p = 1; // VERY WRONG
```



Meaning of * symbol

16

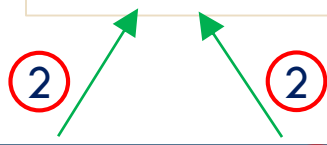
Part of type specifier of `pi` - should be read as “pointer to”

```
int x = 1, y = 2, *pi = &x;
```



Unary dereference operator – should be read as “follow the pointer”

```
*pi = *pi * y;
```



binary multiplication operator

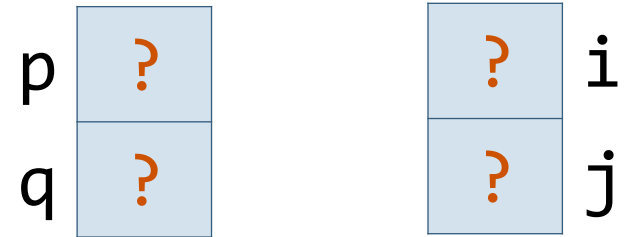


Pointer assignment (1 / 5)

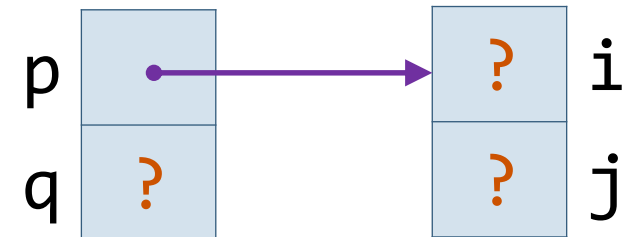
17

- C/C++ allow use of assignment operator to copy pointers of same type

```
int i, j, *p, *q;
```

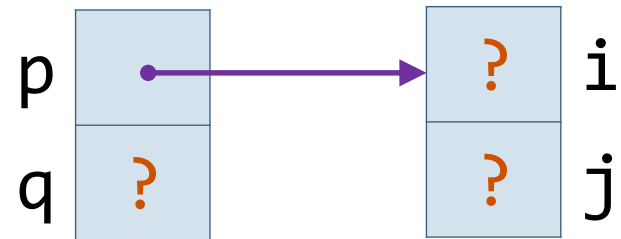


- Consider pointer assignment: `p = &i;`



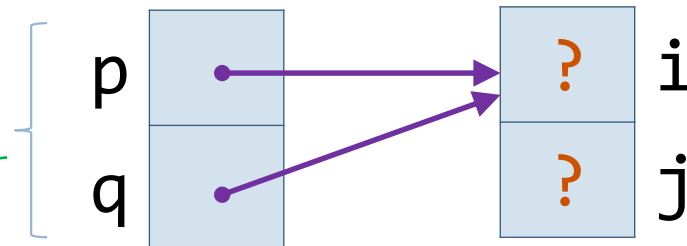
Pointer assignment (2/5)

18



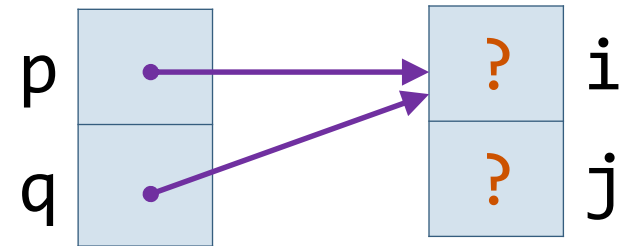
- This pointer assignment makes *q* point to same variable as *p*: `q = p;`

Any number of pointer variables can point to same object!!!



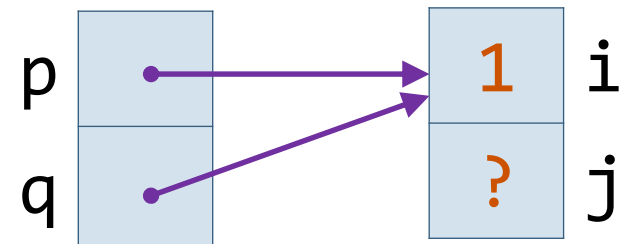
Pointer assignment (3/5)

19

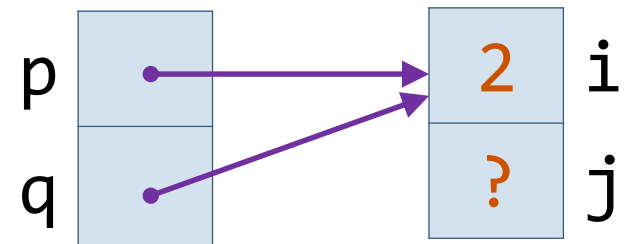


- If *p* and *q* both point to *i*, we can change *i* by assigning new value to either `*p` or `*q`:

`*p = 1;`



`*q = 2;`



Pointer assignment (4/5)

20

□ Given: `int i = 1, j = 2, *p = &i, *q = &j;`

□ Common mistake is to confuse between expressions:

`*q = *p`

Result of evaluation
is type `int`

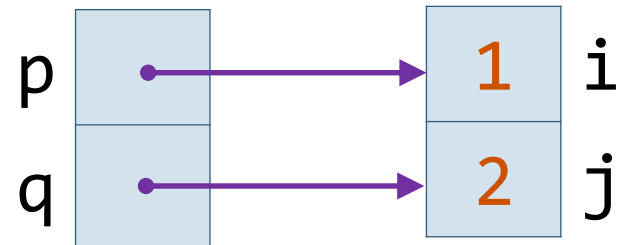
`q = p`

Result of evaluation
is type `int*`

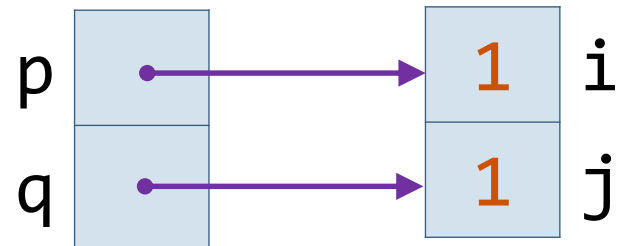
Pointer assignment (5/5)

21

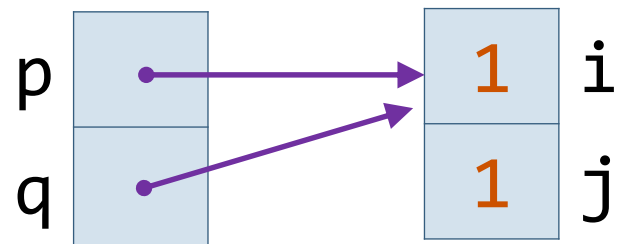
```
int i=1, j=2;  
int *p = &i;  
int *q = &j;
```



```
*q = *p;
```



```
q = p;
```



Pointer to `void` (2/2)

22

- `void` means nothing so pointer variable of type `void*` cannot be dereferenced
- Think of variable of type `void*` as general-purpose pointer that can be used as placeholder for addresses
- Can assign address of any pointer type to `void*` type
- Can assign address in `void*` to any pointer type
- Visualizer

NULL pointer

23

- Only literal integral value that a pointer can be initialized with or assigned to is \emptyset
- Standard library provides macro **NULL** to indicate null pointer

```
#define NULL ((void*) $\emptyset$ )
```

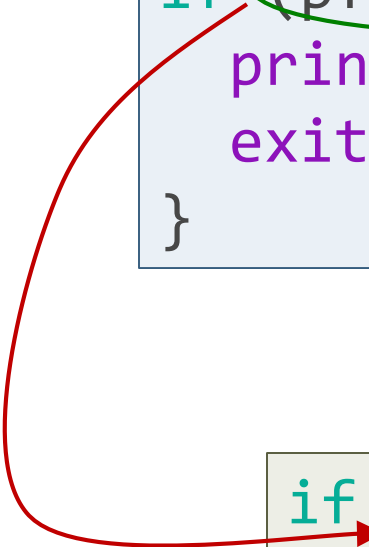
Can write either \emptyset or **NULL**.
But using **NULL** makes it apparent to reader that pointer expression is involved.

C guarantees that it will never store variable or function at address \emptyset .
Thus, \emptyset can be used as sentinel.

Using **NULL** pointer

24

```
FILE *pf = fopen("data.txt", "r");  
if (pf == NULL) {  
    printf("Unable to open data.txt!!!\n");  
    exit(EXIT_FAILURE);  
}
```



```
if (!pf) {  
    // pf is false (or NULL)  
}
```


Why pointers?

25

- ❑ Creating fast and efficient code
- ❑ Implementing data structures such as linked lists, trees and graphs
- ❑ Supporting dynamic memory allocation
- ❑ Making expressions compact and more succinct
- ❑ Provide ability to efficiently pass data structures between functions with minimum overhead

Functions: Pass-by-value convention

26

- Function call operator evaluates *arguments*
 - ▣ *Arguments* are expressions that evaluate to value with certain type
- Next, function call operator *invokes* function
 - ▣ Function *parameters* initialized by arguments
 - ▣ Function itself will not know anything about objects defined in caller's scope and cannot therefore make changes to objects in caller's scope
- Exception to rule?

Passing data by value (1 / 9)

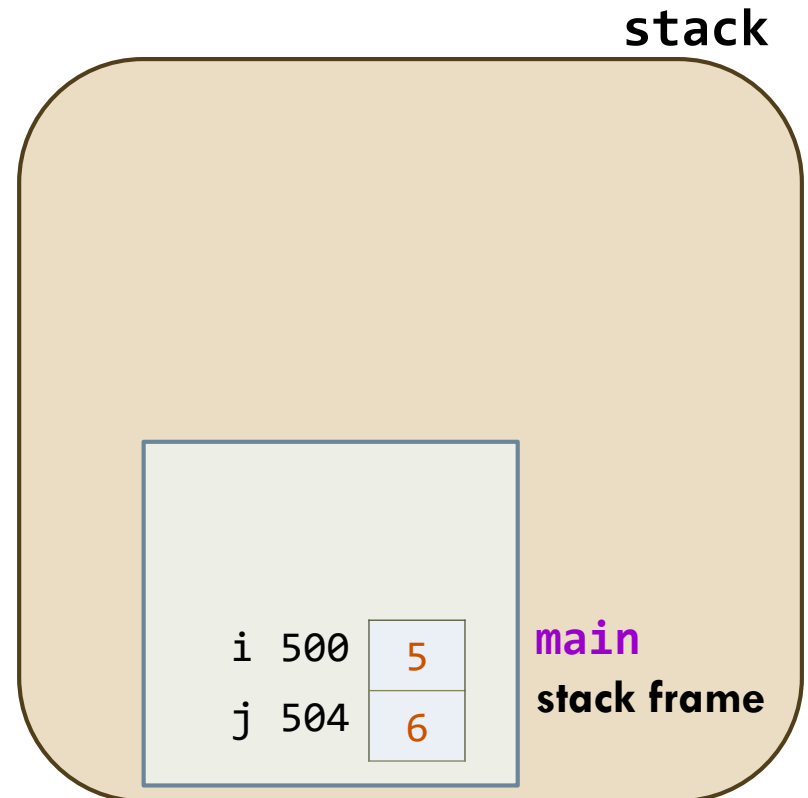
27

- Variables `i` and `j` are defined in `main`'s stack frame

```
#include <stdio.h>

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

int main(void) {
    → int i = 5, j = 6;
    swap(i, j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data by value (2/9)

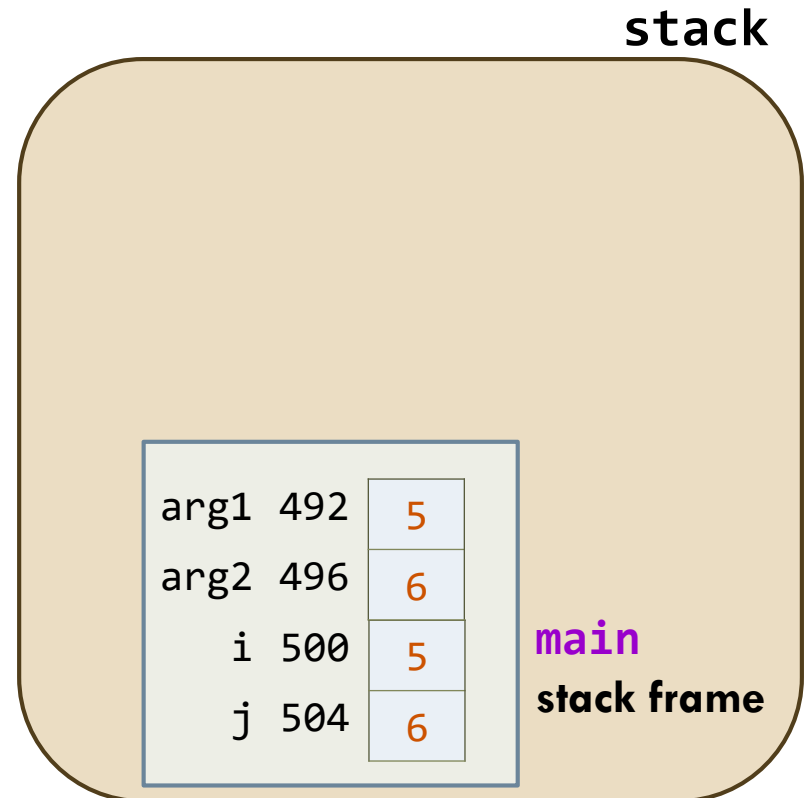
28

- Arguments of **swap** function call operator are evaluated

```
#include <stdio.h>

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

int main(void) {
    int i = 5, j = 6;
    → swap(i, j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data by value (3/9)

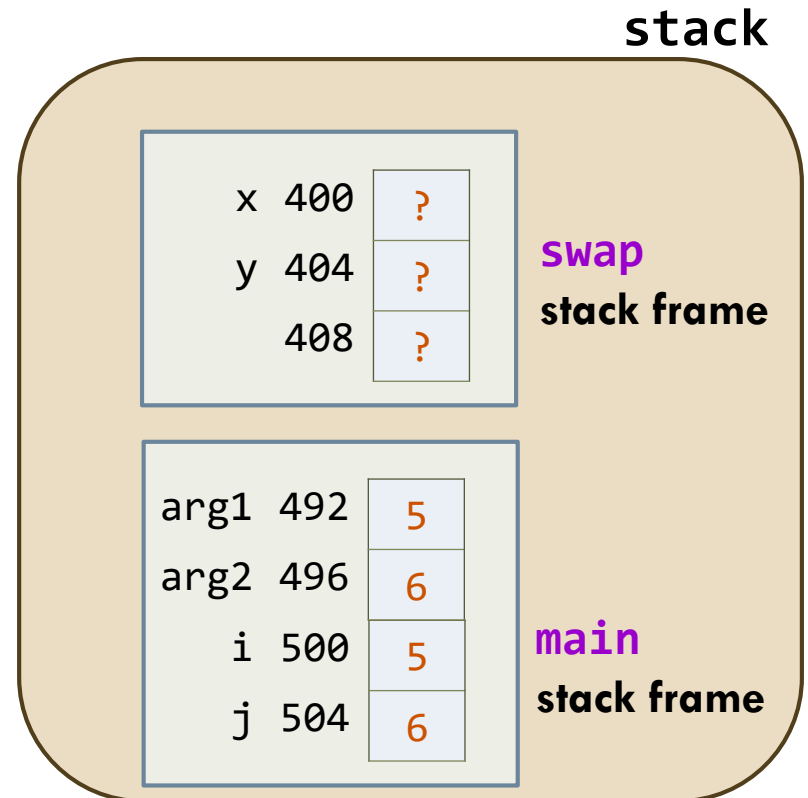
29

- Stack frame of function **swap** is constructed

```
#include <stdio.h>

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

int main(void) {
    int i = 5, j = 6;
    → swap(i, j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data by value (4/9)

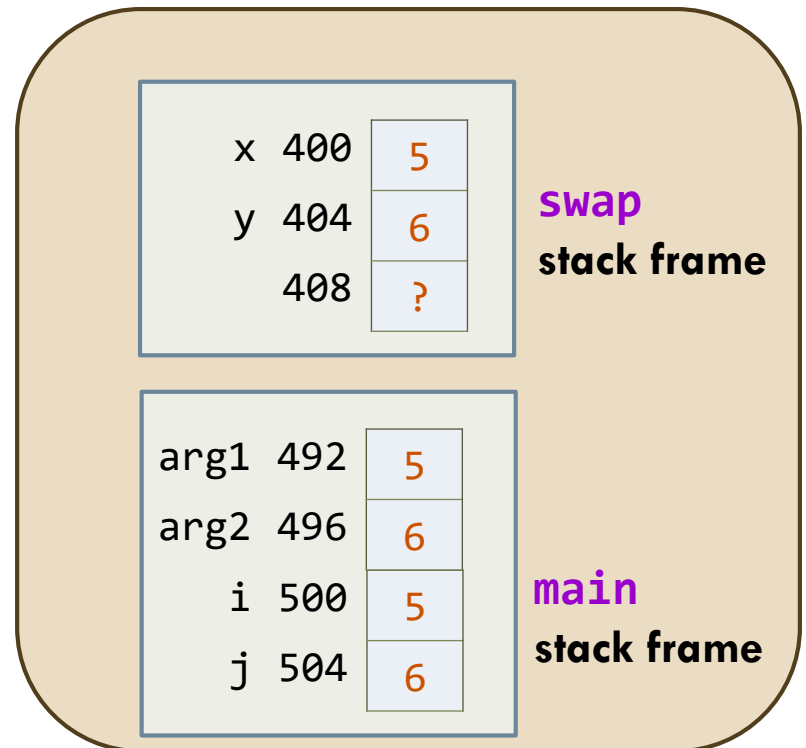
30

- Parameters of function **swap** are initialized

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

```
→ void swap(int x, int y) {  
    int tmp = x;  
    x = y;  
    y = tmp;  
}  
  
int main(void) {  
    int i = 5, j = 6;  
    swap(i, j);  
    printf("%d | %d\n", i, j);  
    return 0;  
}
```

stack



Passing data by value (5/9)

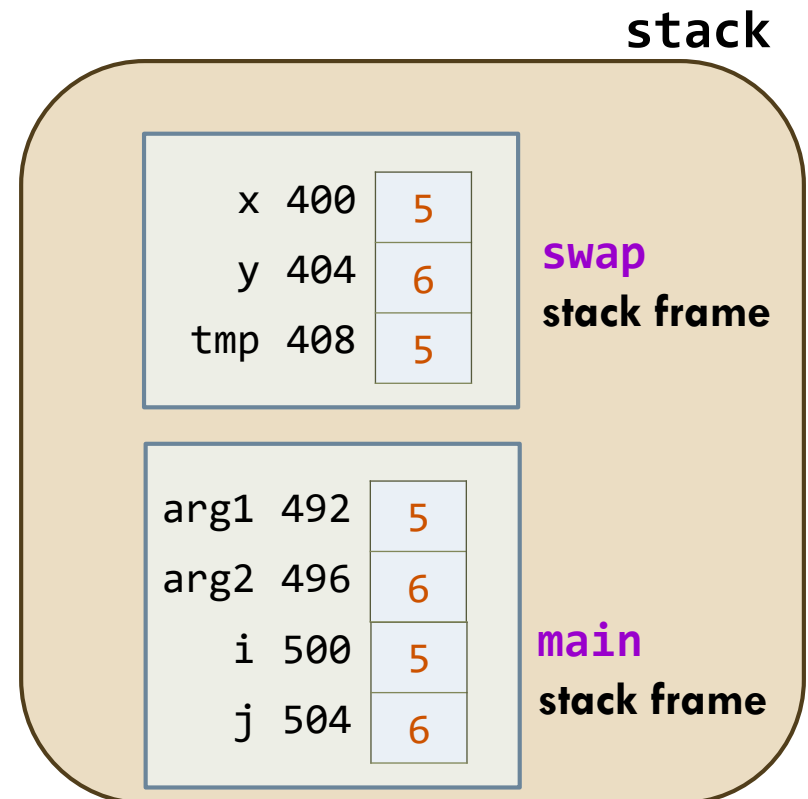
31

- Local variable `tmp` in function `swap` is defined and initialized

```
#include <stdio.h>

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

int main(void) {
    int i = 5, j = 6;
    swap(i, j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data by value (6/9)

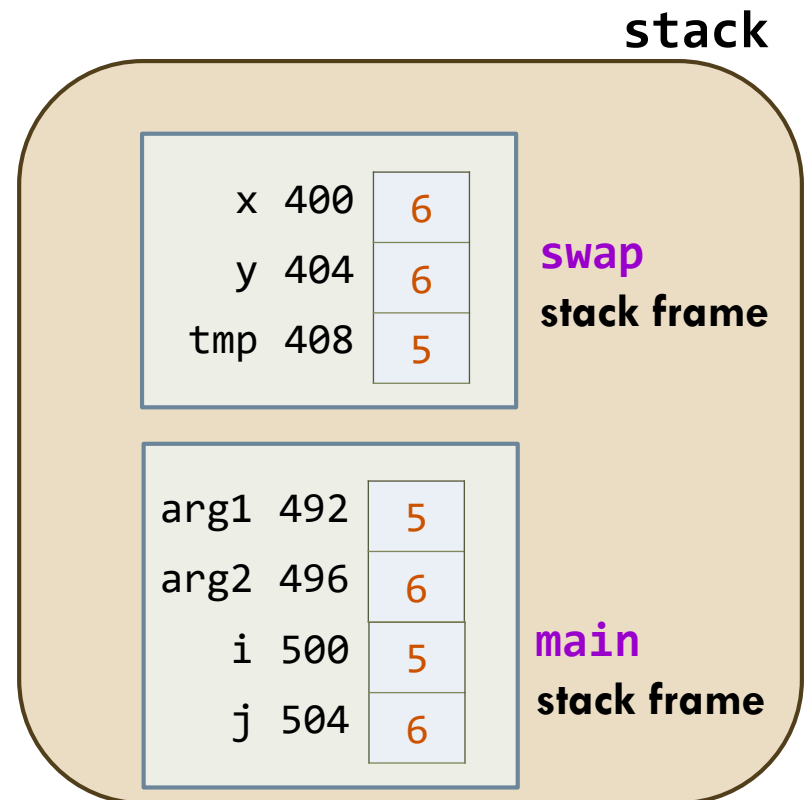
32

□ **X** is assigned y's value

```
#include <stdio.h>

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

int main(void) {
    int i = 5, j = 6;
    swap(i, j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data by value (7/9)

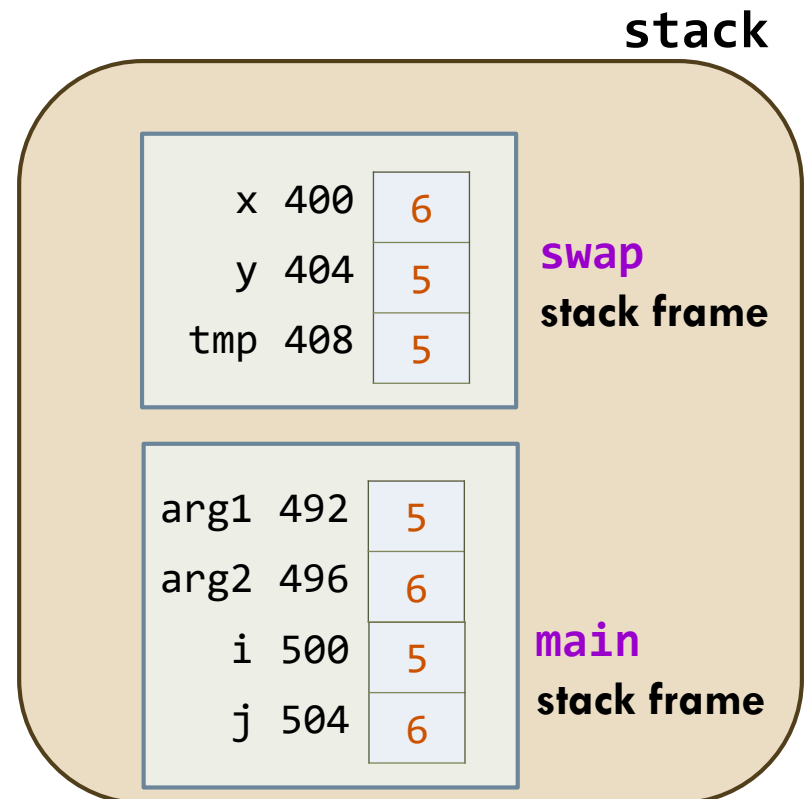
33

□ **y** is assigned tmp's value

```
#include <stdio.h>

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

int main(void) {
    int i = 5, j = 6;
    swap(i, j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data by value (8/9)

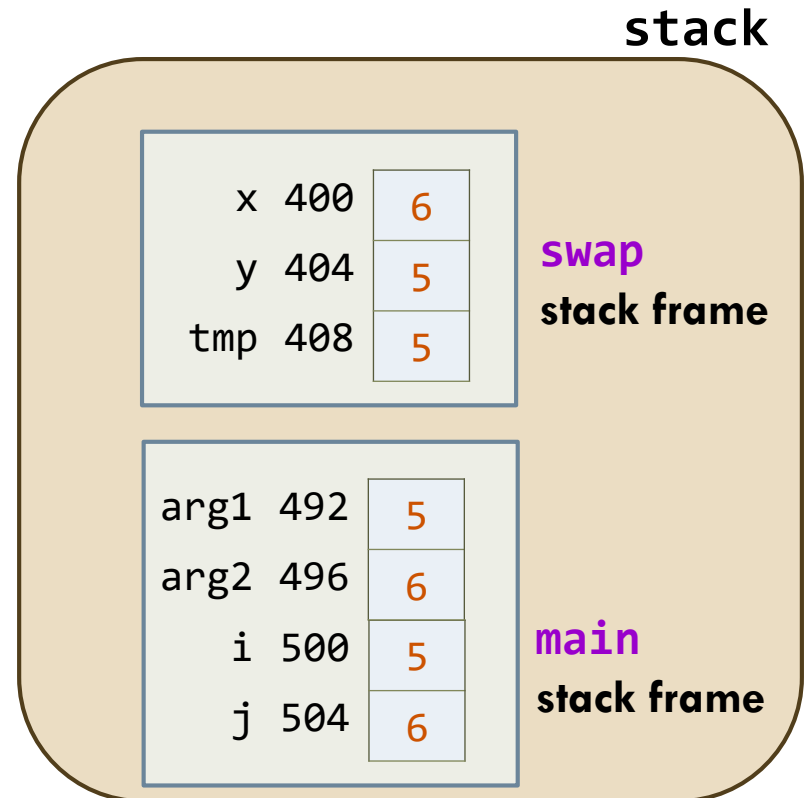
34

- Values of parameters **x** and **y** are swapped!!!
- But not the values of variables **i** and **j**!!!

```
#include <stdio.h>

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

int main(void) {
    int i = 5, j = 6;
    swap(i, j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data by value (9/9)

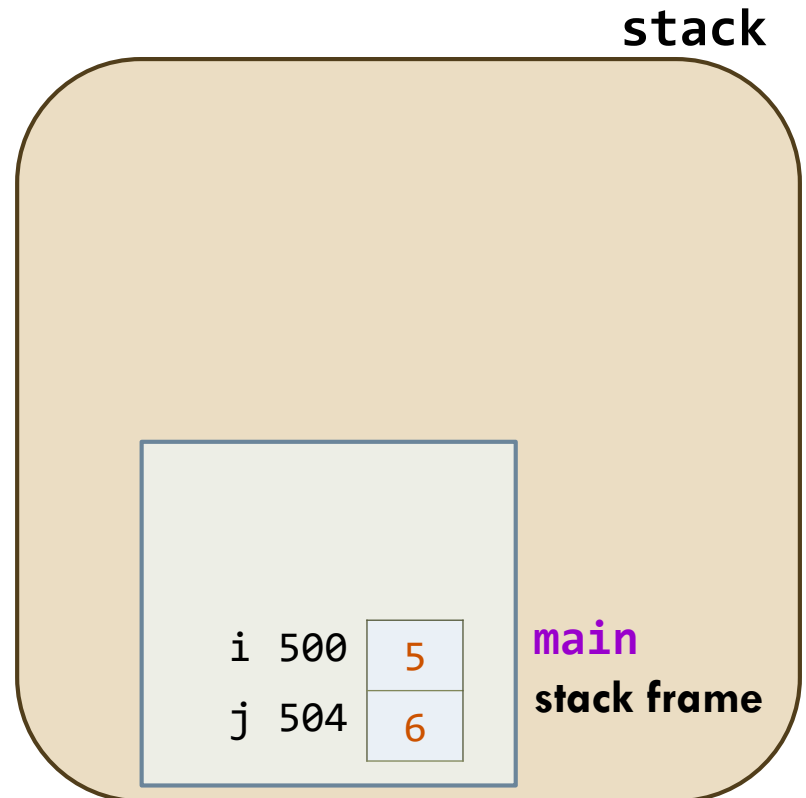
35

- Stack frame of function **swap** is popped off
- Visualizer

```
#include <stdio.h>

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

int main(void) {
    int i = 5, j = 6;
    swap(i, j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data using pointer (1 / 10)

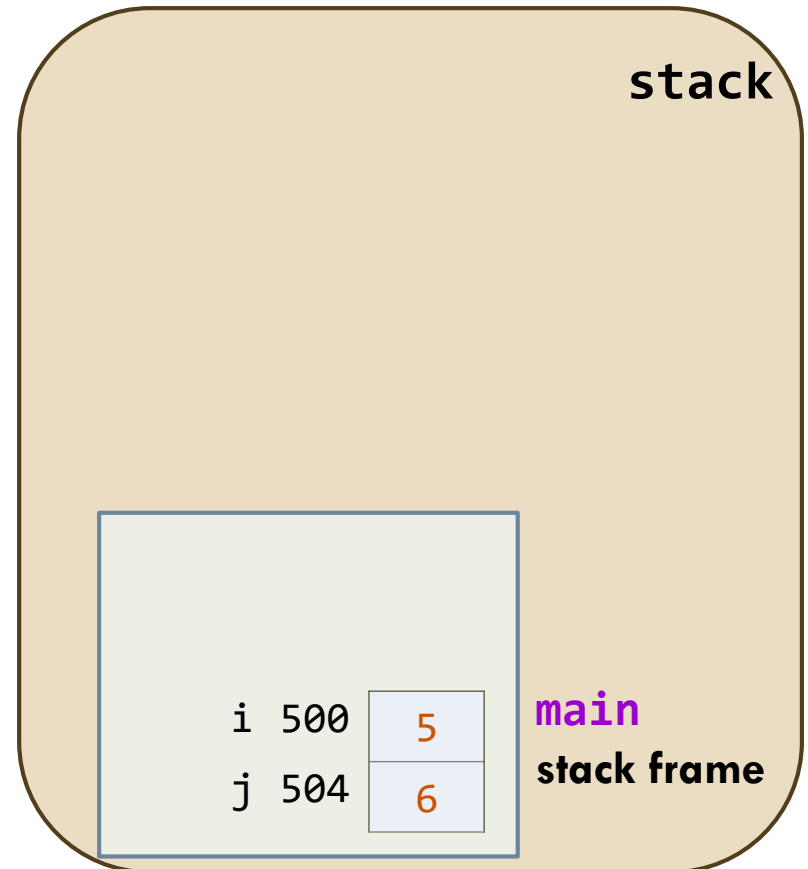
36

- Variables `i` and `j` are defined in `main`'s stack frame

```
#include <stdio.h>

void swap_ptrs(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    → int i = 5, j = 6;
    swap_ptrs(&i, &j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data using pointer (2/10)

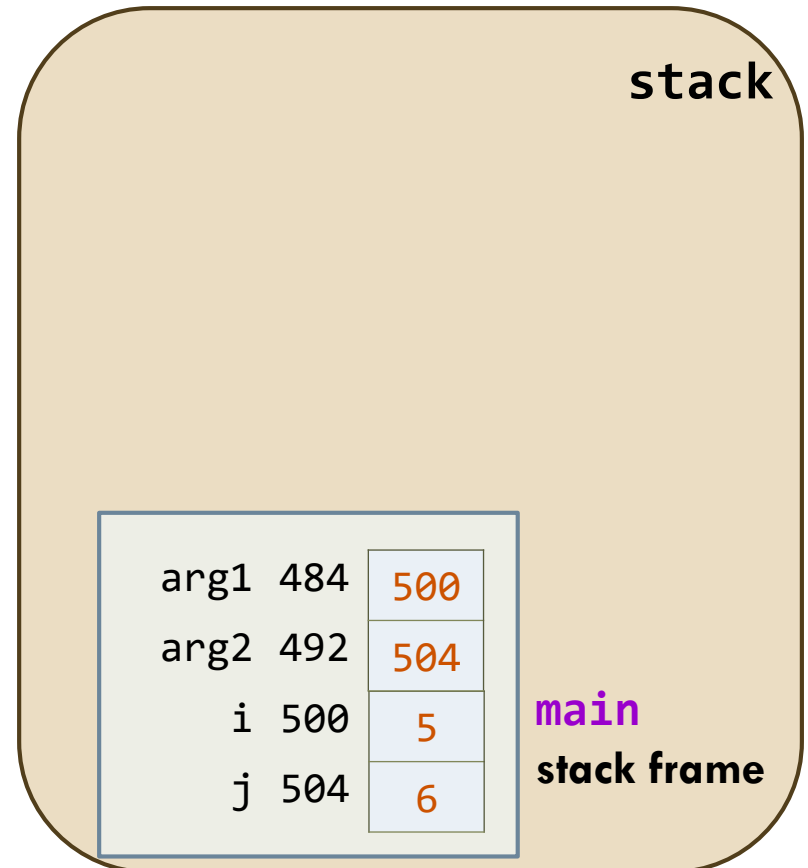
37

- Arguments of `swap_ptrs` function call operator are evaluated

```
#include <stdio.h>

void swap_ptrs(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int i = 5, j = 6;
    → swap_ptrs(&i, &j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data using pointer (3/10)

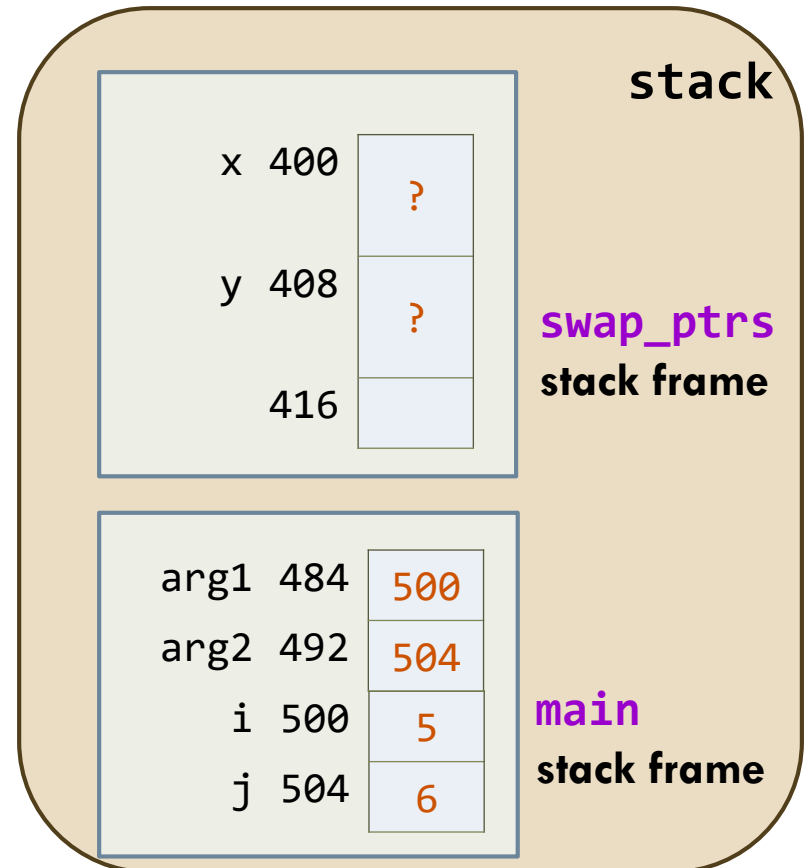
38

- Stack frame of function `swap_ptrs` is constructed

```
#include <stdio.h>

void swap_ptrs(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int i = 5, j = 6;
    → swap_ptrs(&i, &j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data using pointer (4/10)

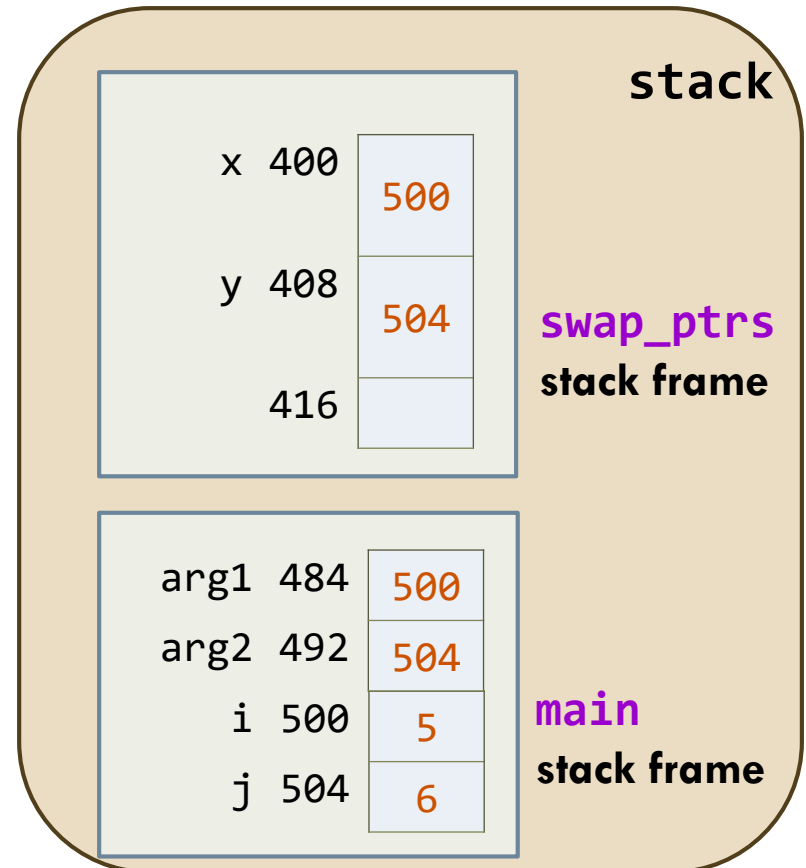
39

- Parameters of function `swap_ptrs` are initialized

```
#include <stdio.h>

→ void swap_ptrs(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int i = 5, j = 6;
    swap_ptrs(&i, &j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data using pointer (5/10)

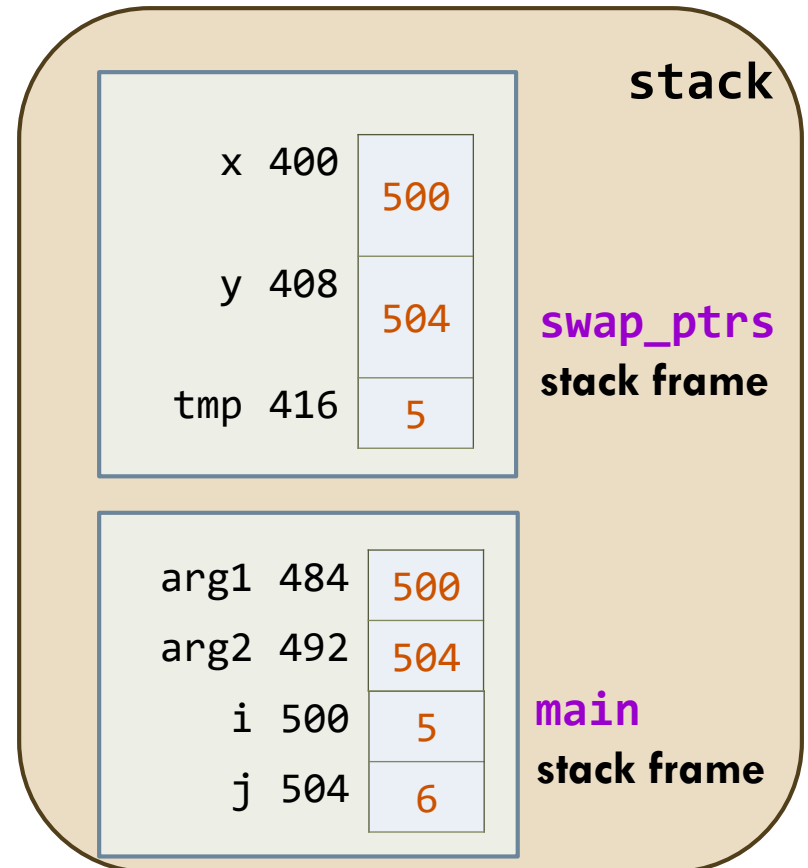
40

- Local variable `tmp` in function `swap_ptrs` is defined and initialized

```
#include <stdio.h>

void swap_ptrs(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int i = 5, j = 6;
    swap_ptrs(&i, &j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data using pointer (6/10)

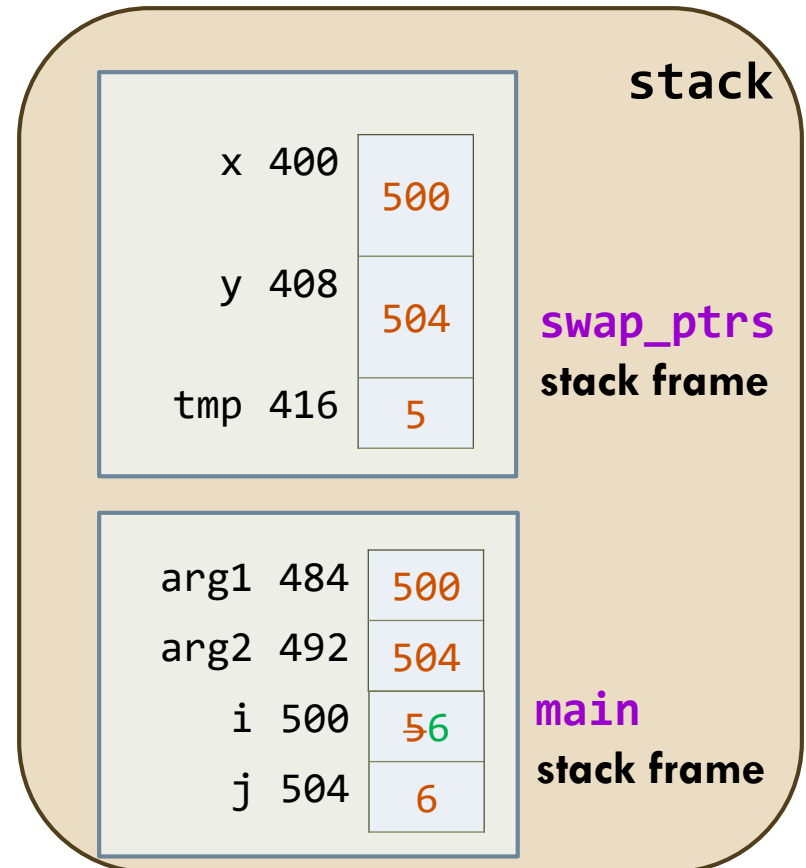
41

- Assign value of **int** pointed to by **y** to **int** pointed to by **x**

```
#include <stdio.h>

void swap_ptrs(int *x, int *y) {
    int tmp = *x;
    → *x = *y;
    *y = tmp;
}

int main(void) {
    int i = 5, j = 6;
    swap_ptrs(&i, &j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data using pointer (7/10)

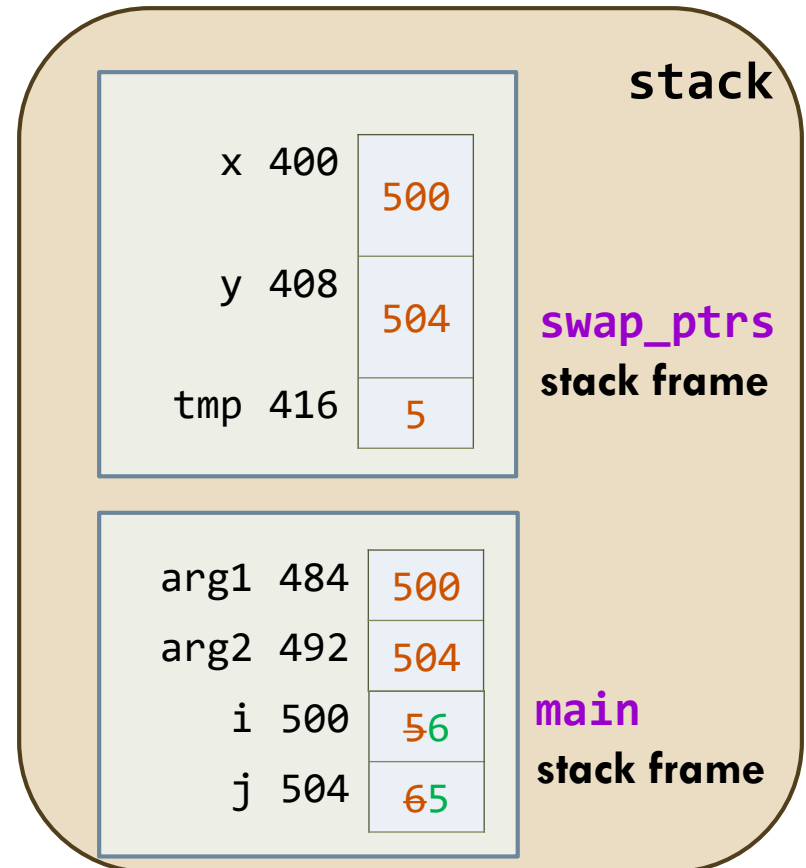
42

- Assign value of tmp to **int** pointed to by **y**

```
#include <stdio.h>

void swap_ptrs(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    → *y = tmp;
}

int main(void) {
    int i = 5, j = 6;
    swap_ptrs(&i, &j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data using pointer (8/10)

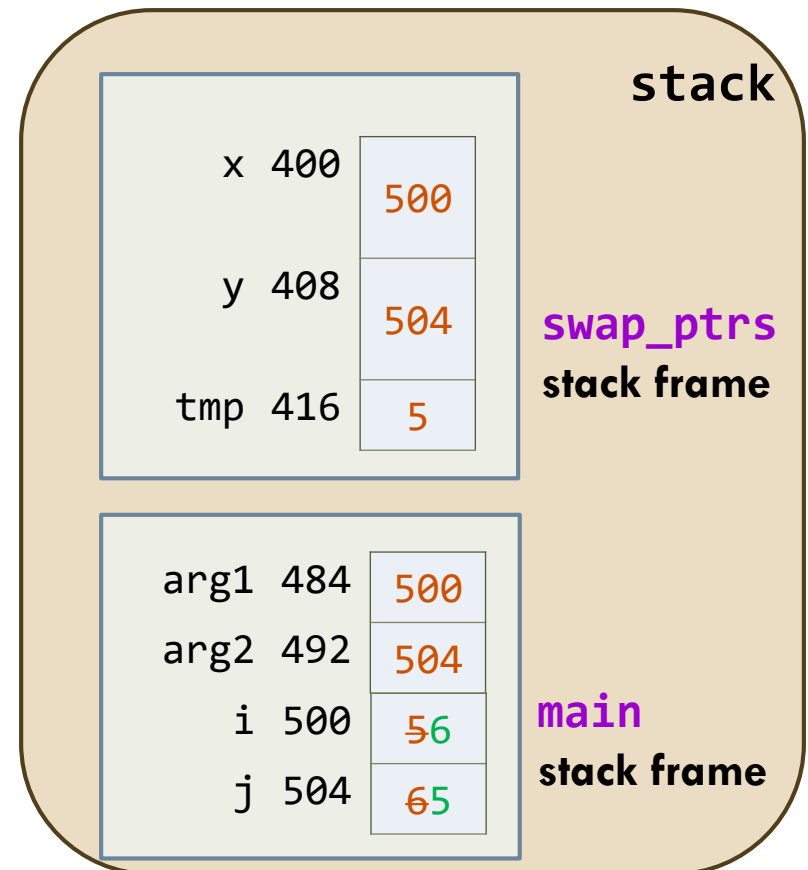
43

- Contents of memory locations pointed to by parameters **x** and **y** are swapped!!!

```
#include <stdio.h>

void swap_ptrs(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int i = 5, j = 6;
    swap_ptrs(&i, &j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data using pointer (9/10)

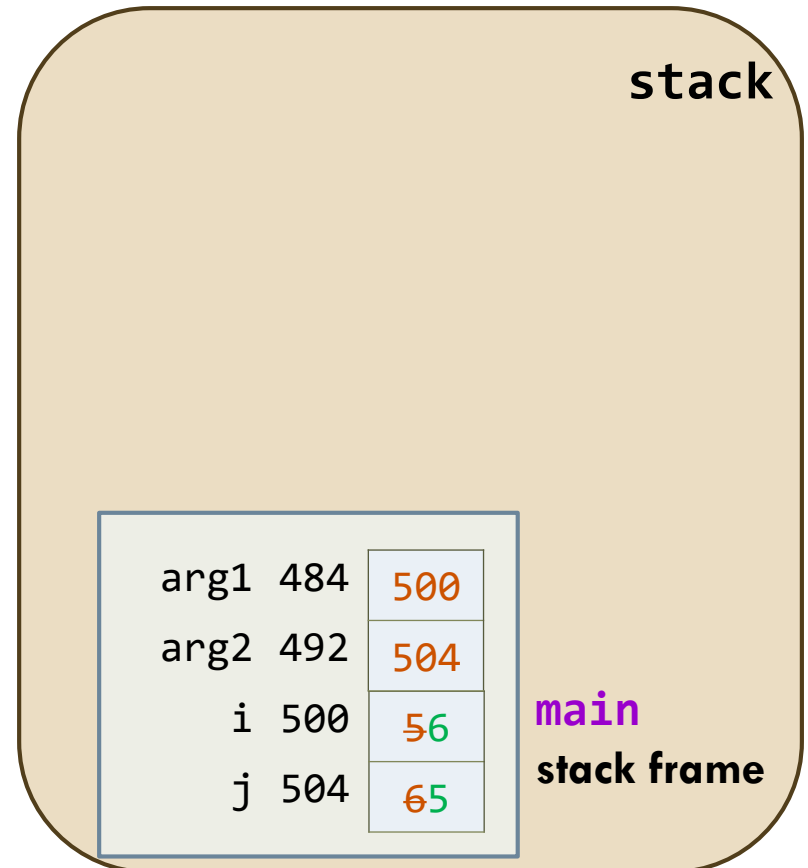
44

- Stack frame of function `swap_ptrs` is popped off
- Visualizer

```
#include <stdio.h>

void swap_ptrs(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int i = 5, j = 6;
    swap_ptrs(&i, &j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Passing data using pointer (10/10)

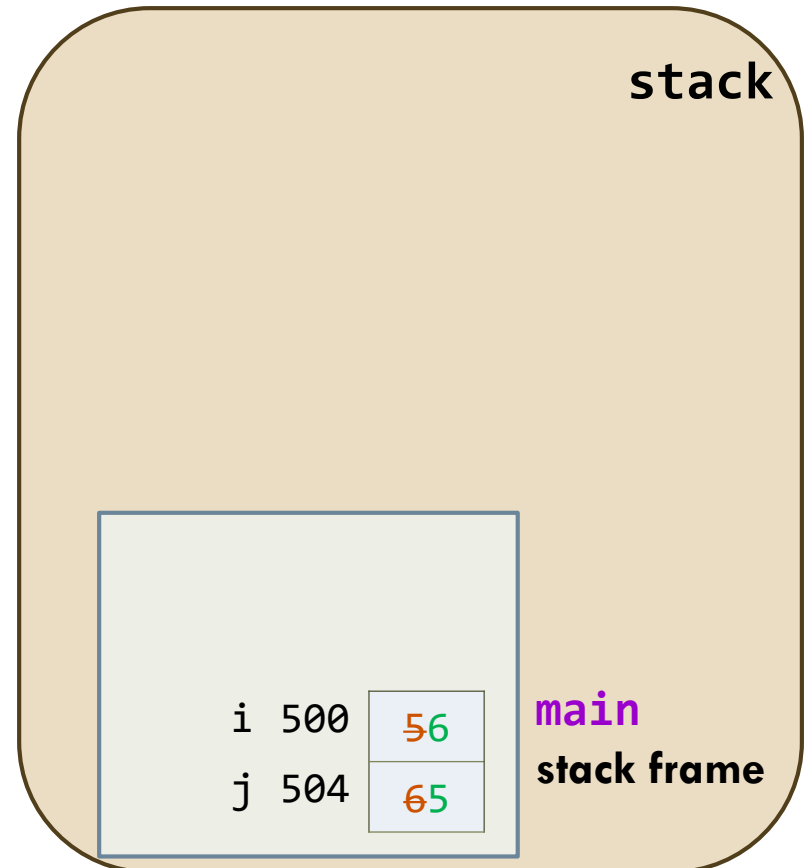
45

- Primary reason for passing data using pointer is to allow the function to modify the data

```
#include <stdio.h>

void swap_ptrs(int *x, int *y) {
    int tmp = *x;
    *x = *y;
    *y = tmp;
}

int main(void) {
    int i = 5, j = 6;
    swap_ptrs(&i, &j);
    printf("%d | %d\n", i, j);
    return 0;
}
```



Pointers and `scanf` (1/2)

46

- Arguments in calls of `scanf` are pointers:

```
int i;  
scanf("%d", &i);
```

- Without `&`, `scanf` receives value of `i`

```
int i;  
  
/* Luckily, flagged by compiler */  
scanf("%d", i);
```

Pointers and `scanf` (2/2)

47

- Although `scanf`'s arguments must be pointers, it's not always true that every argument needs `&` operator:

```
int i, *p;  
p = &i;  
scanf("%d", p);
```

- Using `&` operator in call to `scanf` would be wrong:

```
int i, *p;  
p = &i;  
scanf("%d", &p); /* ** WRONG ** */
```

Array and functions

48

- Pointers provide ability to efficiently pass array between functions with minimum overhead
- Efficient because what is passed is array's base address

Finding largest and smallest elements in array (1 / 2)

49

□ Prototype:

```
void min_max(int arr[], int N, int *pmin, int *pmax);
```

□ Example call:

```
int array[SIZE];  
// assignment values to array elements  
int minval, maxval;  
min_max(array, SIZE, &minval, &maxval);
```

Finding largest and smallest elements in array (2/2)

50

```
void min_max(int arr[], int N, int *pmin, int *pmax) {  
    *pmin = *pmax = arr[0];  
    for (int i = 1; i < N; ++i) {  
        *pmin = (arr[i] < *pmin) ? arr[i] : *pmin;  
        *pmax = (arr[i] > *pmax) ? arr[i] : *pmax;  
    }  
}
```

Using `const` to protect function arguments (1 / 3)

51

- `min_max` only wants to read array elements, not change their values

```
void min_max(int arr[], int N, int *pmin, int *pmax) {  
    *pmin = *pmax = arr[0];  
    for (int i = 1; i < N; ++i) {  
        *pmin = (arr[i] < *pmin) ? arr[i] : *pmin;  
        *pmax = (arr[i] > *pmax) ? arr[i] : *pmax;  
    }  
}
```

Using `const` to protect function arguments (2/3)

52

- Author of `min_max` can document that function will not modify object whose address is passed to function
- Add `const` to array parameter's declaration
 - ▣ Contract established between author and client that array elements will not be changed
 - ▣ Compiler will enforce this contract

```
void min_max(int const arr[], int N, int *pmin, int *pmax);
```

Using `const` to protect function arguments (3/3)

53

- Following declarations are equivalent since type specifiers and type qualifiers can be written in any order:

```
void max_min(int const a[], int N, int *max, int *min);  
void max_min(const int a[], int N, int *max, int *min);  
void max_min(int const *a, int N, int *max, int *min);  
void max_min(const int *a, int N, int *max, int *min);
```

Pointers as return values (1 / 4)

54

- Functions are allowed to return pointers:

```
int* max_ptr_cmp(int *pa, int *pb) {  
    return *pa > *pb ? pa : pb;  
}
```

Pointers as return values (2/4)

55

- Use cases of `max_ptr_cmp` function

```
int x = 10, y = 20, *pi;  
pi = max_ptr_cmp(&x, &y);  
/* pi points to object with largest rvalue */
```

```
int x = 10, y = 20, z = 30; *pi;  
*max_ptr_cmp(&x, &y) = z;  
/* object with largest rvalue assigned z's value */
```

Pointers as return values (3/4)

56

- Never return pointer to *automatic* local variable:

```
int *foo(void) {  
    int x = 10;  
    return &x;  
}
```

- Variable x won't exist after *foo* returns

```
int *pi;  
pi = foo();  
/* runtime error!!! */  
*pi = 20;
```


Pointers as return values (4/4)

57

- Pointers can point to array elements
- If `a` is array, then `&a[i]` is a pointer to element `i` of `a`
- Sometimes useful for function to return pointer to an array element
- Function that returns pointer to middle element of `a`, assuming `a` has at least `N` elements:

```
int *find_middle(int a[], int N) {  
    return &a[N/2];  
}
```