CSCI E-50 WEEK 4

TERESA LEE
[teresa@cs50.net]
February 18 2018

Today

- Problem Solving Strategies
- Input and Output Redirection
- File I/O
- Memory
- Pointers
- Dynamic Memory Allocation

Problem Solving Strategy

How to Solve a CS50 Pset [pdf]

Redirection Commands

`>` - output; print the output of a program to a file instead of `stdout`

- `>>` - append to an output file
- `2>` - only print out error messages to a file

`<` - input; use the contents of some file as input to a program

`|` - pipe; take the output of one program and use it as input in another

Let’s look at an example

Example 1

1. `$ ./hello > 1.txt`
2. `$ ./hello >> 1.txt`
3. `$ ./hello 2> 3.txt`
4. `$ ./hello < 4.txt`

File I/O

- It is just as easy to write out to a file or read in from a file, this time using system commands and the C standard library, instead of input/output redirection!
- Output files do not disappear when your program finishes running!

Let’s Look at an Example

Example 2

Common I/O Functions

- fopen() -- creates a file reference (“r”, “w”, “a”)
- fread() -- reads some amount of data from a file
- fwrite() -- writes some amount of data to a file
- fgets() -- reads a single string from a file (typically, a line)
- fputs() -- writes a single string to a file (typically, a line)
- fgetc() -- reads a single character from a file
- fputc() -- writes a single character from a file
- fseek() -- like rewind and fast forward on YouTube, to navigate around a file
- ftell() -- like the timer on YouTube, tells you where you are in a file (how many bytes in)
- fclose() -- closes a file reference, used once done working with the file

File I/O

Read from the file

- fgetc - returns the next char
- fgets - returns a line of text
- fread - reads a certain # of bytes and places them into an array
- fseek - moves to a certain point

Write to the file

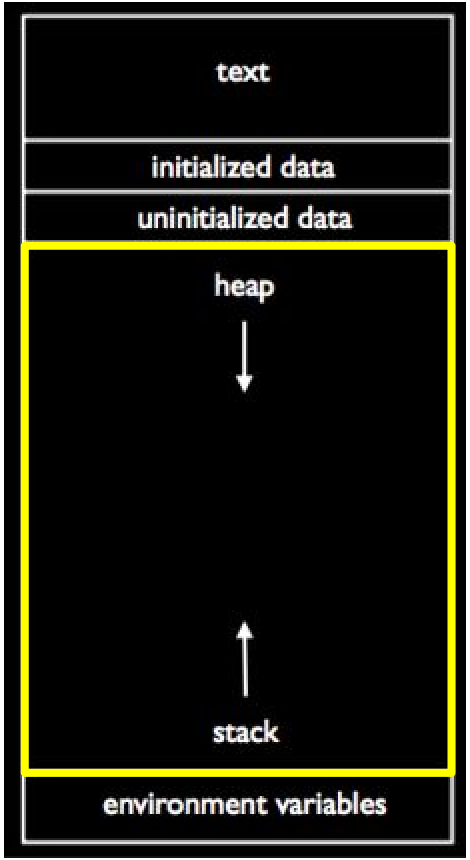
- fputc - write a char
- fputs - returns a line of text
- fprintf - print a formatted output to a file
- fwrite - write an array of bytes to a file

Memory

Stack is a contiguous block of memory set aside when a program starts running:

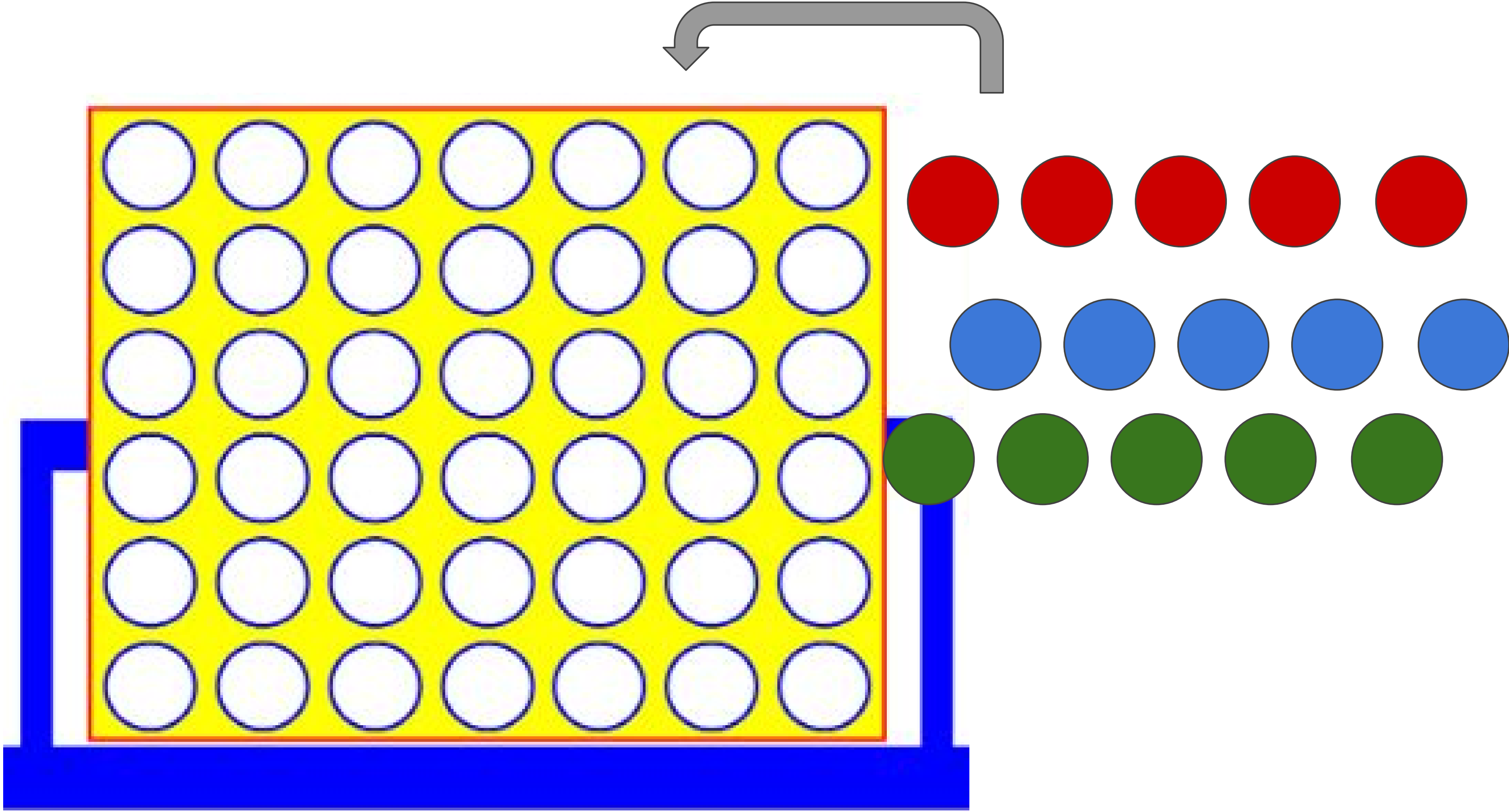
- Metadata
- Any variables held in `read-only` memory
- All local variables - each function has its own stack frame and its variables are protected from other functions. The size of a function's stack frame is dependent largely on its local variables

Heap is essentially a region of unused memory that can be dynamically allocated



Stack frames

- When you call a function system sets aside space in memory for that function to do its job
- More than one frames can be open but only one can ever be active
- When a new function is called, a new frame is pushed onto the top of the stack and becomes active frame
- When a function finishes its work, its frame is popped off of the stack and the frame immediately below it becomes the new active function on the top of the stack.



Memory

- A huge array of 8-bit wide bytes
- Memory is limited!
- Each data type takes up a certain amount

Data Type	**Size (in bytes)**
int	4
char	1
float	4
double	8
long long	8
string	?

Remember this?

1	2	3	4	5	6
[0]	[1]	[2]	[3]	[4]	[5]

Similarly, each location in memory has address!

	A		15					B	O	O	\0
0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0x10	0x11

```
char c = ‘A’;

int date = 15;

string sound = “BOO”;

* We use hexadecimal notation for memory addresses.
```

Pointers

- Pointers are a tool that give us a way to literally pass information between functions
- A pointer is a data item whose value is a memory address and whose type describes the type of data you will find if you visit that memory address.
- If we know exactly where in memory a variable lives, then we can find it from any function, thus allowing us to pass that data around.
- **<type> *<name of the pointer>**

A POINTER IS JUST AN ADDRESS
A POINTER IS JUST AN ADDRESS
A POINTER IS JUST AN ADDRESS
A POINTER IS JUST AN ADDRESS

Pointers

- If we have some variable we know of, particularly one that lives on the stack and has a name, we can find its address by prepending a &. E.g., &num
- To access the data at an address, we need to dereference it, using the * operator.

Pointers: Let’s Look at an Example

Example 3

	c		date					sound			
	A		15					B	O	O	/0
0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB

Int date = 15;
<type> *<variable>
Int *point_to_date;
Point_to_date = &date
*point_to_date

point_to_date

Pointers

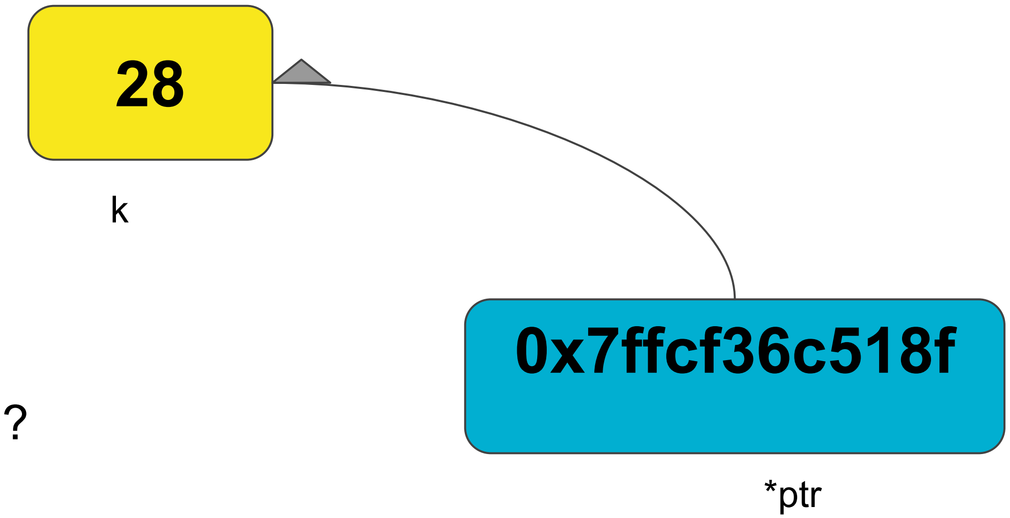
int k;

k = 25;

int *ptr;

ptr = &k;

*ptr = 28; what would happen?



Pointers

Gives you another way of passing data between functions.

Allows us to modify or inspect the location to which it points (* dereferencing the pointer)

Pointers

- The simplest pointer available to us in C is the NULL pointer.
  - What would this point to? **NOTHING!**
- When you create a pointer and you don't set its value immediately, you should **always** set the value of the pointer to NULL.
- You can check whether a pointer is NULL using the equality operator (==).Ex) if (point_to_file == NULL)
  - What if you try to dereference NULL pointer? Sementation fault

Pointers

& is the reference, or address-of, operator. It returns the address in memory at which a variable is stored.

***** is the dereference operator. A pointer's value is a memory address. When the dereference operator is applied to a pointer, it returns the data stored at that memory address.

Pointers: String

In the CS50 library, a string is just another way of saying a char *. A pointer to a character.

- The only way to refer to a string is by a pointer to its first character (aka where its first character lives in memory), so we need some way to mark the end. How???
- We also know that a pointer is just an address. So if a string is a char * (aka a char pointer), how many bytes does it take up?

Data Type	Size (in bytes)
int	4
char	1
float	4
double	8
long long	8
char * (aka string)	4 or 8 (see ex4_size.c

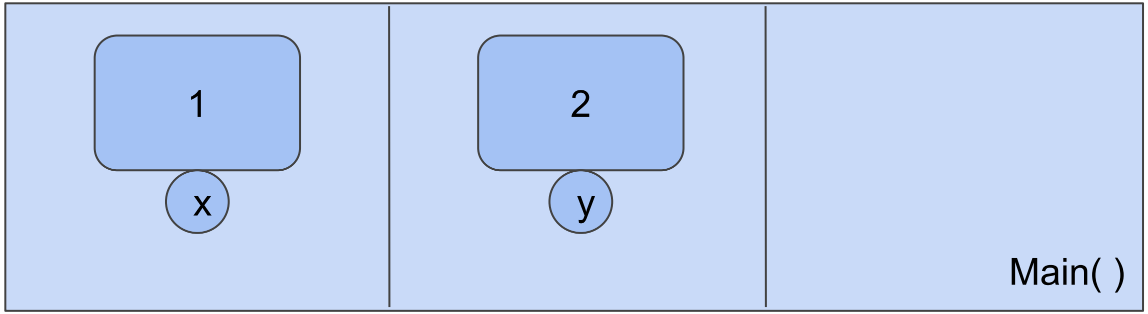
Let’s Look at an Example

ex5_swap.c

```
#include <stdio.h>

void swap(int a, int b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x was %i\n", x);
    printf("y was %i\n", y);

    swap(x, y);

    printf("x is now %i\n", x);
    printf("y is now %i\n", y);
}

void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

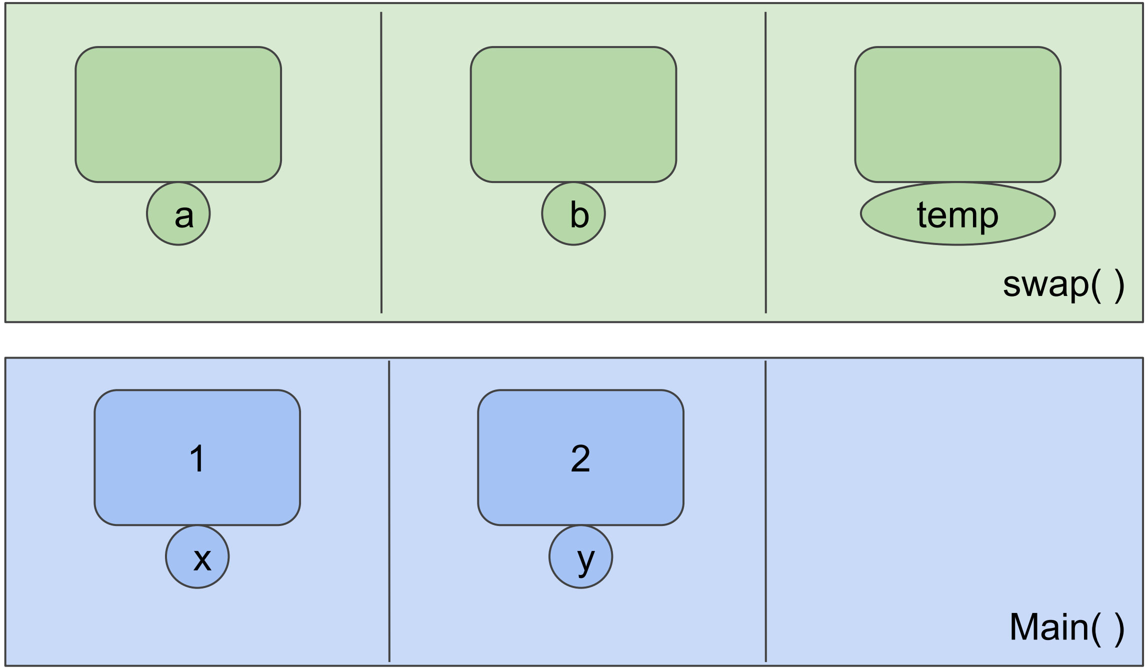
Let's Look at an Example

```
ex5_swap.c

#include <stdio.h>

void swap(int a, int b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x was %i\n", x);
    printf("y was %i\n", y);

    swap(x, y);

    printf("x is now %i\n", x);
    printf("y is now %i\n", y);
}

void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```



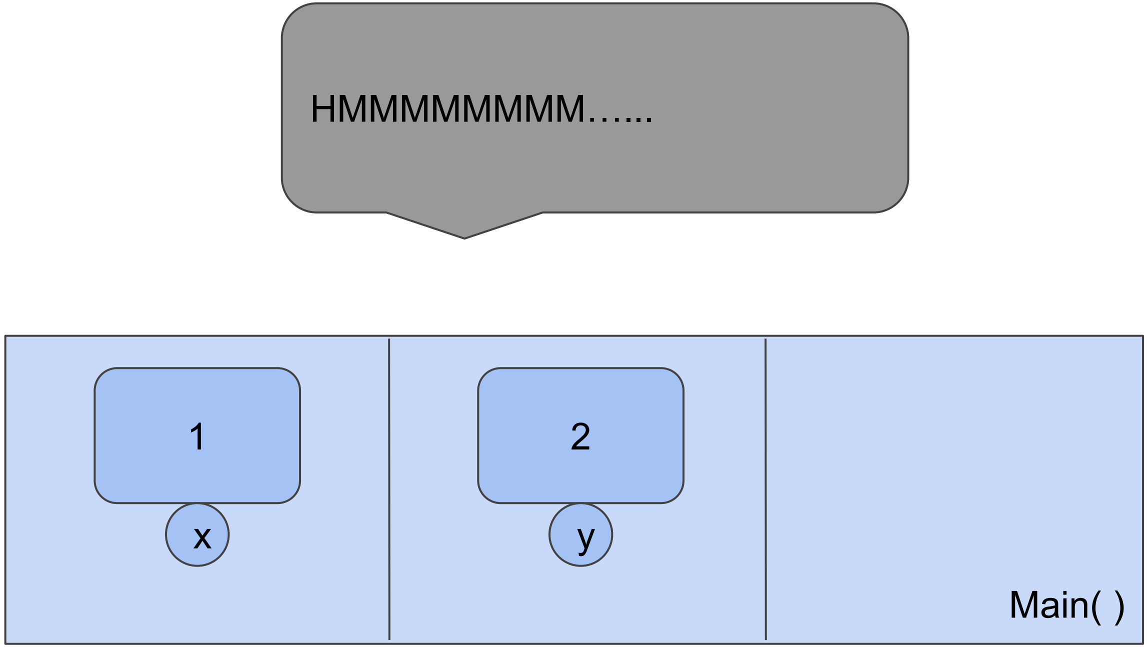
Let's Look at an Example

```c
#include <stdio.h>

void swap(int a, int b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x was %i\n", x);
    printf("y was %i\n", y);

    swap(x, y);

    printf("x is now %i\n", x);
    printf("y is now %i\n", y);
}

void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```



Let’s Look at an Example

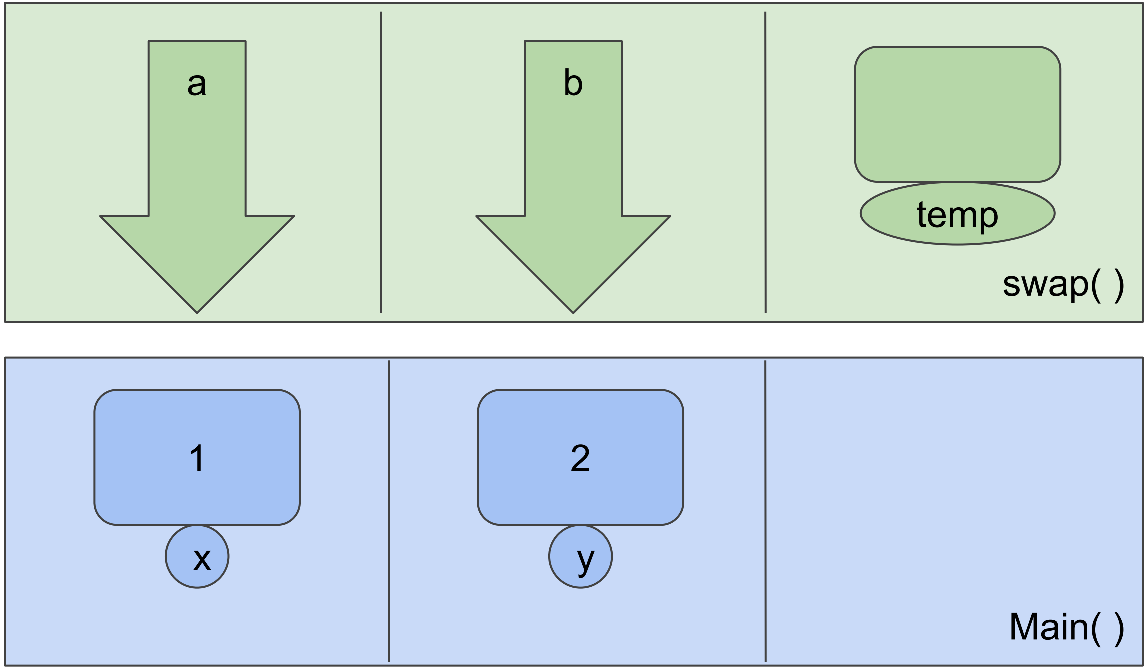
```
#include <stdio.h>

void swap(int a, int b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x was %i\n", x);
    printf("y was %i\n", y);

    swap(x, y);

    printf("x is now %i\n", x);
    printf("y is now %i\n", y);
}

void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```



Let’s Look at an Example

Example 6

```
#include <stdio.h>

void swap(int *a, int *b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x was %i\n", x);
    printf("y was %i\n", y);

    swap(&x, &y);

    printf("x is %i\n", x);
    printf("y is %i\n", y);
}

void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```



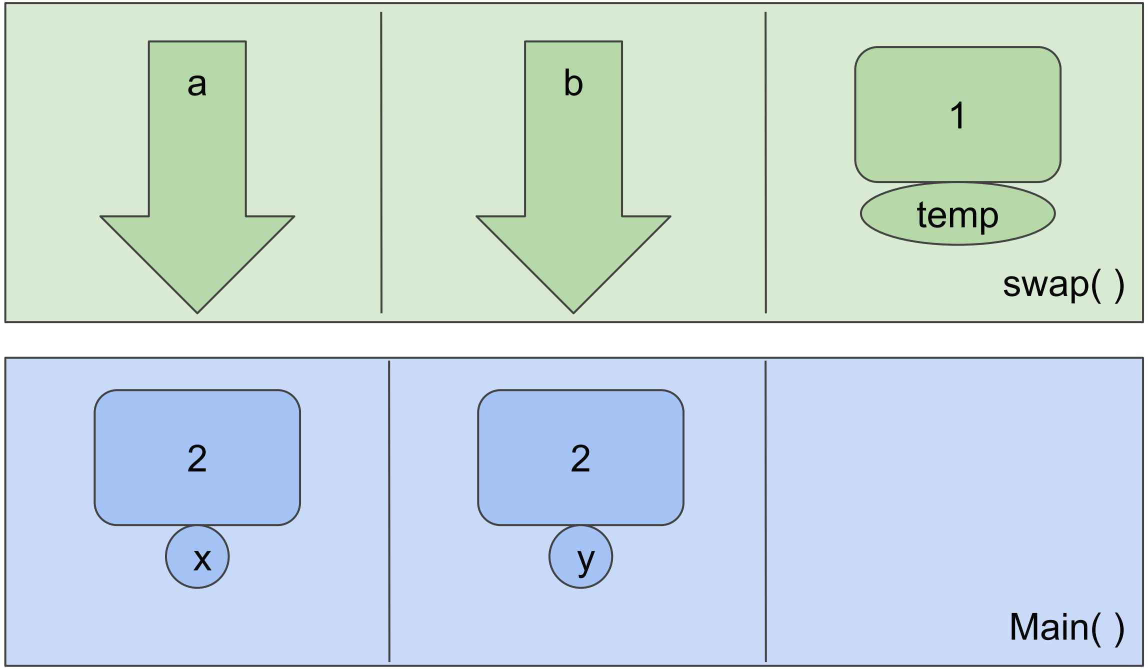
What is the value of a and b?

Let's Look at an Example

Example 6

```
#include <stdio.h>

void swap(int *a, int *b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x was %i\n", x);
    printf("y was %i\n", y);

    swap(&x, &y);

    printf("x is %i\n", x);
    printf("y is %i\n", y);
}

void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

What is the value of a and b?

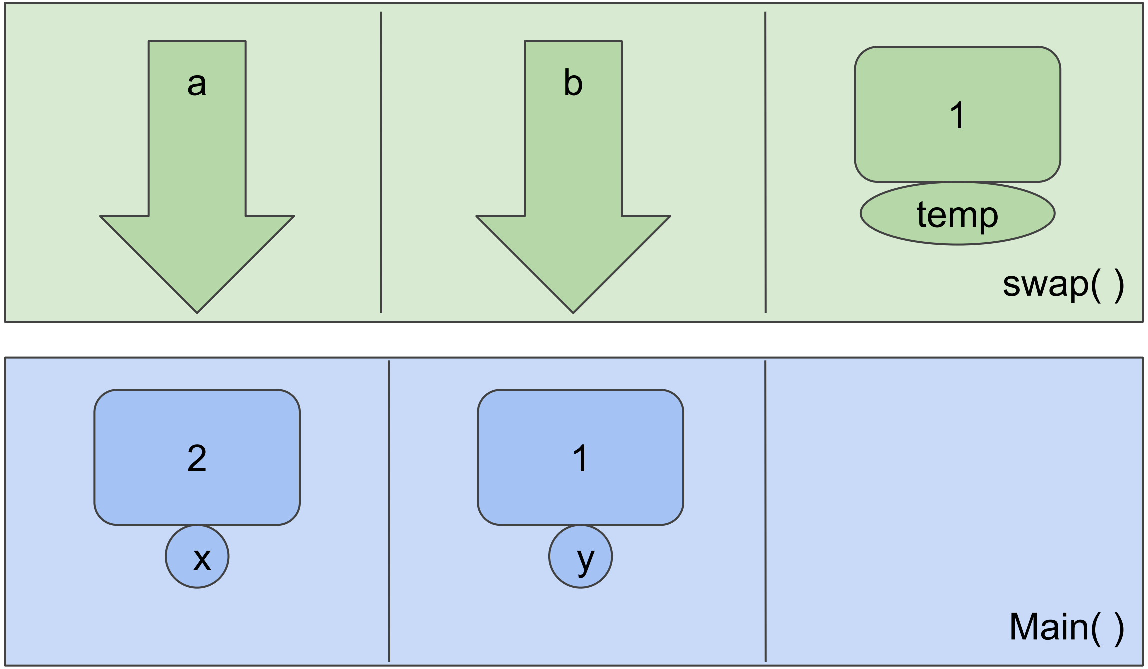


Let’s Look at an Example

Example 6

```
#include <stdio.h>

void swap(int *a, int *b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x was %i\n", x);
    printf("y was %i\n", y);

    swap(&x, &y);

    printf("x is %i\n", x);
    printf("y is %i\n", y);
}

void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

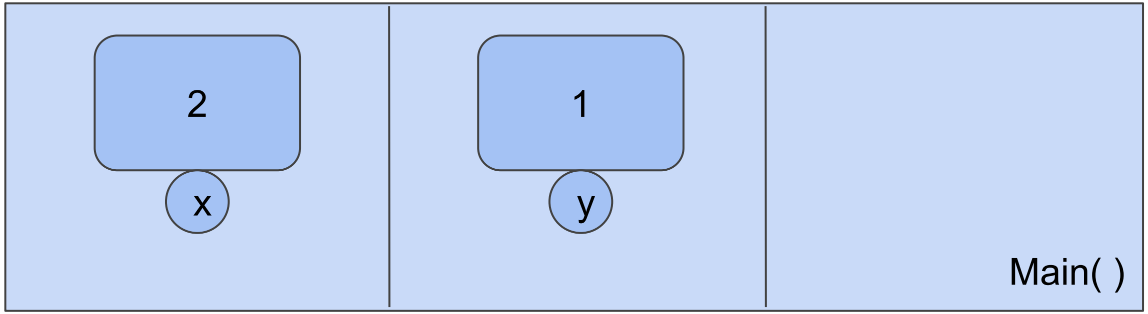
What is the value of a and b?



Let’s Look at an Example

Example 6

```
#include <stdio.h>

void swap(int *a, int *b);

int main(void)
{
    int x = 1;
    int y = 2;

    printf("x was %i\n", x);
    printf("y was %i\n", y);

    swap(&x, &y);

    printf("x is %i\n", x);
    printf("y is %i\n", y);
}

void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```



Dynamic Memory Allocation

- So far, we looked at one way to use pointers -- connecting a pointer variable by pointing it at another variable that already exists in our program.
- But what if we don’t know in advance how much memory we’ll need at compile time? How do we access more memory at runtime?

Dynamic Memory Allocation

- So far, we looked at one way to use pointers -- connecting a pointer variable by pointing it at another variable that already exists in our program. Where?
- But what if we don’t know in advance how much memory we’ll need at compile time? How do we access more memory at runtime?
- Pointers can also be used to do this. Memory allocated *dynamically* (at runtime) comes from a pool of memory called the heap. Memory allocated at compile time typically comes from a pool of memory called the stack.

Dynamic Memory Allocation

- We get this dynamically-allocated memory via a call to the function malloc(), passing as its parameter the number of *bytes* we want. malloc() will return to you a **pointer** to that newly-allocated memory.
- If malloc() can't give you memory (because, say, the system ran out), you get a NULL pointer. ALWAYS CHECK FOR NULL!

```
// Statically obtain an integer
int x;

// Dynamically obtain an integer
int *px = malloc(sizeof(int));
```

Dynamic Memory Allocation

Instead of

```
int *px = malloc(4);
```

Use

```
int *px = malloc(sizeof(int));
```

Dynamic Memory Allocation

```
// Get an integer from the user
int x = get_int();

// Array of floats on the stack
float stack_array[x];

// Array of floats on the heap
float *heap_array = malloc(x * sizeof(float));
```

Dynamic Memory Allocation

- Dynamically allocated memory is not automatically returned to the system for later use when no longer needed.
- Failing to return memory back to the system when you no longer need it results in a **memory leak**, which compromises your system’s performance.
- All memory that is dynamically allocated must be released back by free()-ing its pointer.

Dynamic Memory Allocation

- Every block of memory that you malloc(), you must later free().
- **Only** memory that you obtain with malloc() should you later free().
- Do not free() a block of memory more than once.

Dynamic Memory Allocation

```
// create array of words on heap
char *word = malloc(50 * sizeof(char));

// do stuff with word

// now we’re done
free(word);
```

Let’s Look at an Example

`Ex6_memory.c`

Watch CS50 Shorts

Call Stacks;
File Pointers;
Pointers;
Dynamic Memory Allocation;
Hexadecimal

pset4

1. Figure out whodunit.
2. Resize some images.
3. Recover some photos.

`Read bmp.h`