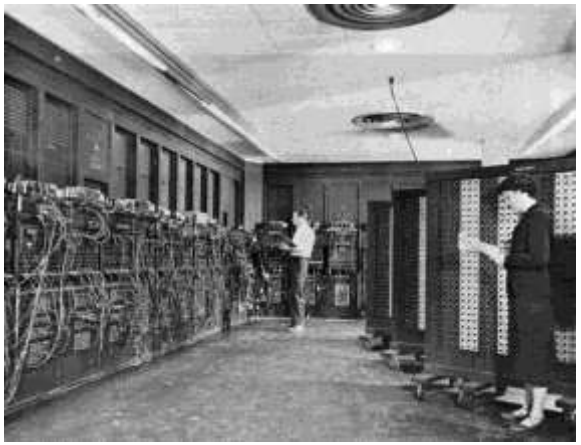
HIGH-LEVEL PROGRAMMING I

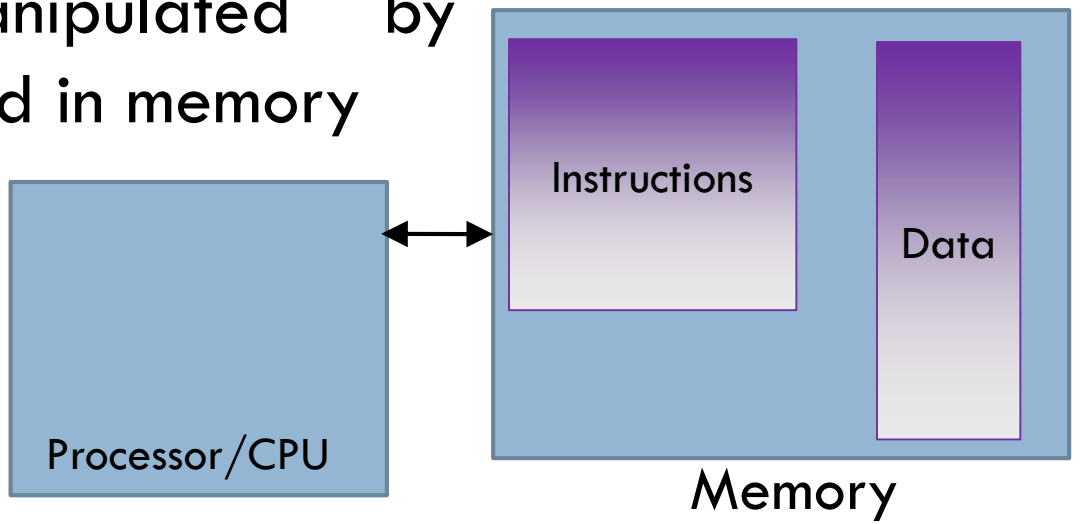
Program memory map and Heap by Prasanna Ghali

Stored-Program Computer

- Computers based on von Neumann architecture use *stored-program* concept:
  - Program that manipulates data is stored in memory
  - Data to be manipulated by program is also stored in memory



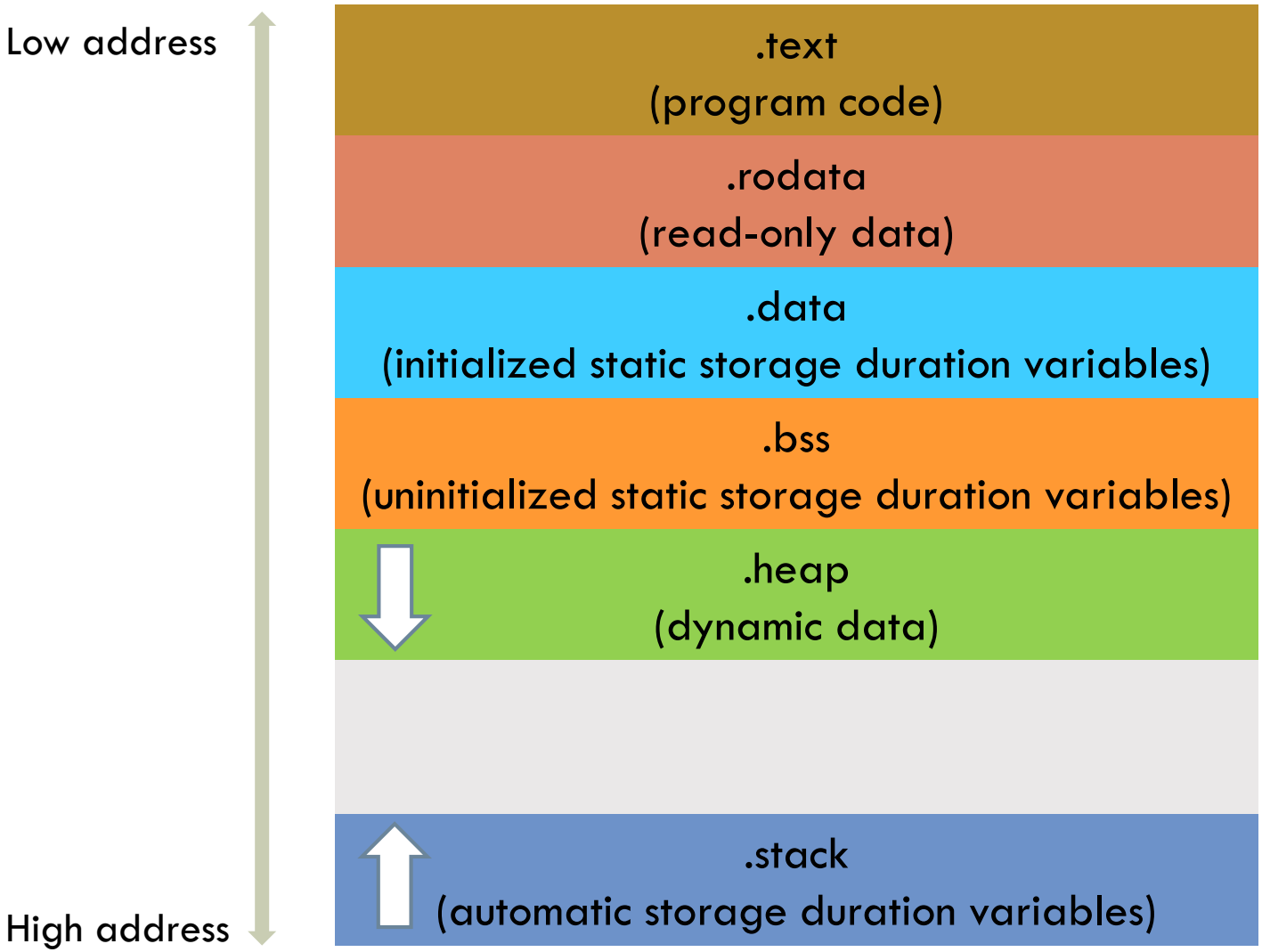
Reference



Program memory map (1/2)

- C program is single binary file containing all the information necessary to load program into memory and run it
  - Machine code of program
  - Read-only data such as format strings in `printf` statements
  - Initialized and uninitialized variables with static storage duration
- When program is executed, OS program called *loader* copies code and data from executable file in disk to memory

Program memory map (2/2)



.text segment

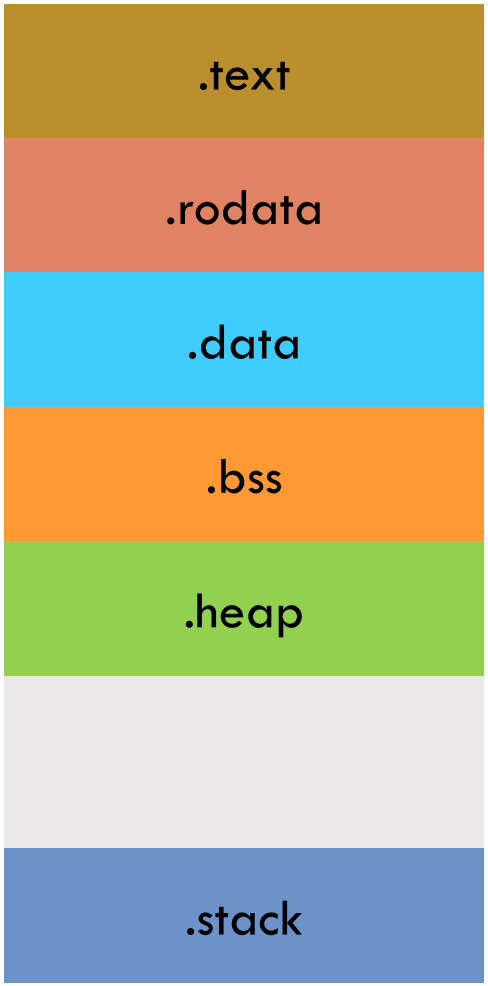
- Machine code of compiled program plus external library code added by linker

.rodata segment


```
char const ival = 40;
char const *pc  = "Singapore";
printf( "Hello World" );
```

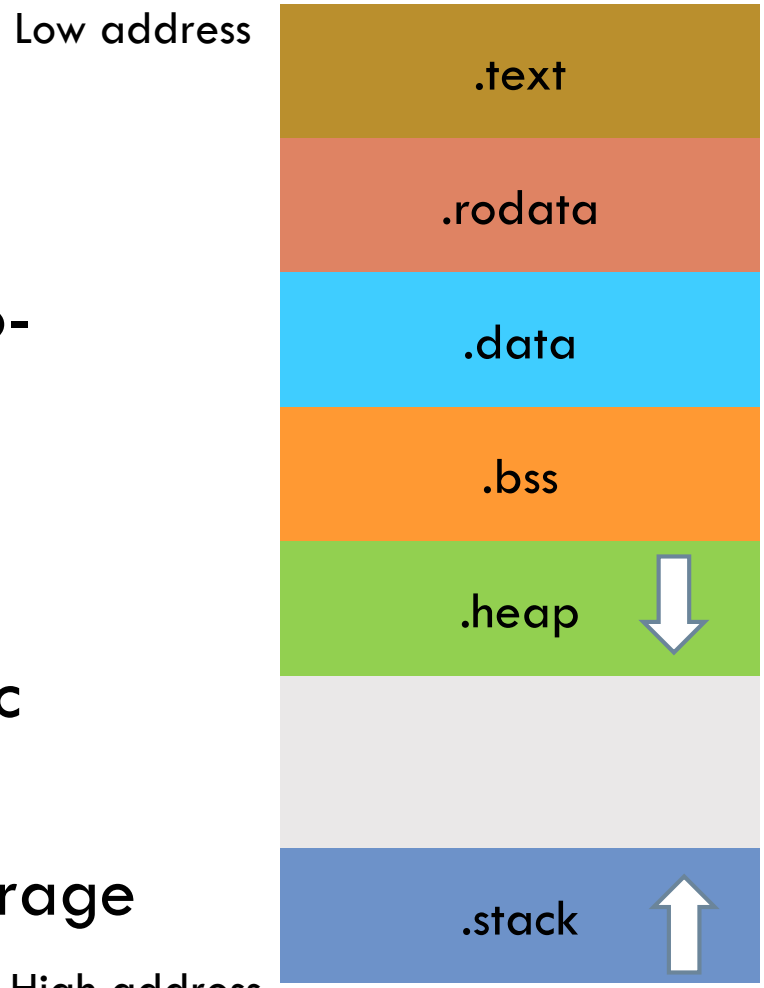
External `const` variables `ival` and `pc` and string literals `"Singapore"`, and `"Hello World"` are stored in .rodata

Internal `const` variables stored in stack or values represented in machine language instructions



Data segment (1/2)

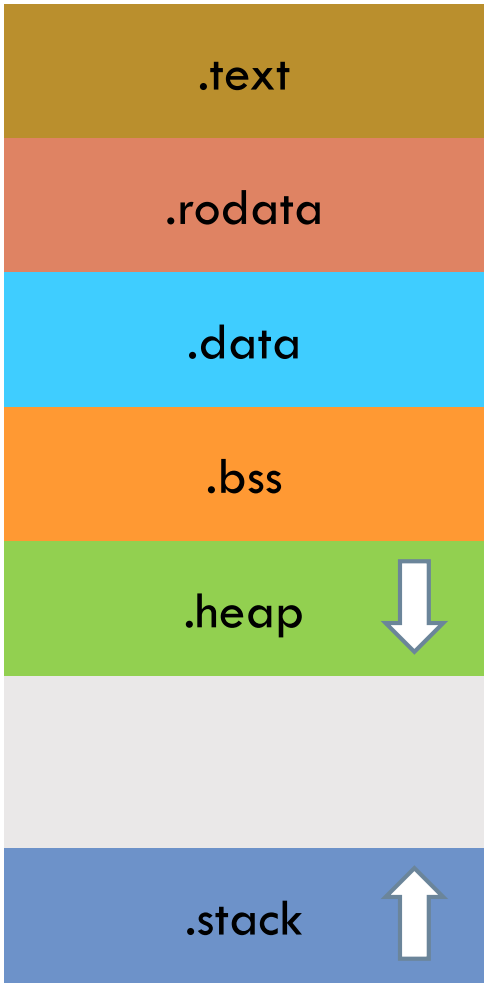
- .data: Non-zero initialized variables with static storage duration
- .bss (Block Storage Start): Zero-initialized and uninitialized variables with static storage duration
- .stack: Variables with automatic storage duration
- .heap: Values with dynamic storage duration



Data segment (2/2)


```c
int varA_bss;
double varB_bss = 0;
int varC_data = 1;
int const varD_rodata = 2;
// string literal also stored in rodata
char const *ptrE_rodata = "String";

void foo(int paramF_stack) {
  int varG_stack;
  float varH_stack;
  static int varI_bss;
  static double varJ_bss = 0.0;
  static int varK_data = 10;
  char * ptrL_TO_HEAP_MEM_stack = malloc(81);
  // more code here
  free(ptrL_TO_HEAP_MEM_stack);
}
```



.stack segment

- stack is memory region where local variables and function parameters live
- Portion of stack allocated for that function called *stack frame*
- When function is called, stack frame allocated for that function
- When function returns, function's stack frame goes away

.stack segment

- stack is memory region where local variables and function parameters live
- Portion of stack allocated for that function called *stack frame*
- When function is called, stack frame allocated for that function
- When function returns, function's stack frame goes away

The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```

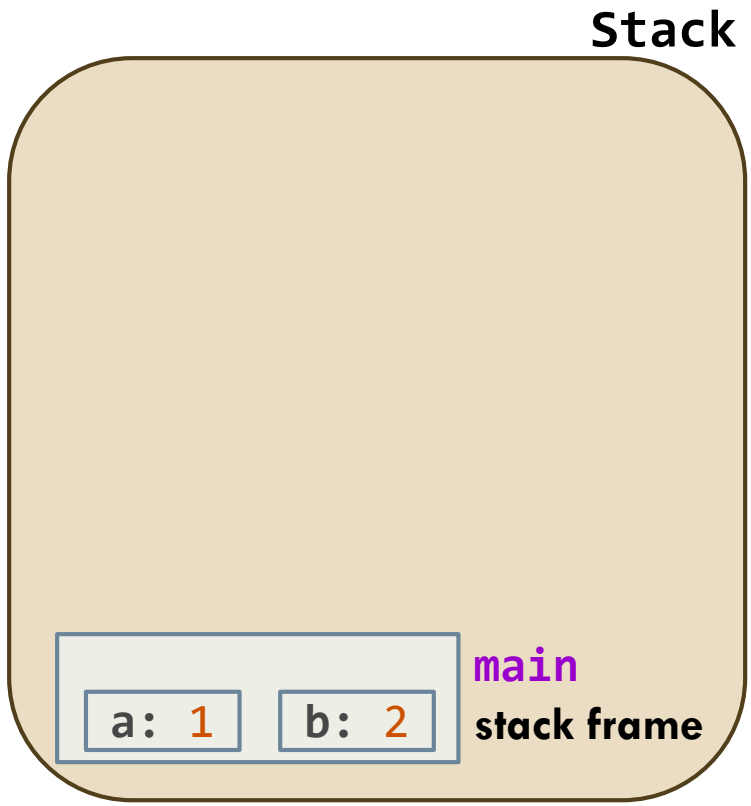
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



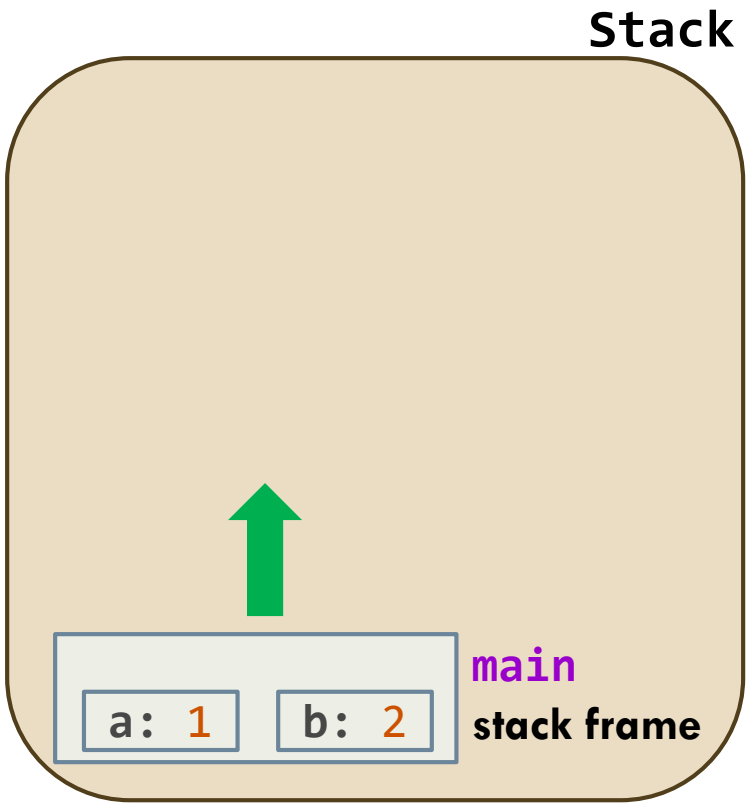
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



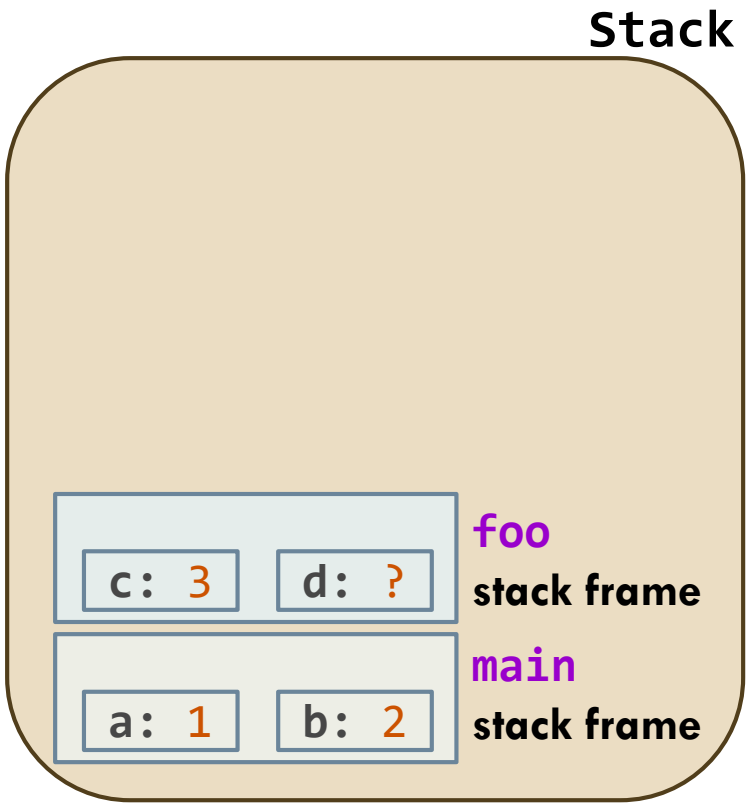
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



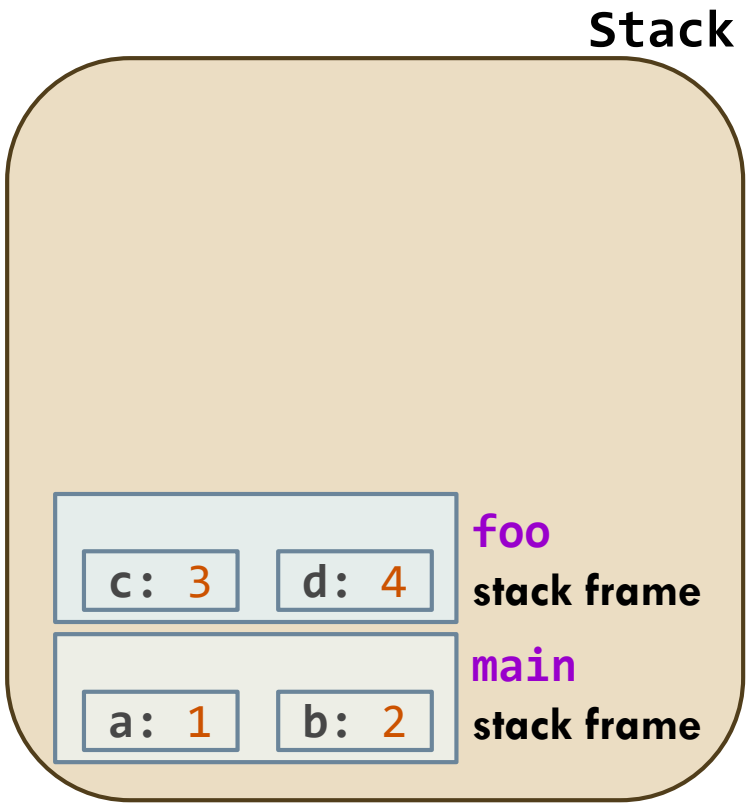
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



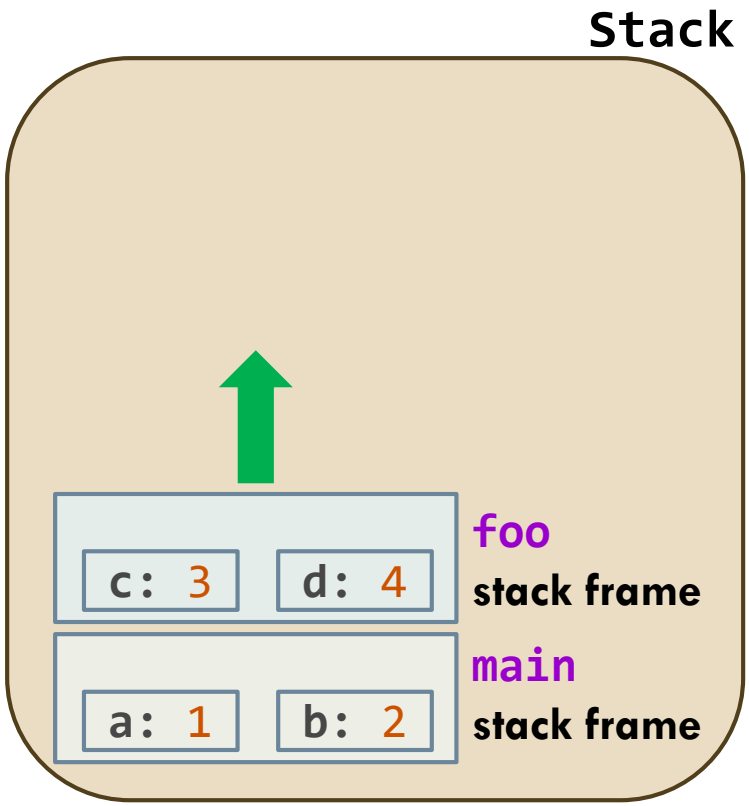
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



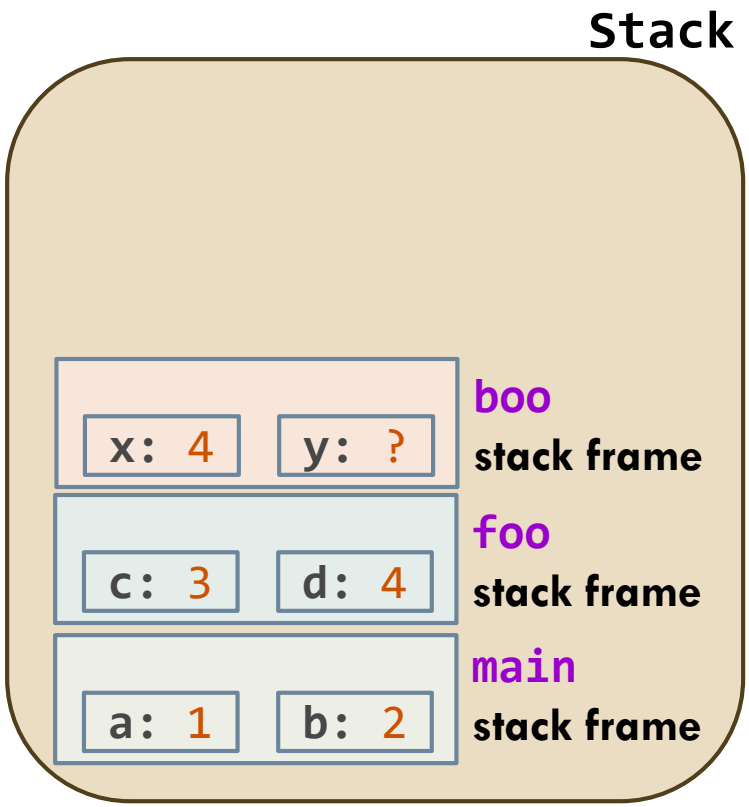
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



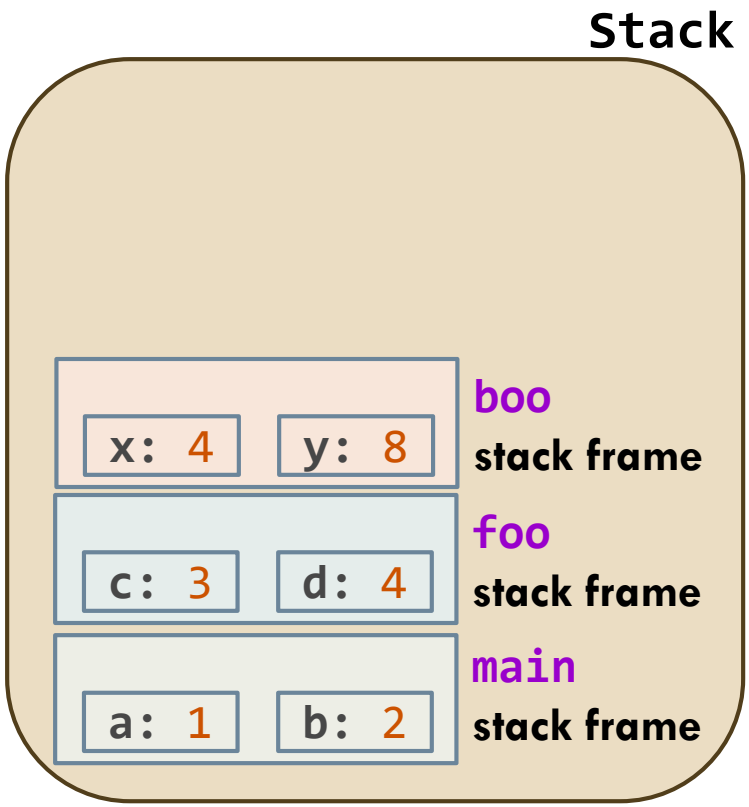
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



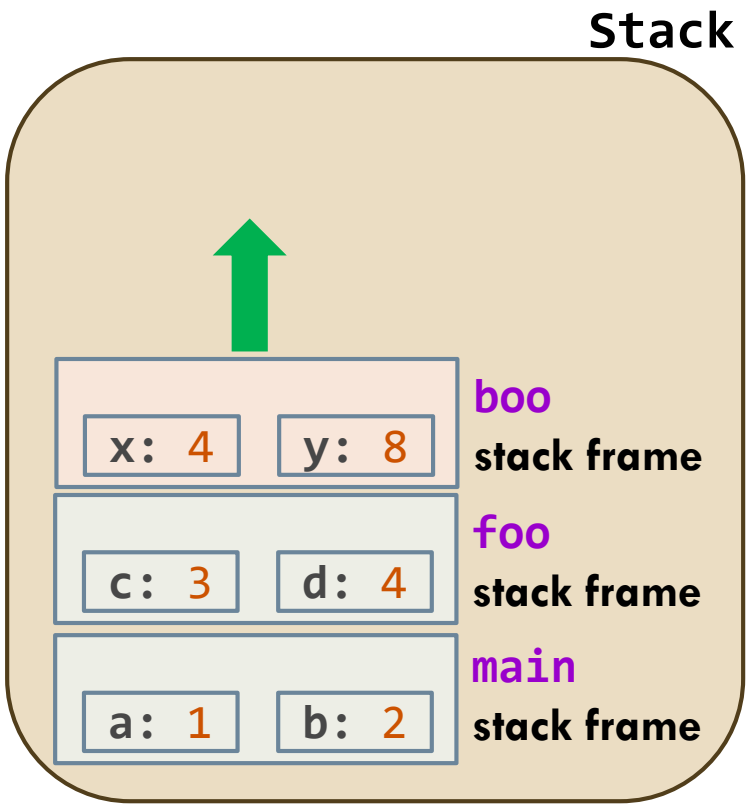
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



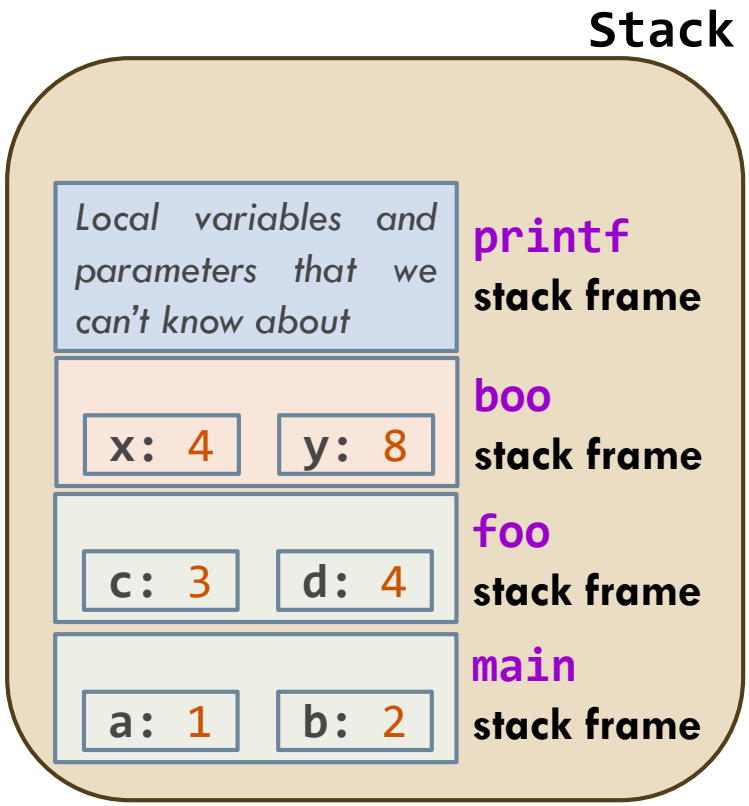
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



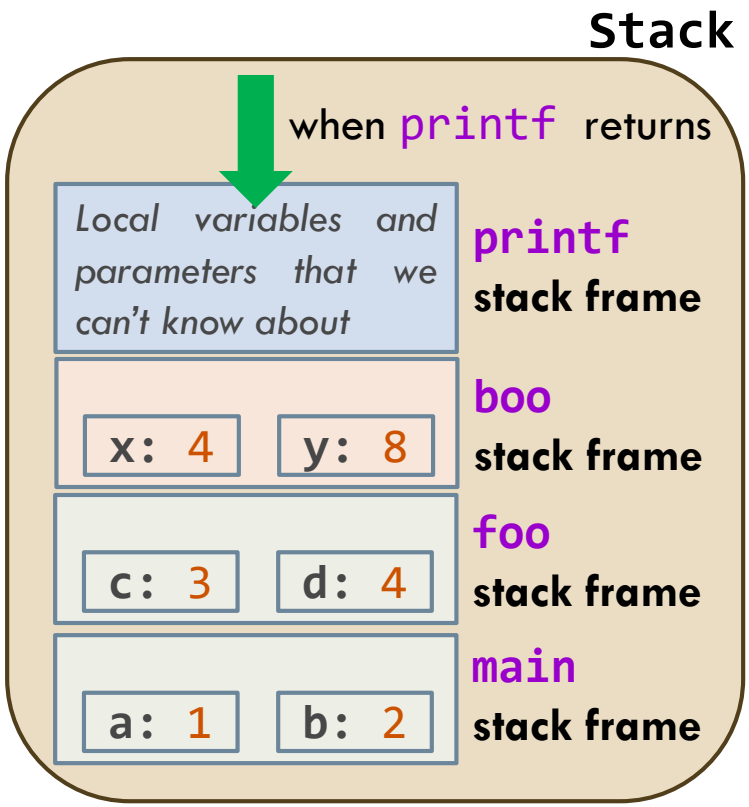
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



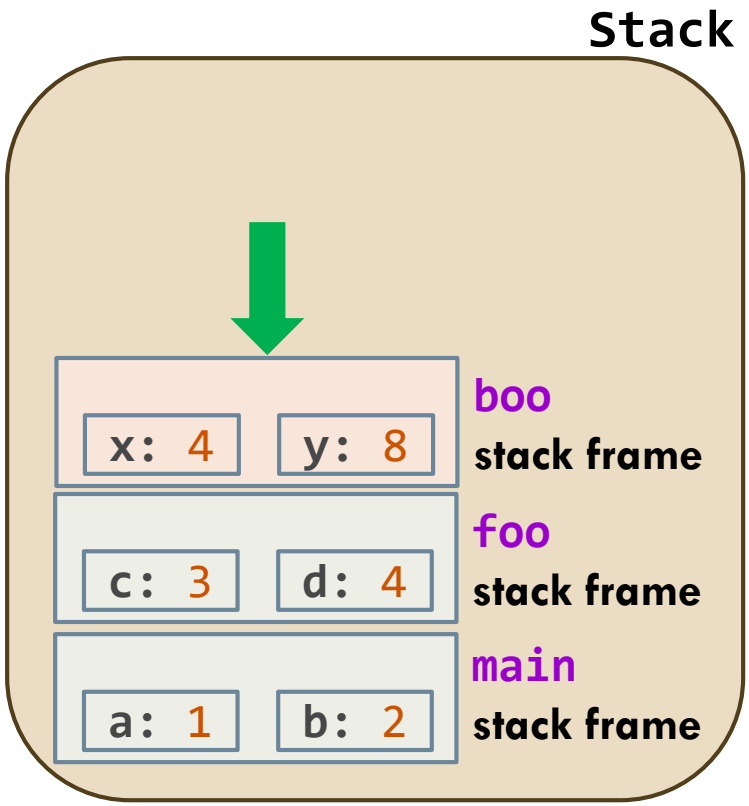
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



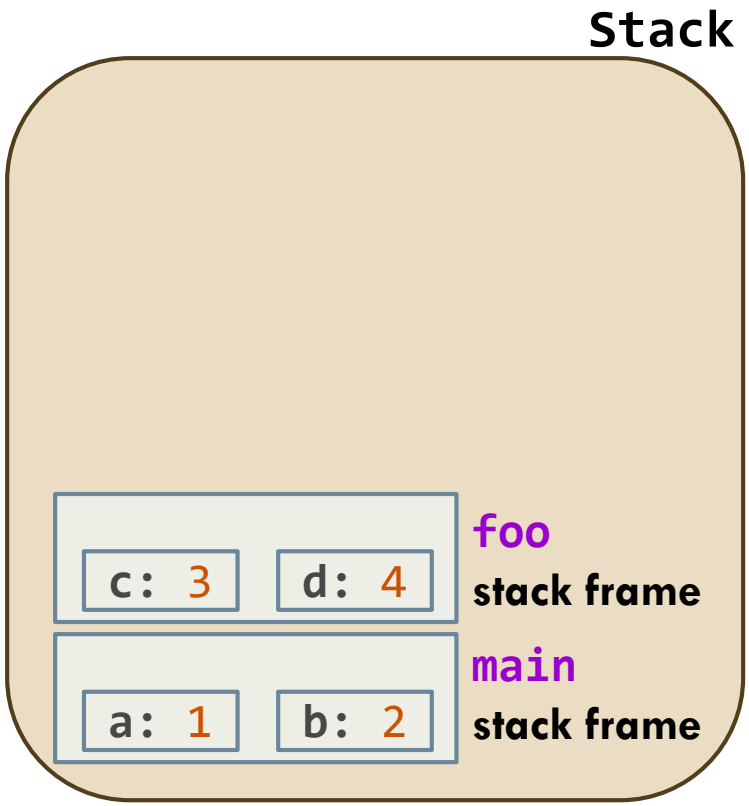
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



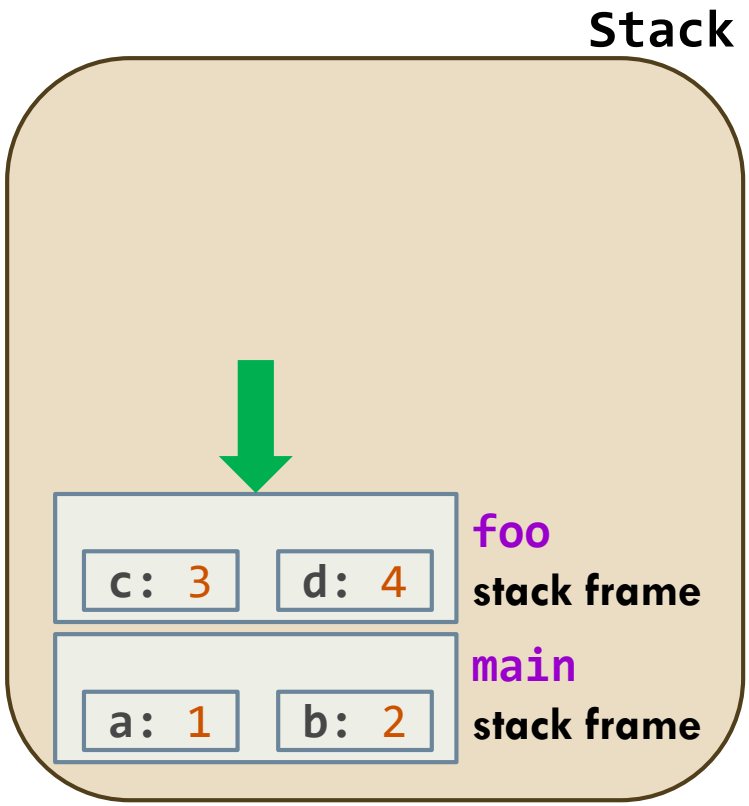
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



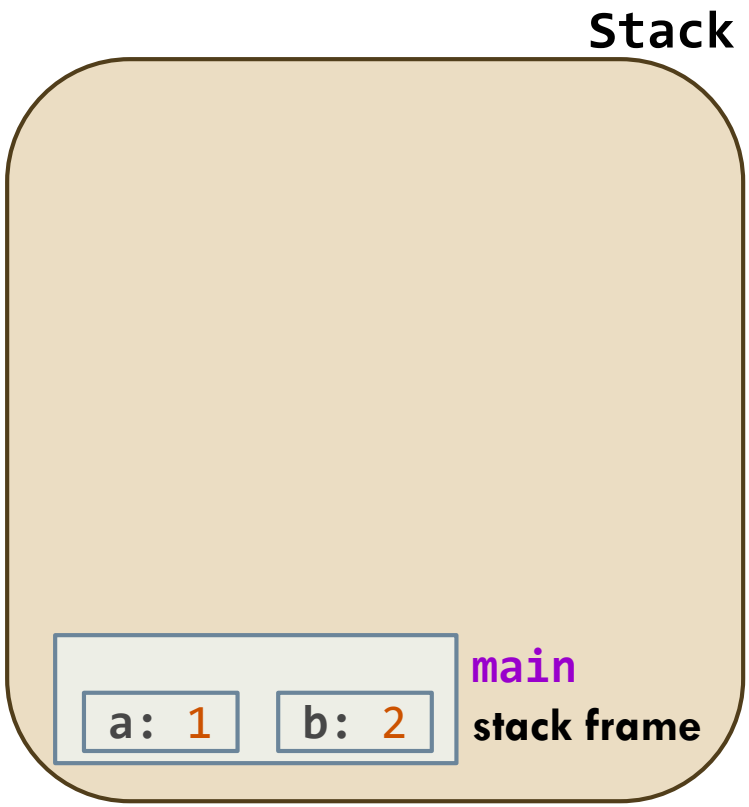
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



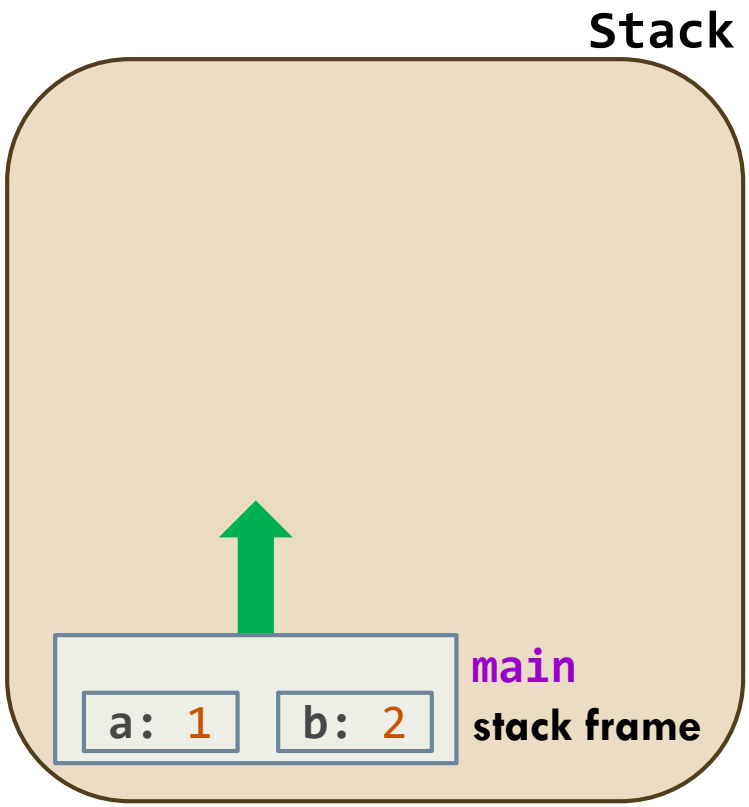
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



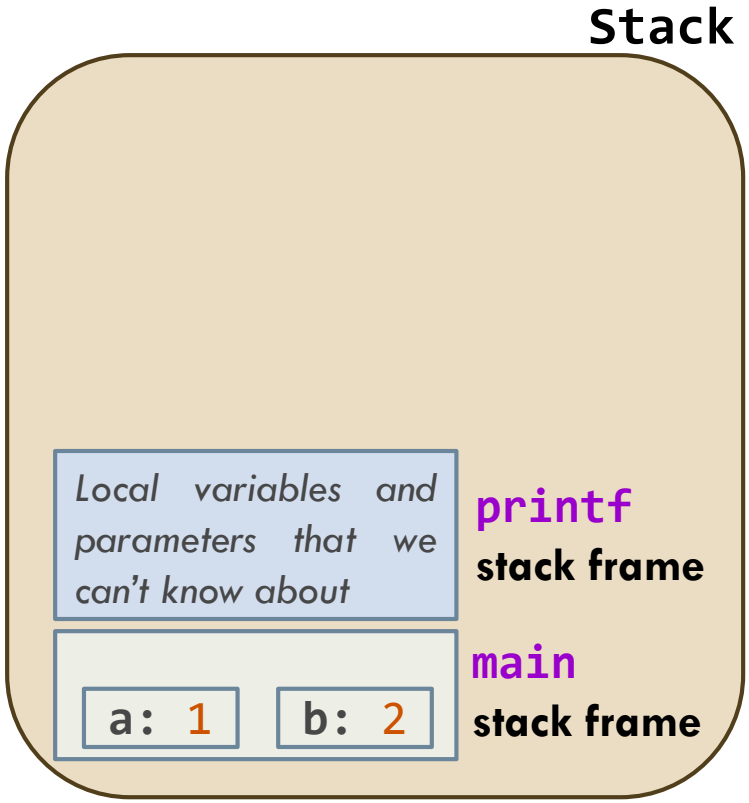
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



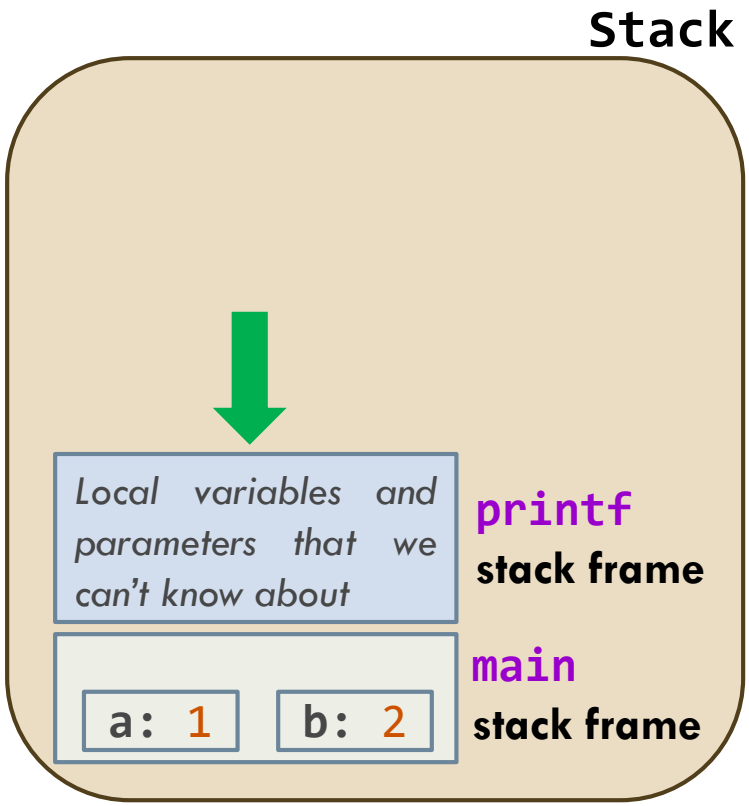
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



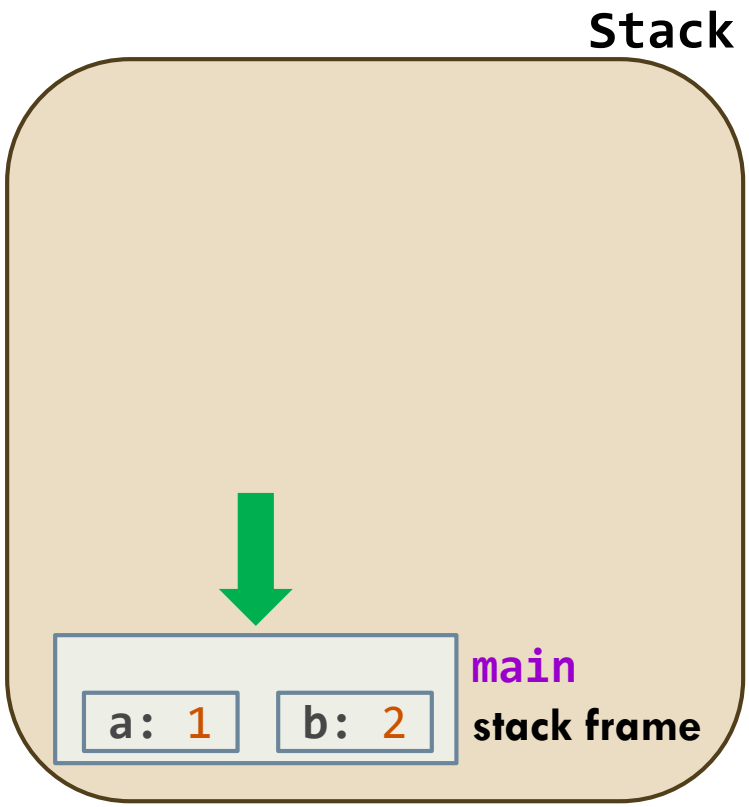
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



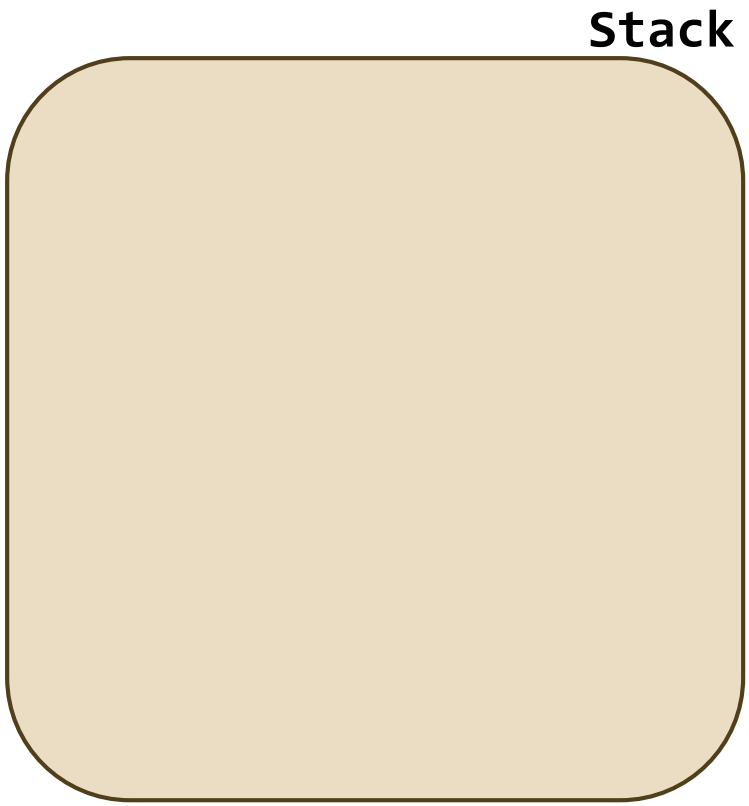
The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```



The Stack


```c
#include <stdio.h>

void boo(int x) {
  int y = x * 2;
  printf("y: %d\n", y);
}

void foo(int c) {
  int d = c + 1;
  boo(d);
}

int main(void) {
  int a = 1, b = 2;
  foo(a+b); // call to function foo
  printf("a+b: %d\n", a+b);
  return 0;
}
```

Main takeaways:

Each function call has separate stack frame for its own copy of variables.

These variables are called ***local variables***.

Local variables are ***not visible nor accessible*** outside the scope of the function.

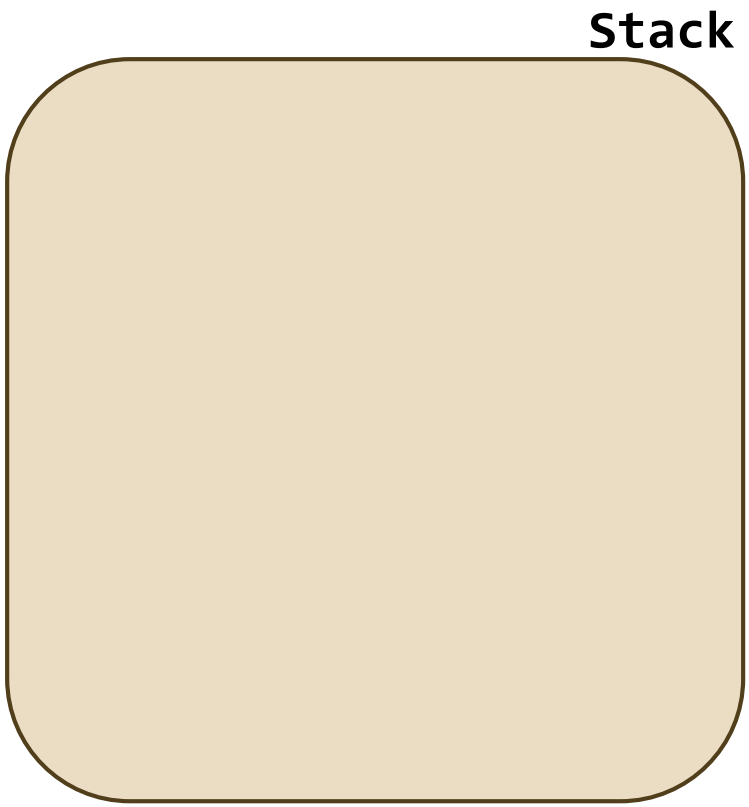
Local variables come alive when function is invoked and die when function returns.

The Stack: Limitations


```c
#include <stdio.h>

char* strfactory(char c, int n) {
  char str[6];
  for (int i = 0; i < n; ++i) {
    str[i] = c;
  }
  str[n] = '\0';
  return str;
}

int main() {
  char *ps = strfactory('z', 5);
  printf("%s", ps);
  return 0;
}
```

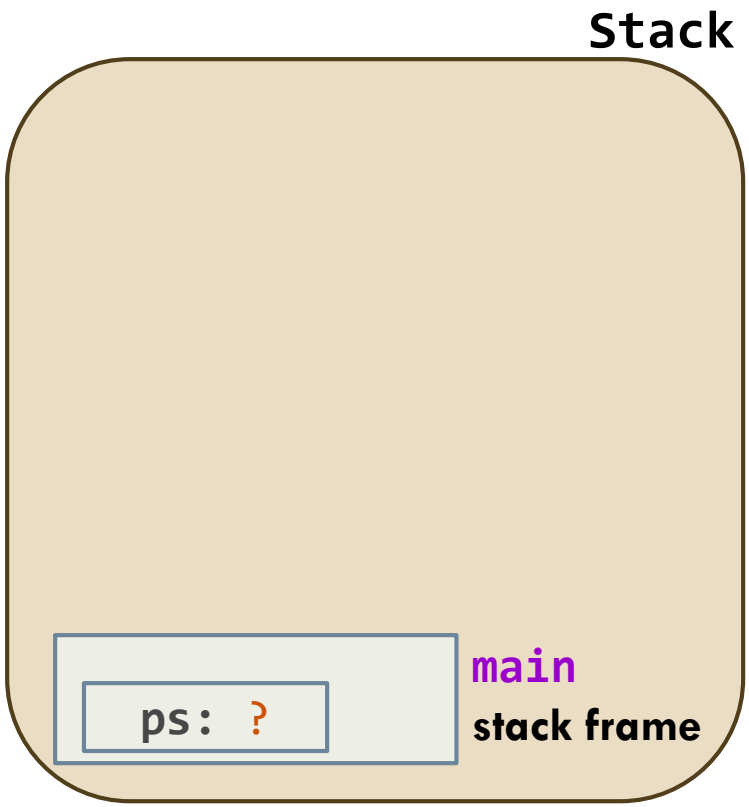


The Stack: Limitations


```c
#include <stdio.h>

char* strfactory(char c, int n) {
  char str[6];
  for (int i = 0; i < n; ++i) {
    str[i] = c;
  }
  str[n] = '\0';
  return str;
}

int main() {
  char *ps = strfactory('z', 5);
  printf("%s", ps);
  return 0;
}
```

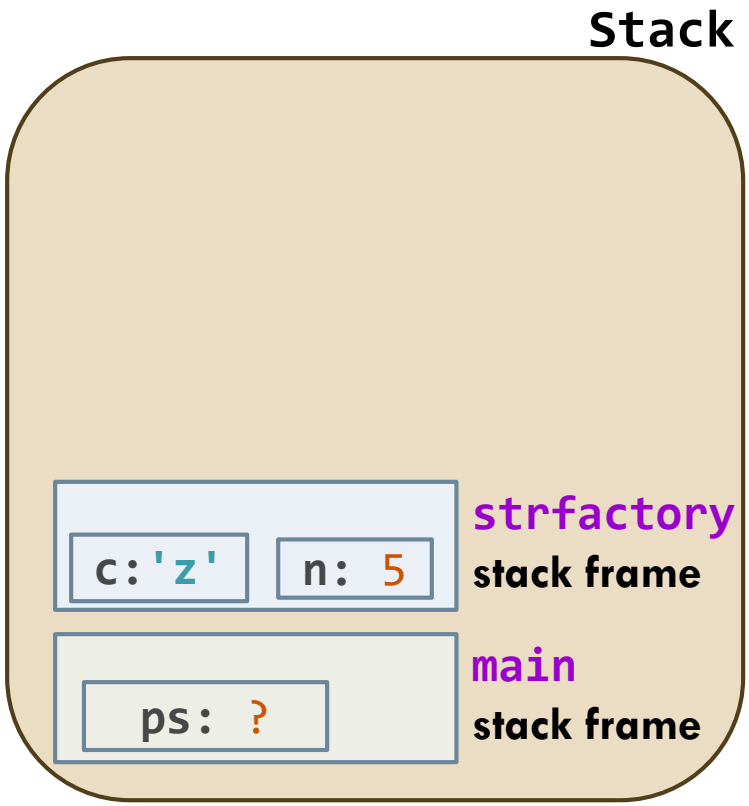


The Stack: Limitations


```c
#include <stdio.h>

char* strfactory(char c, int n) {
  char str[6];
  for (int i = 0; i < n; ++i) {
    str[i] = c;
  }
  str[n] = '\0';
  return str;
}

int main() {
  char *ps = strfactory('z', 5);
  printf("%s", ps);
  return 0;
}
```

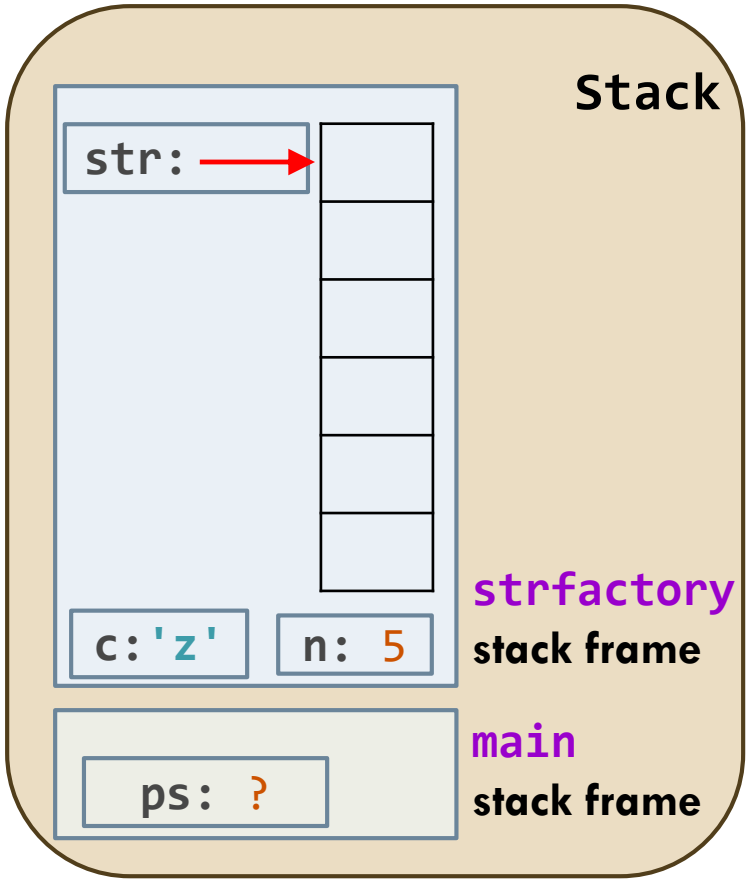


The Stack: Limitations


```
#include <stdio.h>

char* strfactory(char c, int n) {
  char str[6];
  for (int i = 0; i < n; ++i) {
    str[i] = c;
  }
  str[n] = '\0';
  return str;
}

int main() {
  char *ps = strfactory('z', 5);
  printf("%s", ps);
  return 0;
}
```

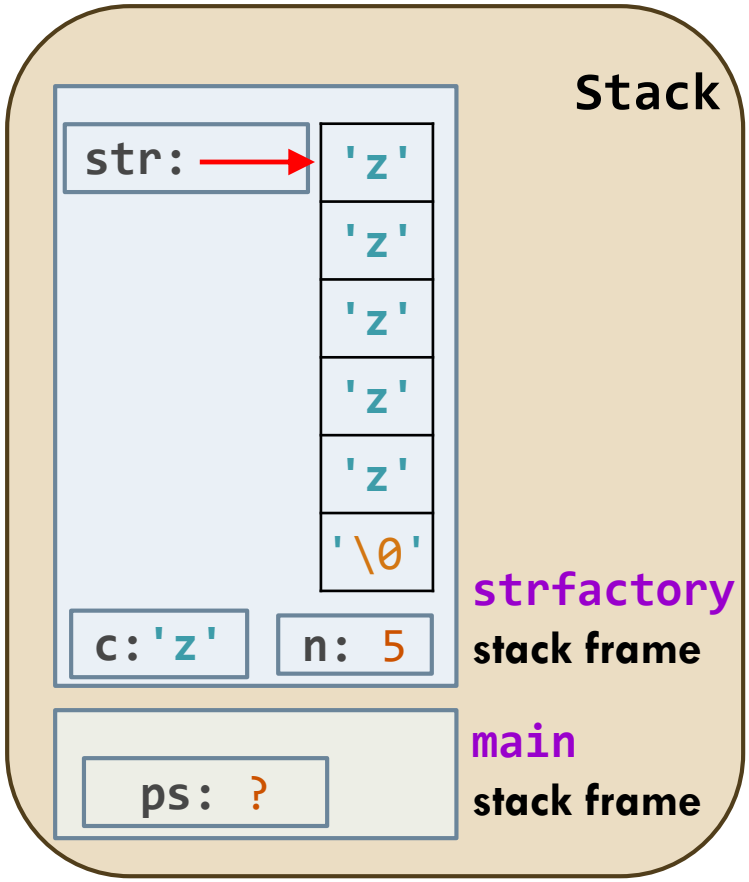


The Stack: Limitations


```c
#include <stdio.h>

char* strfactory(char c, int n) {
  char str[6];
  for (int i = 0; i < n; ++i) {
    str[i] = c;
  }
  str[n] = '\0';
  return str;
}

int main() {
  char *ps = strfactory('z', 5);
  printf("%s", ps);
  return 0;
}
```



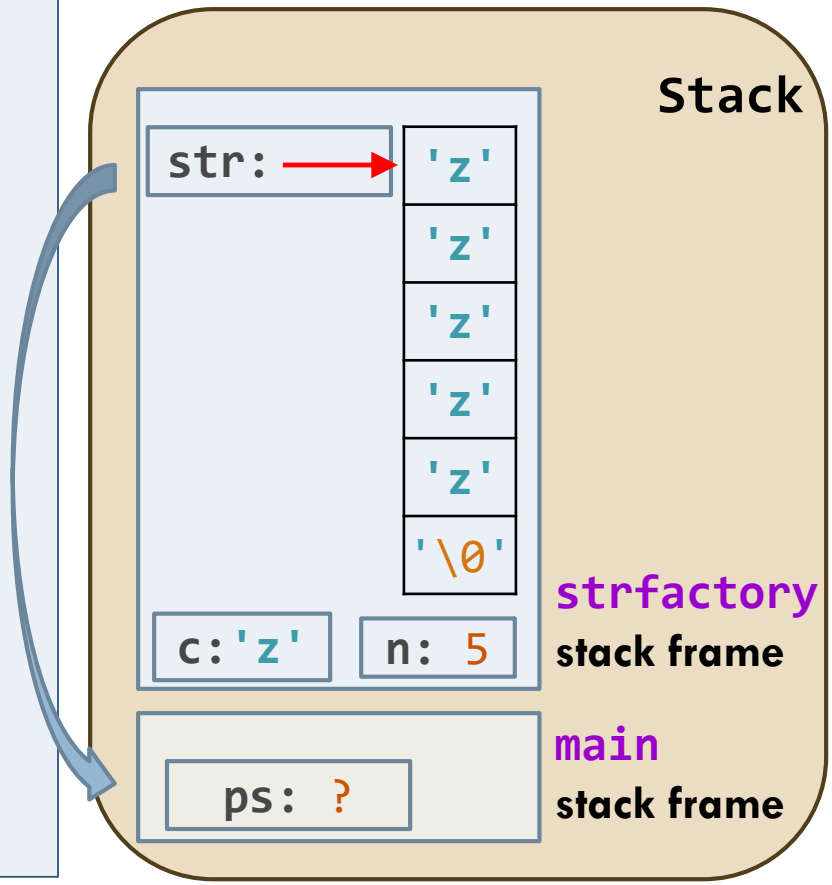
The Stack: Limitations


```
#include <stdio.h>

char* strfactory(char c, int n) {
  char str[6];
  for (int i = 0; i < n; ++i) {
    str[i] = c;
  }
  str[n] = '\0';
  return str;
}

int main() {
  char *ps = strfactory('z', 5);
  printf("%s", ps);
  return 0;
}
```

Returns base address of array `str` which will be used to initialize ps in `main()`

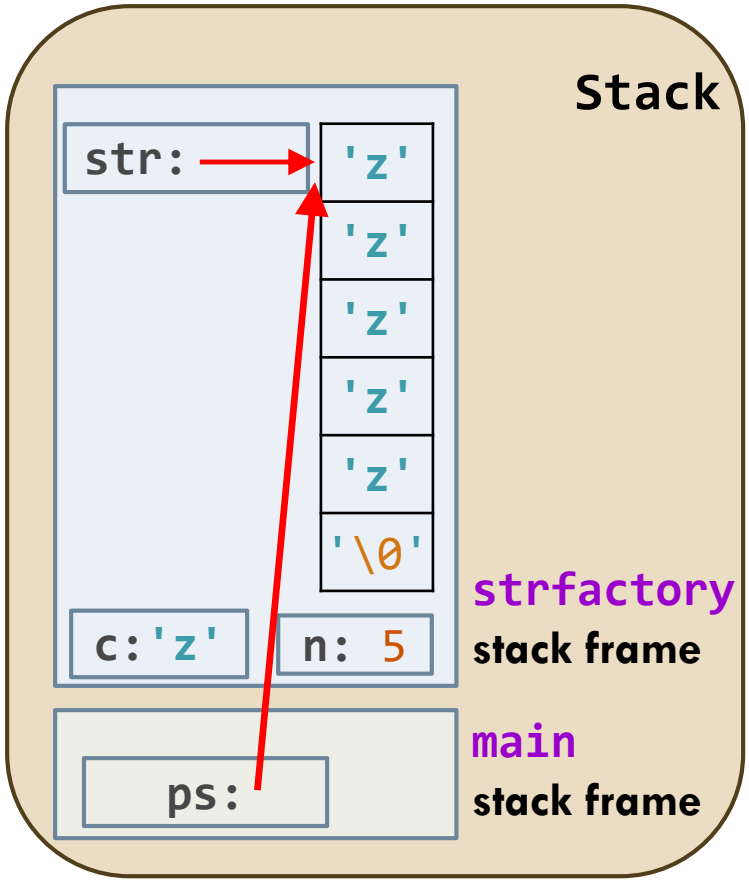


The Stack: Limitations


```c
#include <stdio.h>

char* strfactory(char c, int n) {
  char str[6];
  for (int i = 0; i < n; ++i) {
    str[i] = c;
  }
  str[n] = '\0';
  return str;
}

int main() {
  char *ps = strfactory('z', 5);
  printf("%s", ps);
  return 0;
}
```

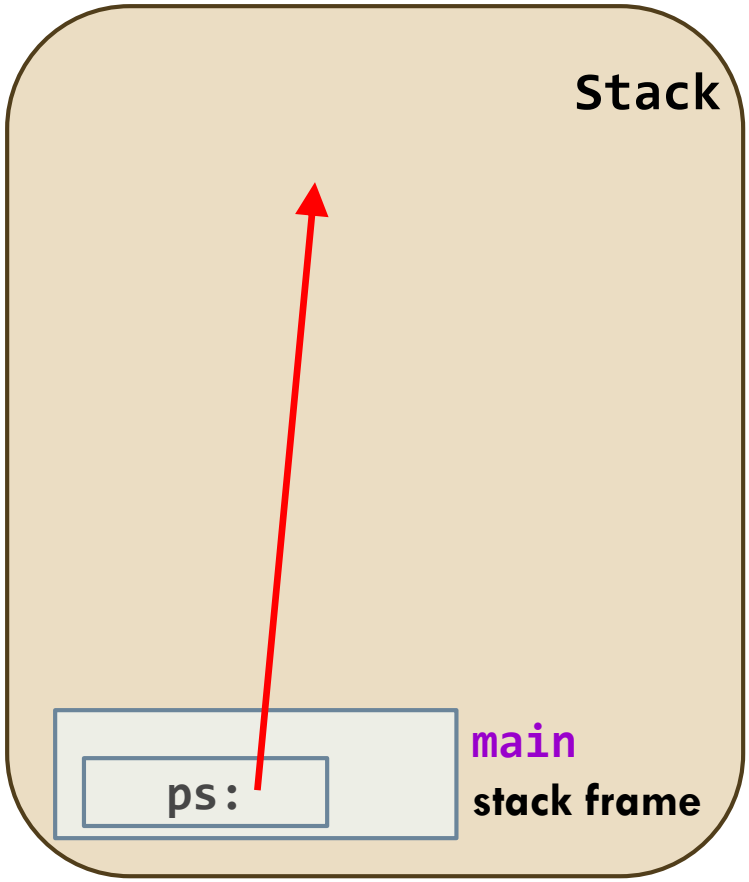


The Stack: Limitations


```c
#include <stdio.h>

char* strfactory(char c, int n) {
  char str[6];
  for (int i = 0; i < n; ++i) {
    str[i] = c;
  }
  str[n] = '\0';
  return str;
}

int main() {
  char *ps = strfactory('z', 5);
  printf("%s", ps);
  return 0;
}
```



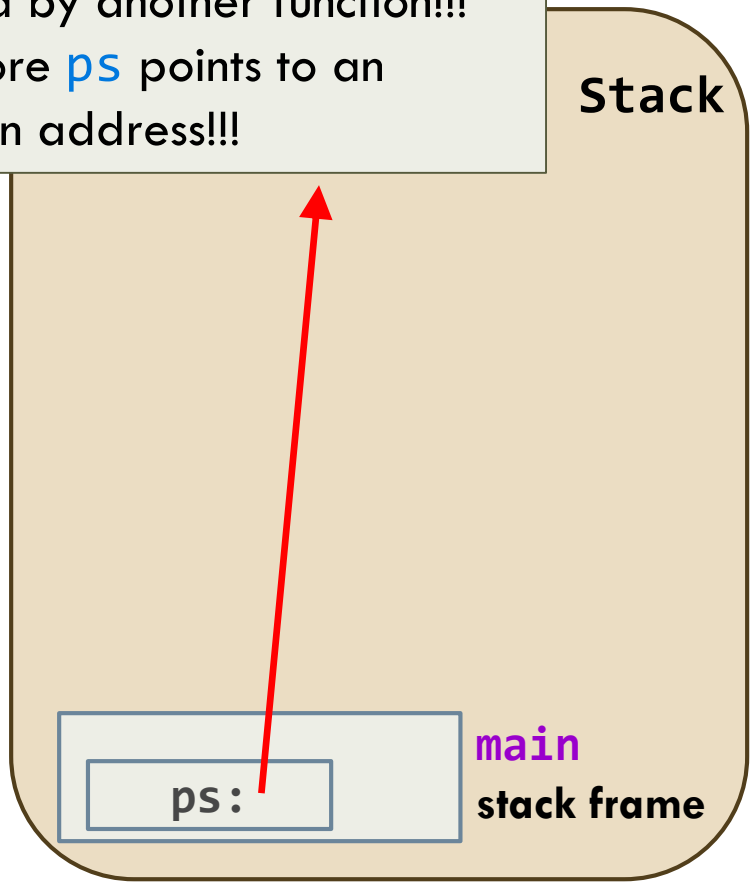
The Stack: Limitations

Local variables go away when function `strfactory` returns!!!
Array `str` no longer exists!!!
The memory exists but it will be used by another function!!!
Therefore `ps` points to an unknown address!!!

```c
#include <stdio.h>

char* strfactory(char c, int n) {
  char str[6];
  for (int i = 0; i < n; ++i) {
    str[i] = c;
  }
  str[n] = '\0';
  return str;
}

int main() {
  char *ps = strfactory('z', 5);
  printf("%s", ps);
  return 0;
}
```



.stack segment

- Fact that local variables are cleaned up when function returns is sometimes a problem
- How can we have memory that exists independent of whether a function is executing or not?

.heap segment

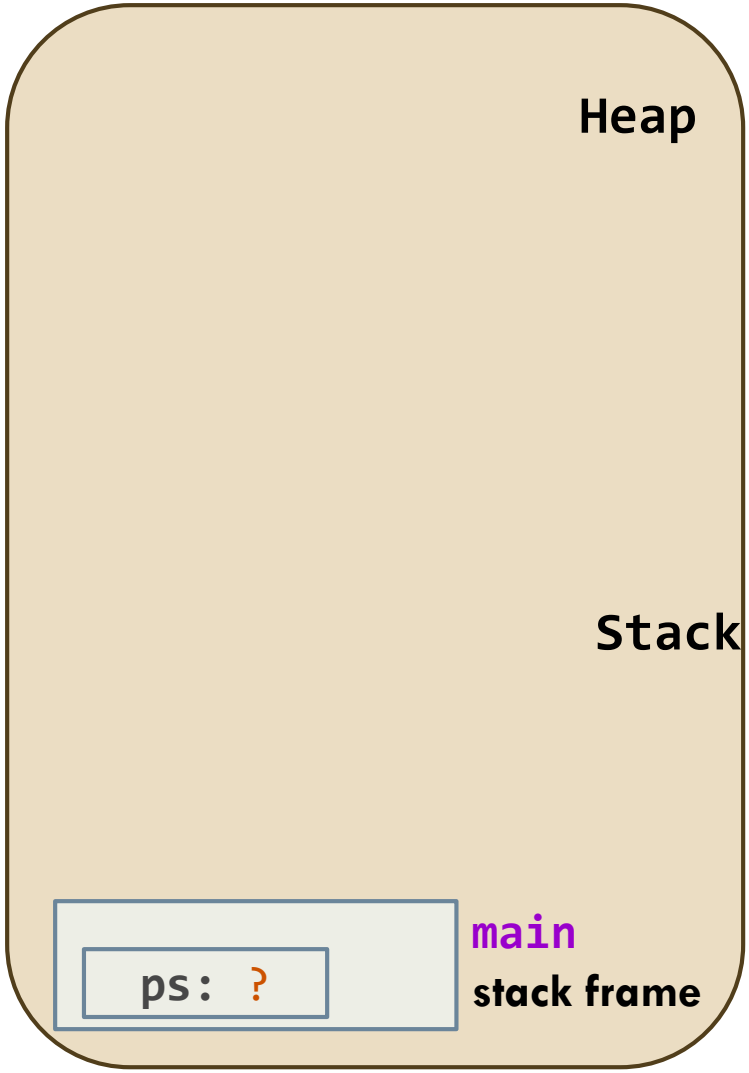
- Part of program memory that is managed by programmer
- You grab some memory, pass address around among functions, and then you return that memory back to heap manager

The Heap


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

char* str_factory(char ch, int num) {
  char *str = malloc(num+1);
  for (int i = 0; i < num; ++i) {
    str[i] = ch;
  }
  str[num] = '\0';
  return str;
}

int main() {
  char *ps = str_factory('z', 5);
  printf("%s\n", ps);
  return 0;
}
```

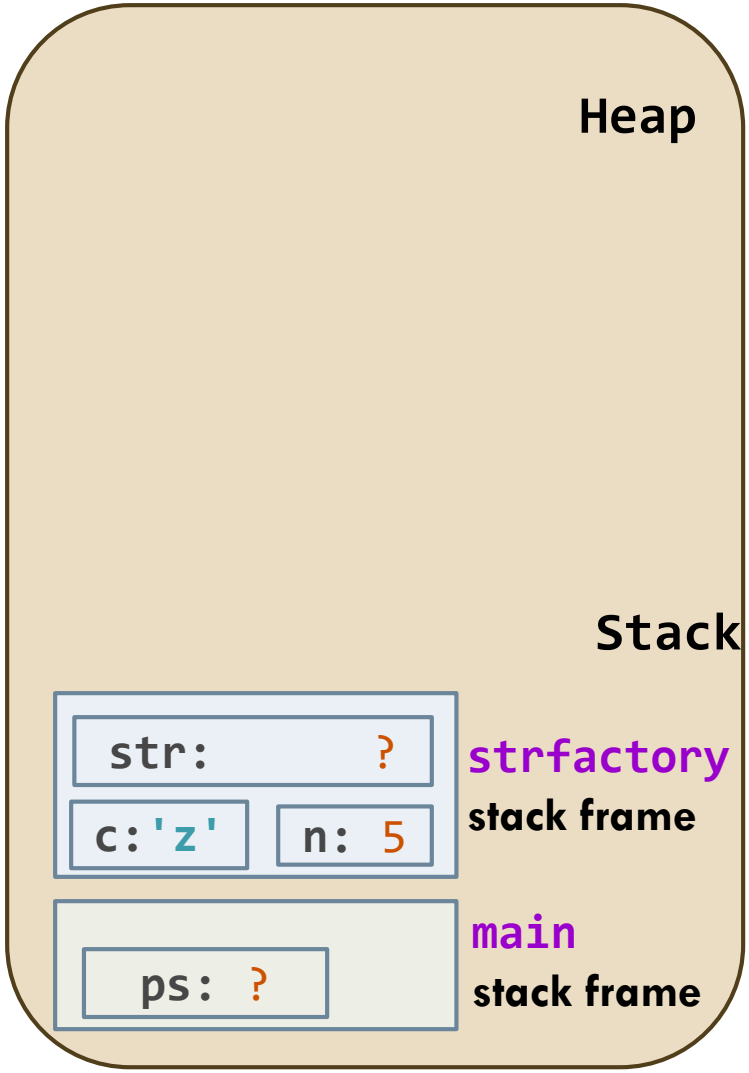


The Heap


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

char* str_factory(char ch, int num) {
  char *str = malloc(num+1);
  for (int i = 0; i < num; ++i) {
    str[i] = ch;
  }
  str[num] = '\0';
  return str;
}

int main() {
  char *ps = str_factory('z', 5);
  printf("%s\n", ps);
  return 0;
}
```

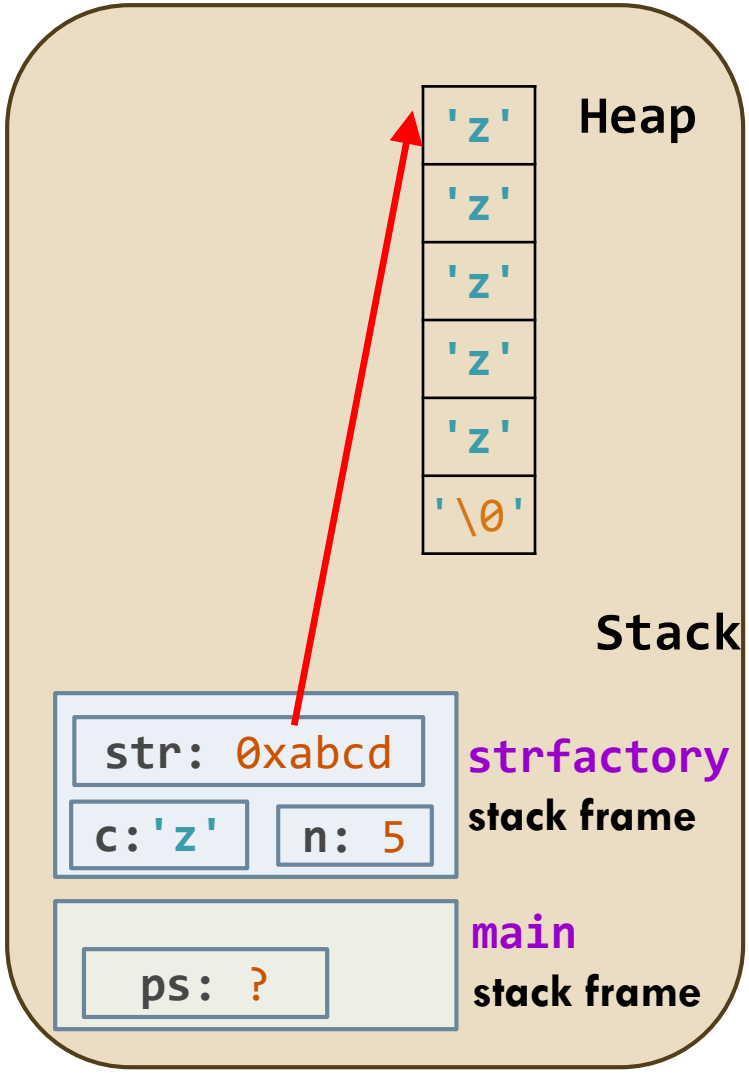


The Heap


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

char* str_factory(char ch, int num) {
  char *str = malloc(num+1);
  for (int i = 0; i < num; ++i) {
    str[i] = ch;
  }
  str[num] = '\0';
  return str;
}

int main() {
  char *ps = str_factory('z', 5);
  printf("%s\n", ps);
  return 0;
}
```



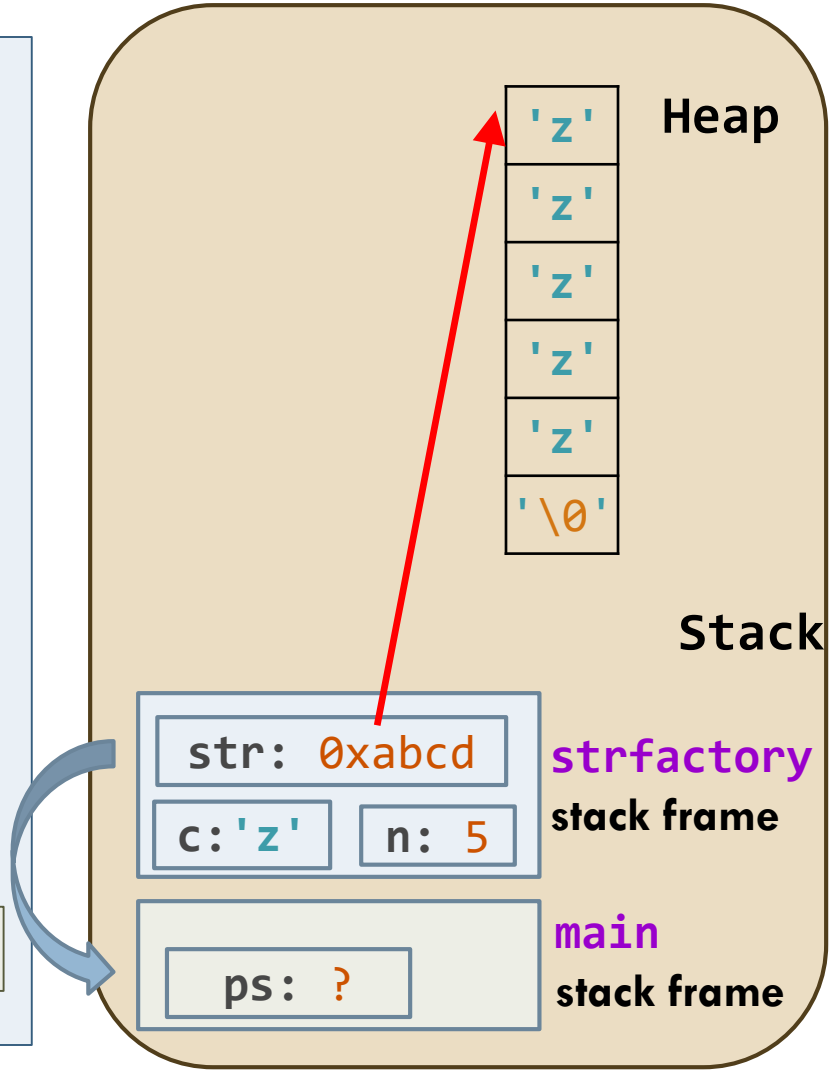
The Heap


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

char* str_factory(char ch, int num) {
  char *str = malloc(num+1);
  for (int i = 0; i < num; ++i) {
    str[i] = ch;
  }
  str[num] = '\0';
  return str;
}

int main() {
  char *ps = str_factory('z', 5);
  printf("%s\n", ps);
  return 0;
}
```

Returns address 0xabcd

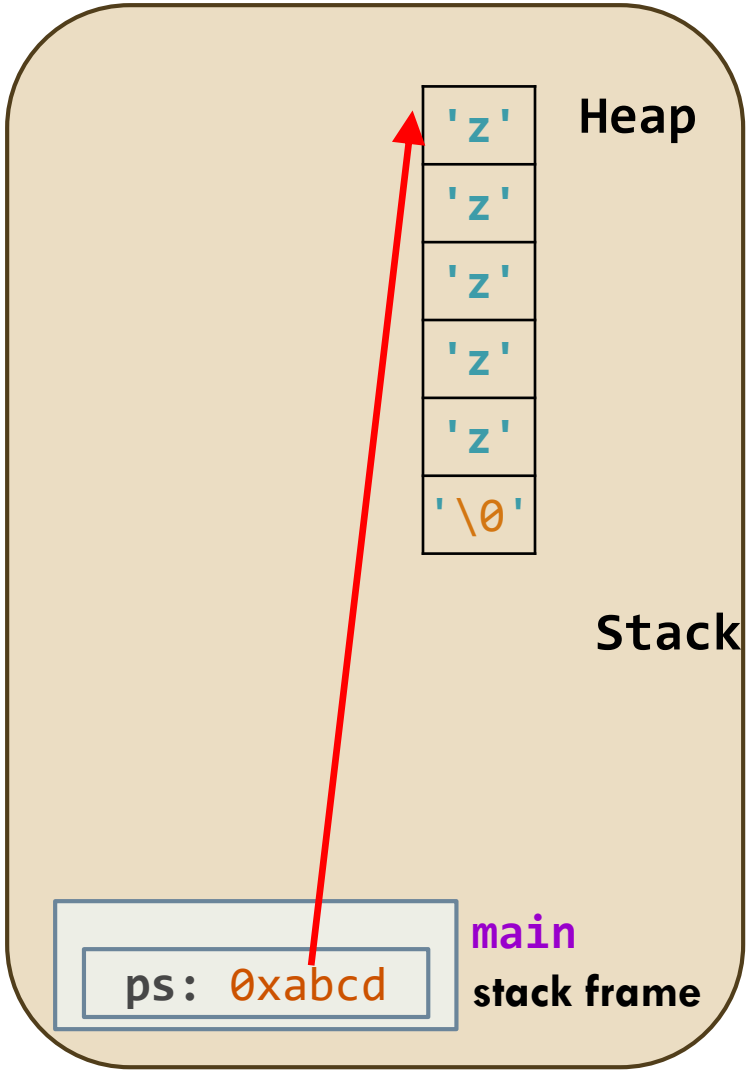


The Heap


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

char* str_factory(char ch, int num) {
  char *str = malloc(num+1);
  for (int i = 0; i < num; ++i) {
    str[i] = ch;
  }
  str[num] = '\0';
  return str;
}

int main() {
  char *ps = str_factory('z', 5);
  printf("%s\n", ps);
  return 0;
}
```

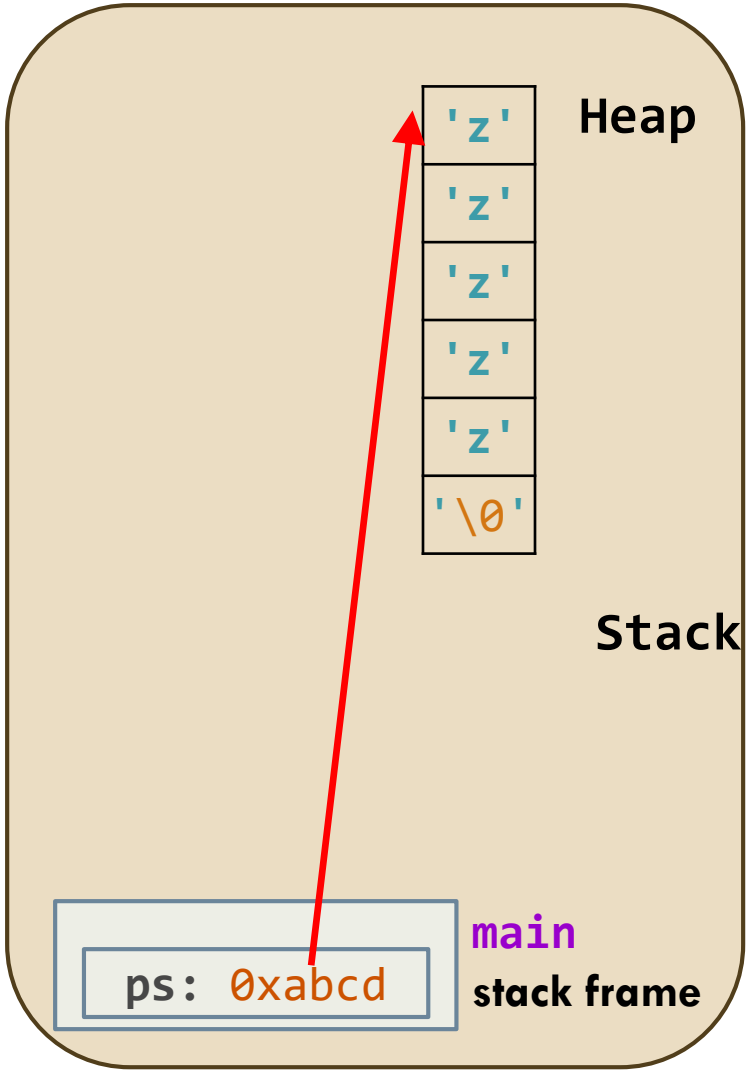


The Heap


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

char* str_factory(char ch, int num) {
  char *str = malloc(num+1);
  for (int i = 0; i < num; ++i) {
    str[i] = ch;
  }
  str[num] = '\0';
  return str;
}

int main() {
  char *ps = str_factory('z', 5);
  printf("%s\n", ps);
  return 0;
}
```



Heap Memory Allocation & Deallocation Functions


```
// declared in <stdlib.h>

// functions for dynamically allocating heap memory
void *malloc(size_t size);
void *calloc(size_t count, size_t size);
void *realloc(void *ptr, size_t size);

// function for returning dynamically allocated
// memory back to heap
void free(void *ptr);
```

Memory Errors

- Dynamic memory is error-prone!!!
- Possible problems:
  - Leaked or orphaned memory
  - Premature deletion
  - Double deletion
  - Dereferencing uninitialized pointers
  - Accessing freed memory