HIGH-LEVEL PROGRAMMING I

Functions

by Prasanna Ghali

References

- Chapter 7 of the text book

Consider Complex Problem

- You have to send flowers to your grandmother (who lives in Japan) for her birthday
  - Plant flowers
  - Water flowers
  - Pick flowers
  - Fly to Japan with flowers

Another Complex Problem

- You're asked to organize catering for a wedding
  - Make up guest list
  - Invite guests
  - Select appropriate menu
  - Book reception hall
  - …

Another Complex Problem

- You're asked to organize catering for a wedding
  - Make up guest list
    - Get list from groom
    - Get list from bride
    - Check for conflicts
      - Check with bride about groom's list
      - Check with groom about bride's list
      - Check final list with groom's parents
      - Check final list with bride's parents
      - …
  - Invite guests
    - …
  - Select appropriate menu
  - Book reception hall
  - …

Procedural Programming Paradigm (1/2)

- Breaking down tasks into smaller subtasks is good plan of attack for solving complex programming problems too
  - Each "large" task is decomposed into smaller subtasks and so forth
  - Process is continued until subtask can be implemented by single ***algorithm***
- Synonyms for this strategy: ***top-down design, procedural abstraction, functional decomposition, divide-and-conquer, stepwise refinement***

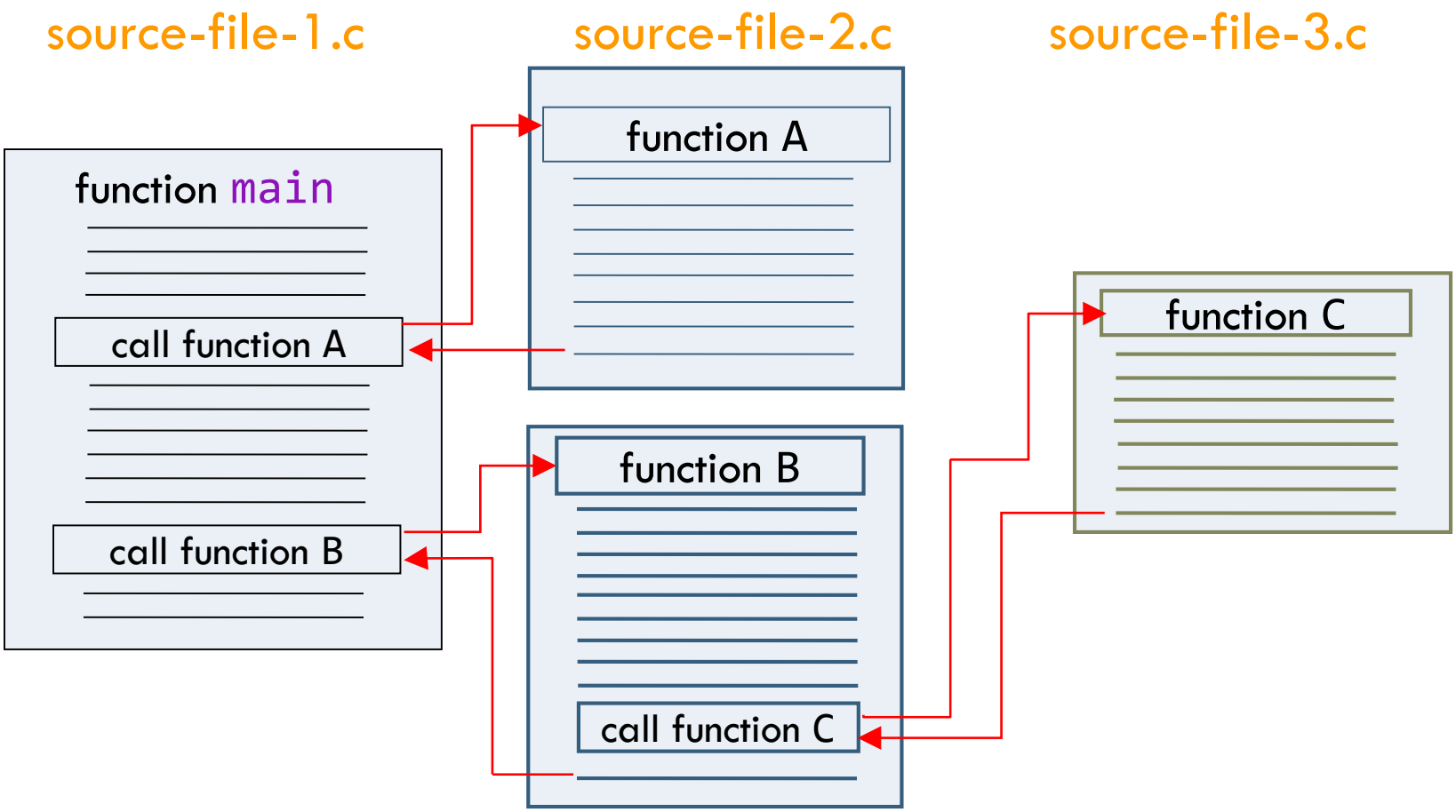
Procedural Programming Paradigm (2/2)

- In C/C++, algorithm is packaged into building block called ***function***
  - Other languages refer to function as ***procedure*** or ***subroutine*** or ***method***
- Program is organized into these smaller, independent units called ***functions***

Advantages of Functions

- Divide and conquer approach to complexity
  - Divide complicated whole into simpler and more manageable units
  - Standalone, independent functions are easier to understand, implement, maintain, test and debug
- Cost and pace of development
  - Different people can work on different functions simultaneously
- Building blocks for programs
  - Write function once and use it many times in same program or many other programs

Organization of C Programs (1/2)



- Every C program must have *one and only one* function called `main` → not a C/C++ keyword!!!

Organization of C Programs (2/2)

- Related functions are organized into a source file
- Think of C program as one or more source files with each source file containing one or more related functions

```
// source-file-1.c
preprocessing directives

function prototypes/declarations

data declarations (global)

return-type
main (parameter declarations)
{
  data declarations (local)
  statements
}

other functions
```

...

```
// source-file-n.c
preprocessing directives

function prototypes/declarations

data declarations (global)

return-type
function-name (parameter declarations)
{
  data declarations (local)
  statements
}

other functions
```

Function Prototype/Declaration

- To use particular function in your program, *function prototype* (in C) or *function declaration* (in C++) must be known:
  - Name of function
  - Number of inputs – each input is called *parameter*
  - Data type of each *parameter*
  - Data type of function - type of value returned by function

```
// somewhere in math.h
double sqrt(double);
```

this line is called *function prototype* in C and *function declaration* in C++

General Syntax of Function Prototype or Declaration


```
function-return-type function-name(parameter-list);
```

```
double area(double width, double height);
int    volume(int width, int height, int depth);
double cube(double);
```

Function Definition (1/2)

- To use particular function in your program, *function prototype* (in C) or *function declaration* (in C++) must be known:
  - Name of function
  - Number of inputs – each input is called *parameter*
  - Data type of each *parameter*
  - Data type of function - the type of value returned by function
- One more thing is required during linking to generate executable – the code that will implement the algorithm encapsulated by function

Function Definition (2/2)

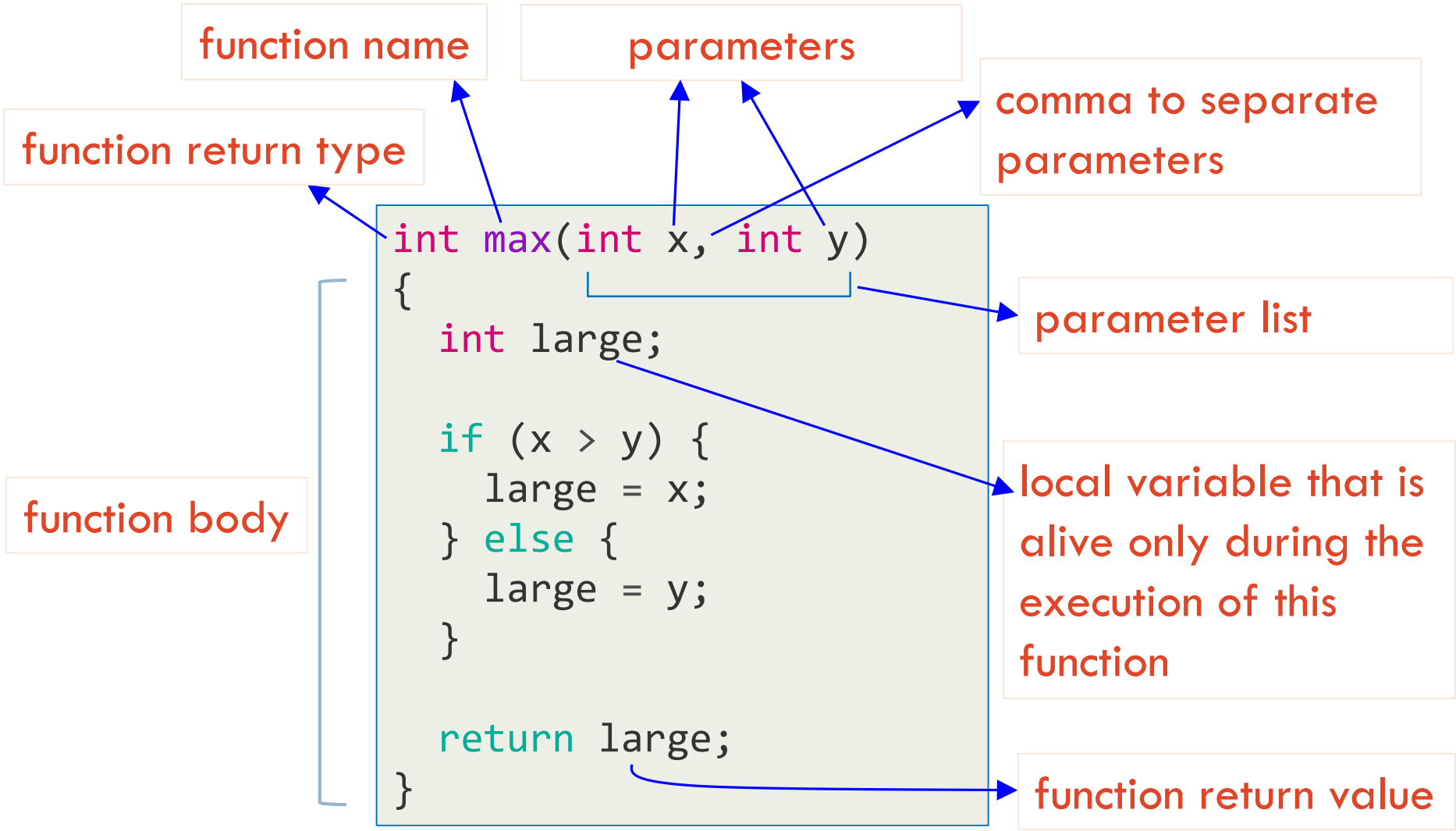
function definition = function prototype + code

```
return-type function-name(parameter-list)
{
   statements
}
```

this variable is called *formal parameter* or just *parameter*

```
int myabs(int number) {
  return number < 0 ? -number : number;
}
```

Parts of Function Definition



Function Call Operator

Prototype of function
sqrt is in <math.h>

( ) is called *function call* operator

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

int main(void) {
  double value = 9.0;
  double root = sqrt(value);
  printf("Root of %f is %f\n", value, root);
  return 0;
}
```

function name

this expression is called *function argument*

Will Program Compile? Link? (1/3)


```c
#include <stdio.h>

int main(void) {
  double x = 10.0, y = 20.0;
  double z = average(x, y);

  printf("Average is: %d\n", z);
  return 0;
}

double
average(double x, double y) {
  return (x+y)/2.0;
}
```

In C, compiler will assume **average** is function that takes unknown number of parameters and returns an **int** value.

However, C++ compiler will not make any such assumptions and simply flag an error!!!

To make C compiler behave as C++ compiler, we'll use options –Wstrict-prototypes and –pedantic-errors!!!

Will Program Compile? Link? (2/3)


```c
#include <stdio.h>

double
average(double x, double y);

int main(void) {
  double x = 10.0, y = 20.0;
  double z = average(x, y);

  printf("Average is: %d\n", z);
  return 0;
}
```

This is function declaration!!!

In both C and C++, compiler will compile this source file because it finds a declaration for this function.
However, the linker will flag an error because the definition of function **average** is not present!!!

Will Program Compile? Link? (3/3)


```c
#include <stdio.h>

double
average(double x, double y) {
  return (x+y)/2.0;
}

int main(void) {
  double x = 10.0, y = 20.0;
  double z = average(x, y);

  printf("Average is: %d\n", z);
  return 0;
}
```

Every function definition is also a declaration!!!

In both C and C++, compiler will compile this source file because it finds a declaration for this function. The linker will generate an executable since it can find the definition for function **average**!!!

Examples


```c
// return, no parameters
double pi(void) {
  return 3.14159;
}
```

```c
// no return, no parameters
void say_hello(void) {
  printf("Hello!!!\n");
  return; // optional
}
```

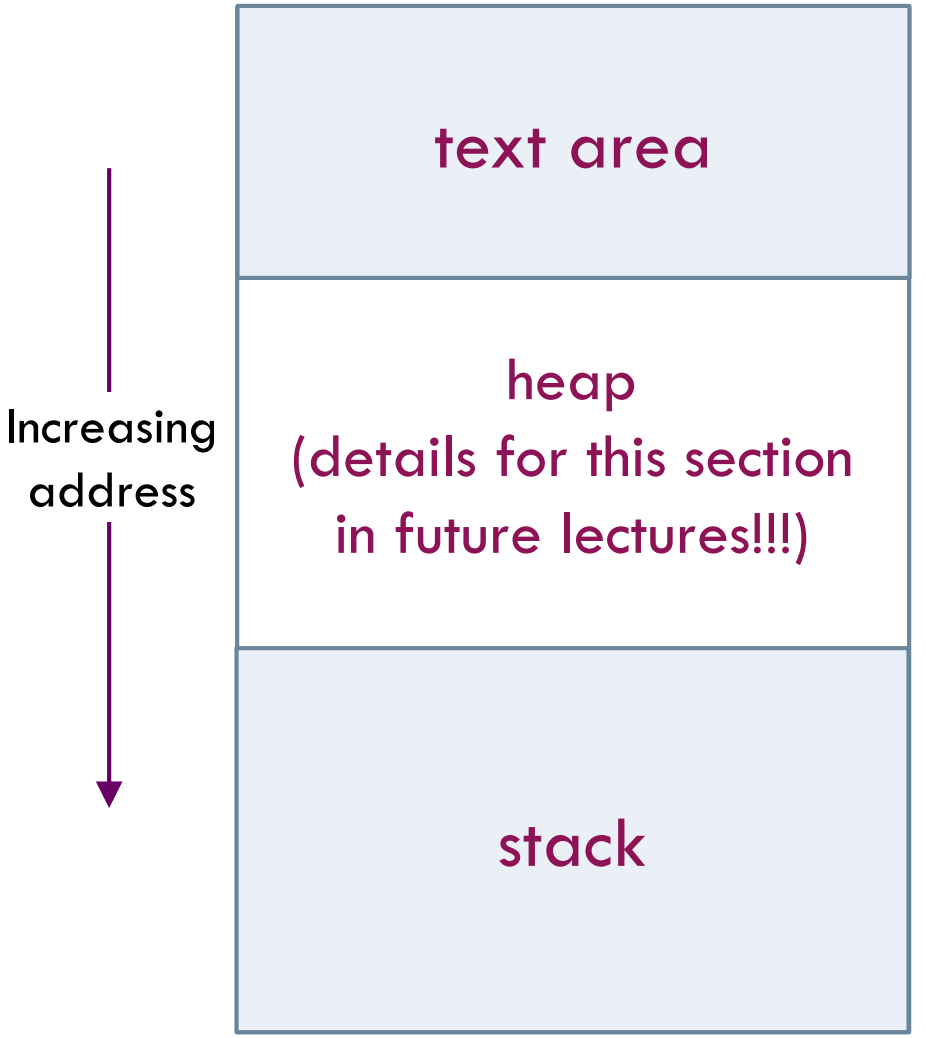
```c
// no return, 1 parameter
void say_hello_alot(int count) {
  for (int i = 0; i < count; ++i) {
    printf("Hello!!!\n");
  }
  // return statement is optional since
  // function is returning void
}
```

Calling Functions


```
void   say_hello(void);      // defined in previous slide
void   say_hello_alot(int);  // defined in previous slide
double pi(void);             // defined in previous slide

int main(void) {
  say_hello(); // ok: say_hello has no arguments
               // parentheses represent function call operator
  say_hello_alot(5);    // ok: one argument
  double d = pi()+56.0; // ok: pi has no arguments
  pi();                 // ok: caller can ignore return
                        // value from function
  d = pi;  // error: function call operator is required!!!
  pi;      // ok: pi without function call operator
           // means "pointer to function pi"
           // more on pointers later ...
  d = say_hello(); // error: say_hello has no return value!!!
  return 0;
}
```

Program Memory (1/2)



text area consists of machine language code for every function present in executable.

Programs make use of *stack* to support function call mechanism. Machine uses stack to pass arguments, return values, provide storage for local variables in functions, and save registers for later restoration.
Portion of stack allocated for single function call is called *stack frame*.

Program Memory (2/2)

- C/C++ function call mechanism is implemented by machine using portion of program memory called ***stack***
  - Stack is program memory used for *inter-function communication*: passing arguments, for storing return value, for saving registers
  - Stack is also used for providing *storage for variables* defined in a function
  - Portion of stack allocated for single function is called *stack frame*

Functions: Pass-by-Value Convention (1/20)

this variable is called *formal parameter* or just *parameter*

```
int myabs(int number) {
  return number < 0 ? -number : number;
}
```

client calls function myabs using function call operator ()

```
int num = 10;
num = myabs(-num)
```

this expression is called *function argument*

1) At runtime, expression (or argument) -num is evaluated
2) Result of evaluation is used to initialize parameter number
3) Changes made to parameter number are localized to function myabs
4) Function myabs terminates by returning value of type int
5) When function myabs terminates, variable number ceases to exist

Functions: Pass-by-Value Convention (2/20)

- Example

```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

- Output

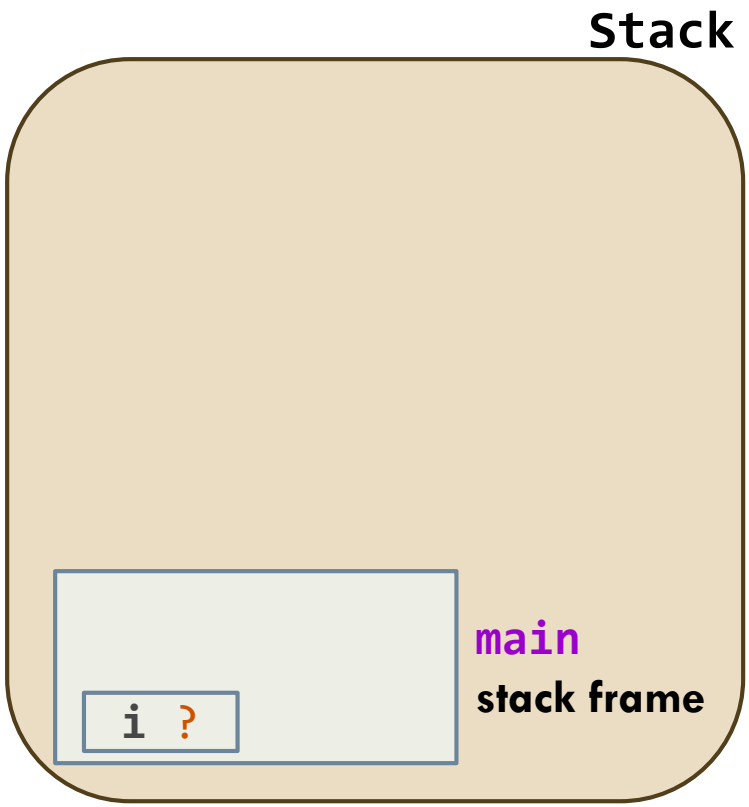
```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
After call: i is 5
```

Functions: Pass-by-Value Convention (3/20)


```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

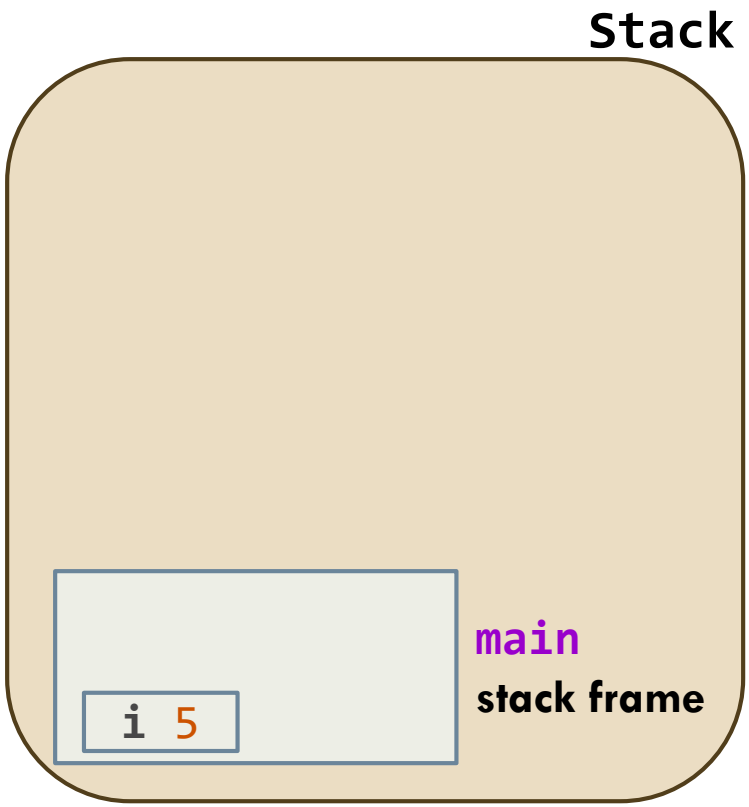


Functions: Pass-by-Value Convention (4/20)


```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

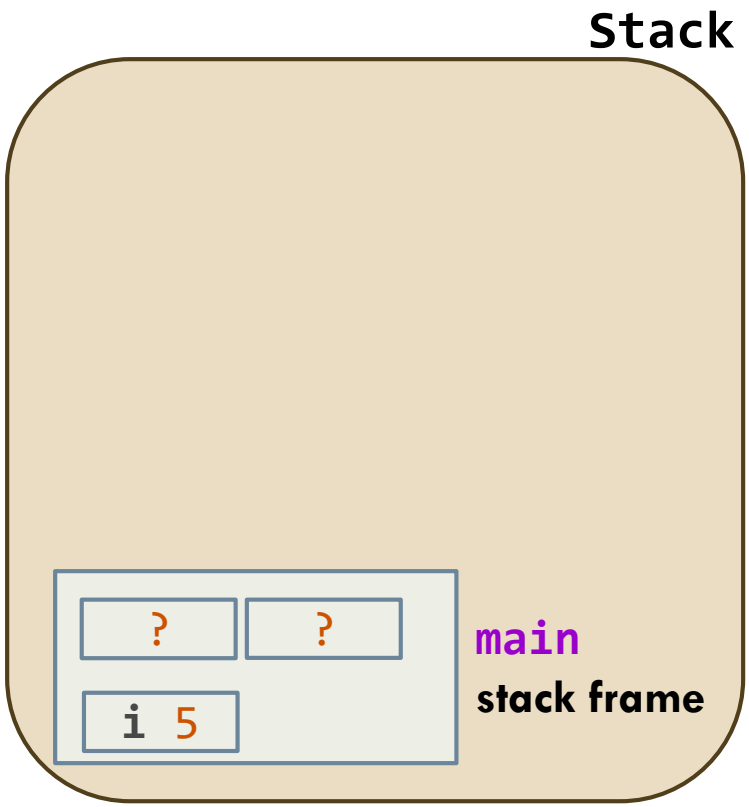


Functions: Pass-by-Value Convention (5/20)


```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

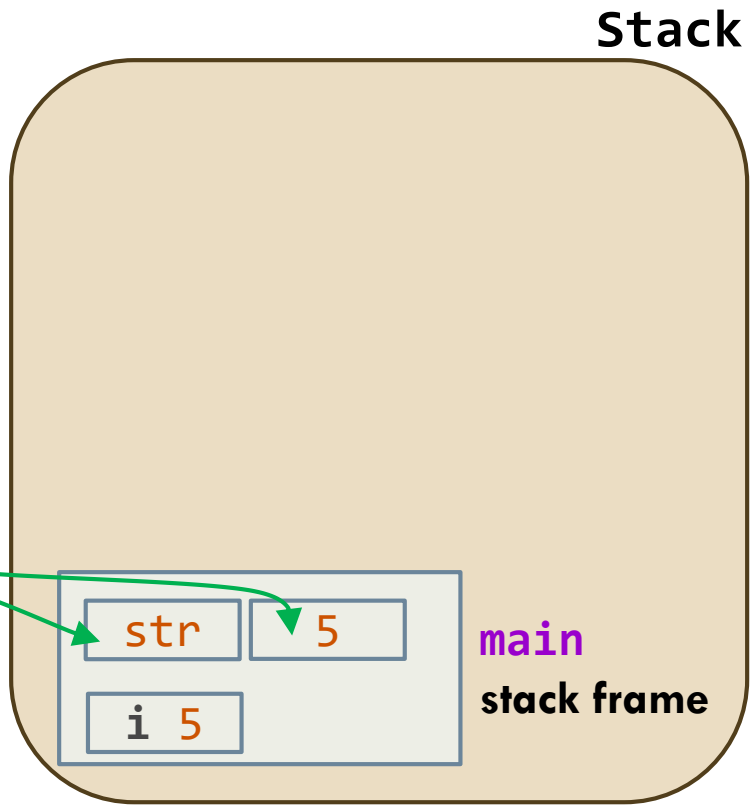


Functions: Pass-by-Value Convention (6/20)


```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



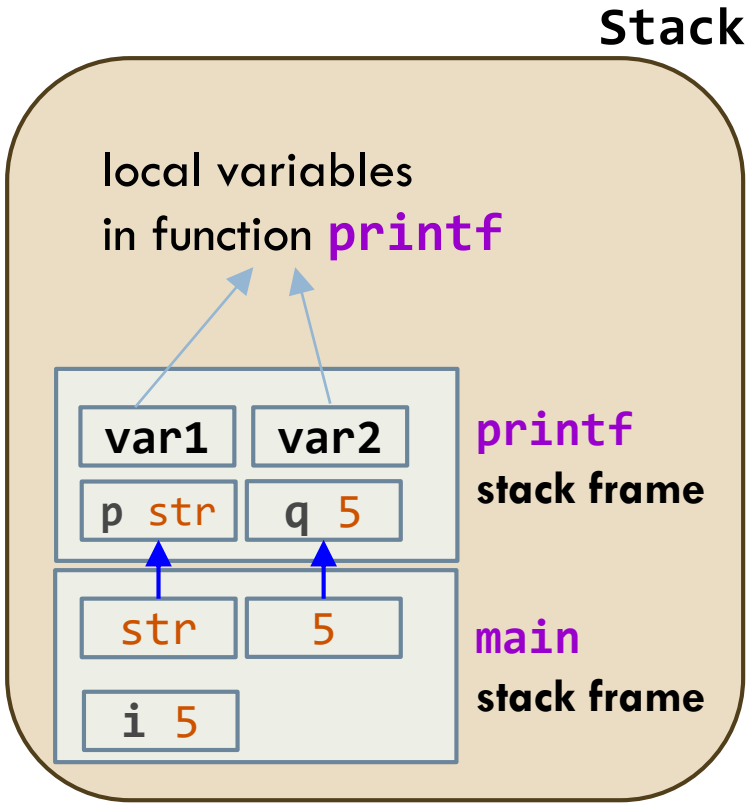
Functions: Pass-by-Value Convention (7/20)

Before call: i is 5

```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



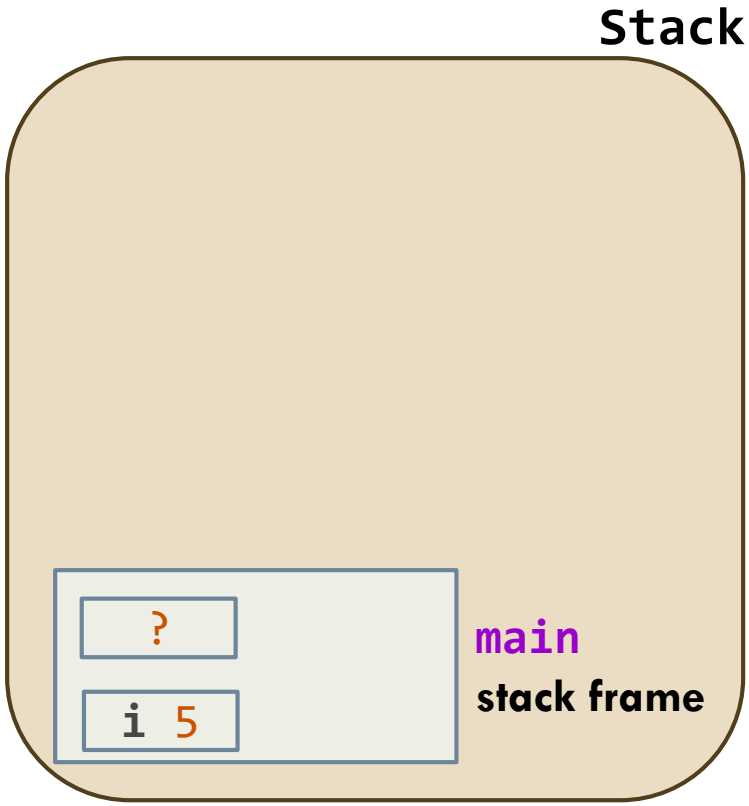
Functions: Pass-by-Value Convention (8/20)

Before call: i is 5

```
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



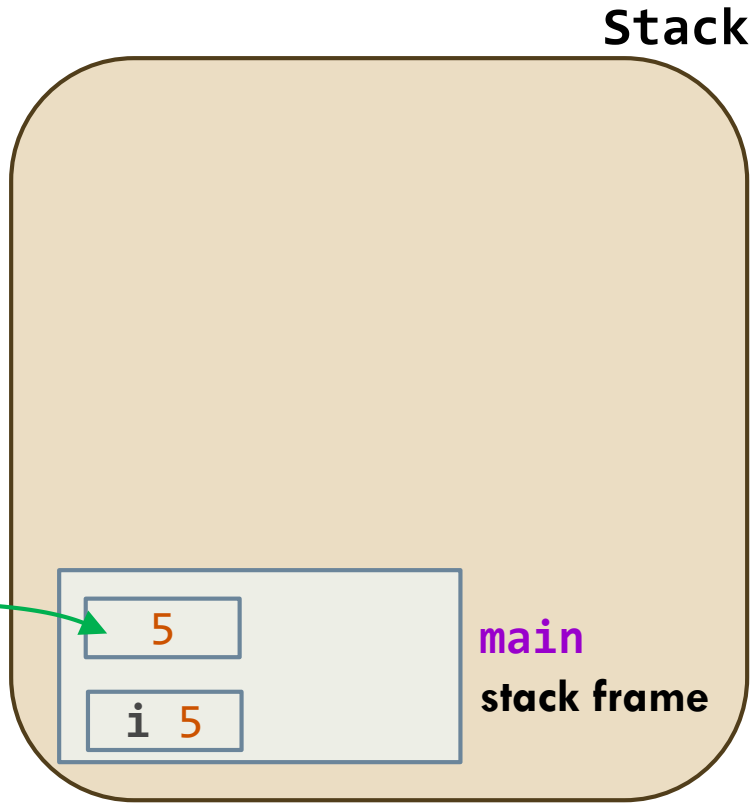
Functions: Pass-by-Value Convention (9/20)

Before call: i is 5

```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



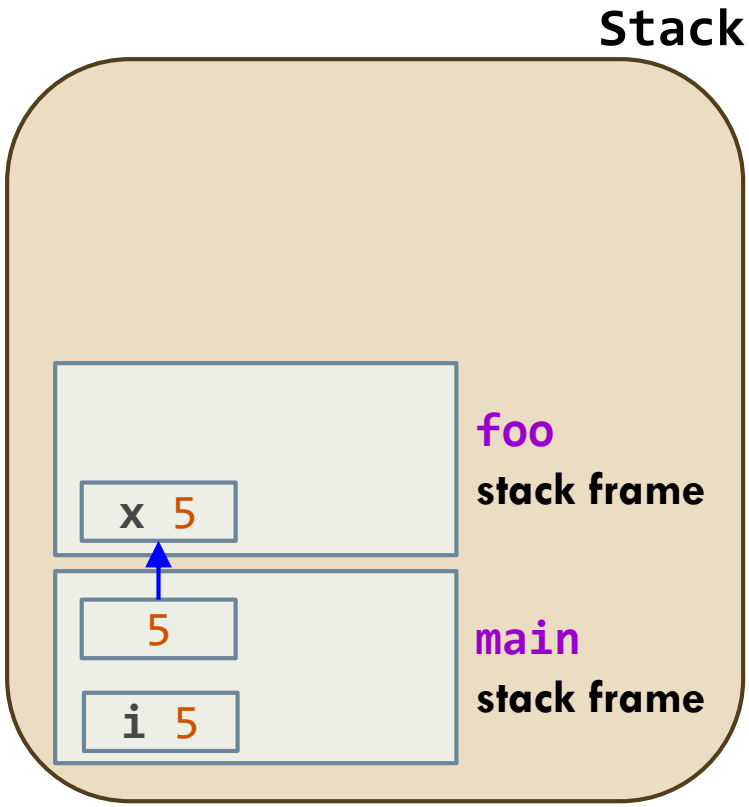
Functions: Pass-by-Value Convention (10/20)

Before call: i is 5

```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



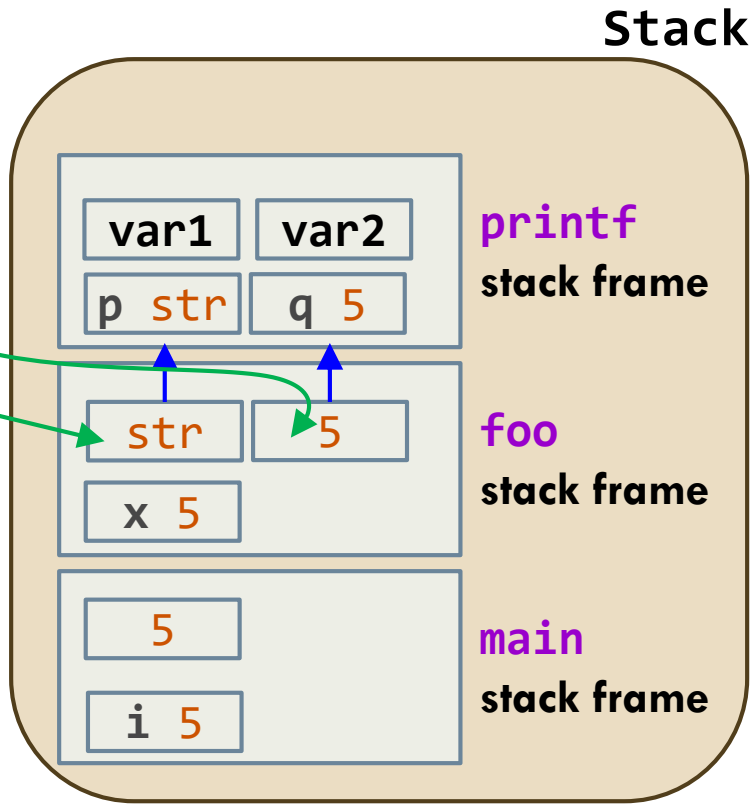
Functions: Pass-by-Value Convention (11/20)


```
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

Before call: i is 5
In foo, x is 5



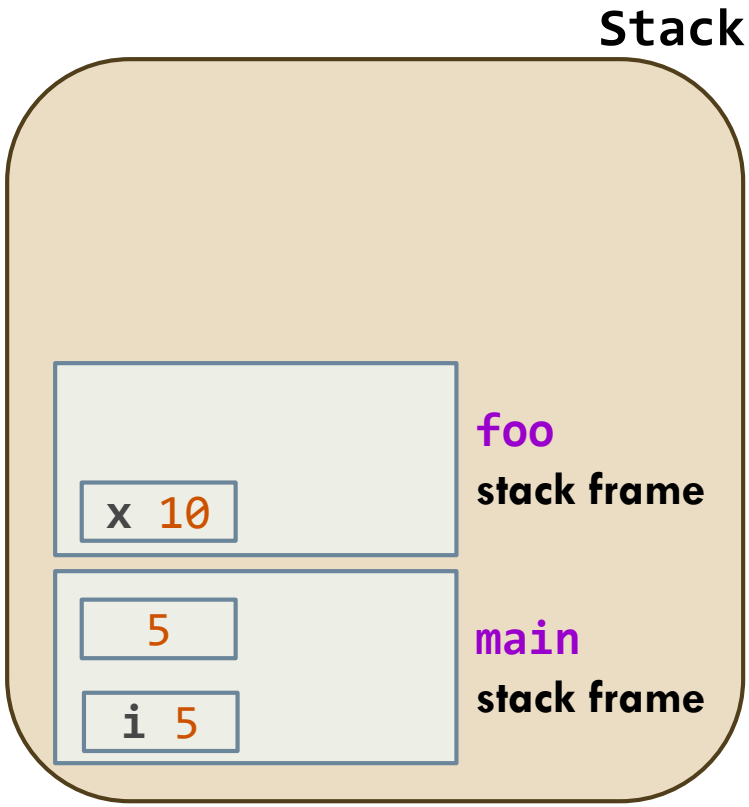
Functions: Pass-by-Value Convention (12/20)

Before call: i is 5
In foo, x is 5

```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```



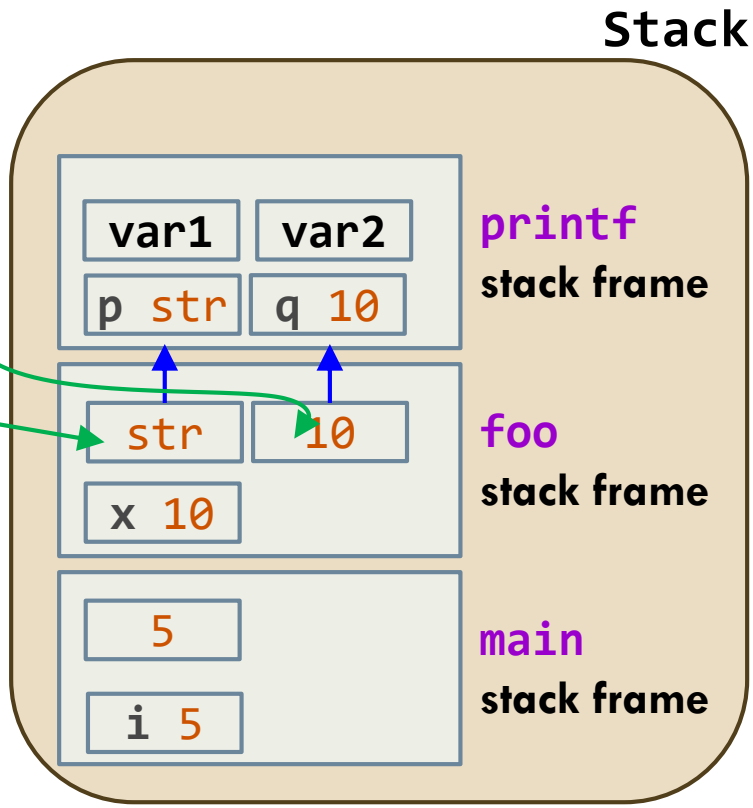
Functions: Pass-by-Value Convention (13/20)


```
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
```



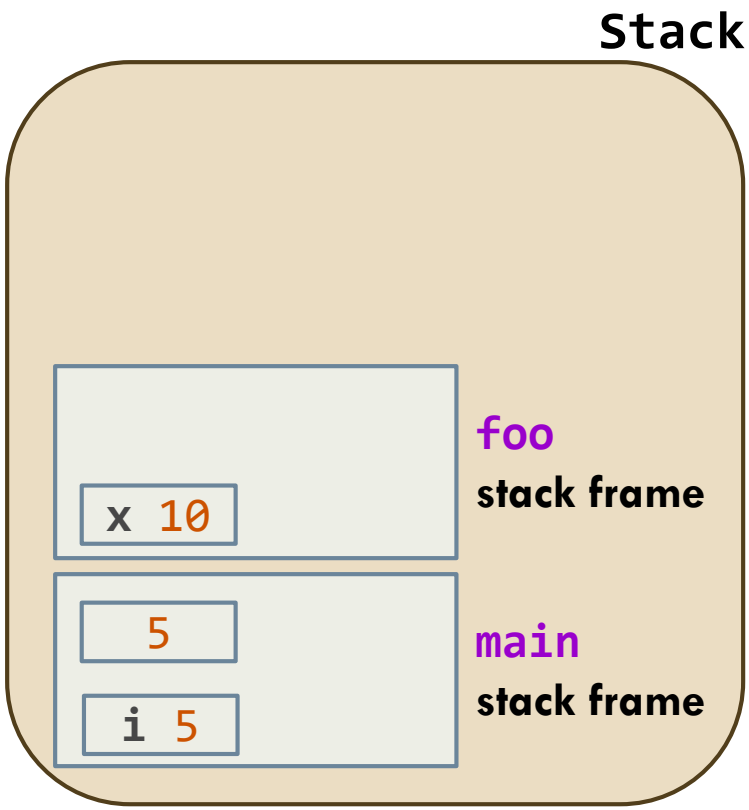
Functions: Pass-by-Value Convention (14/20)


```
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
```



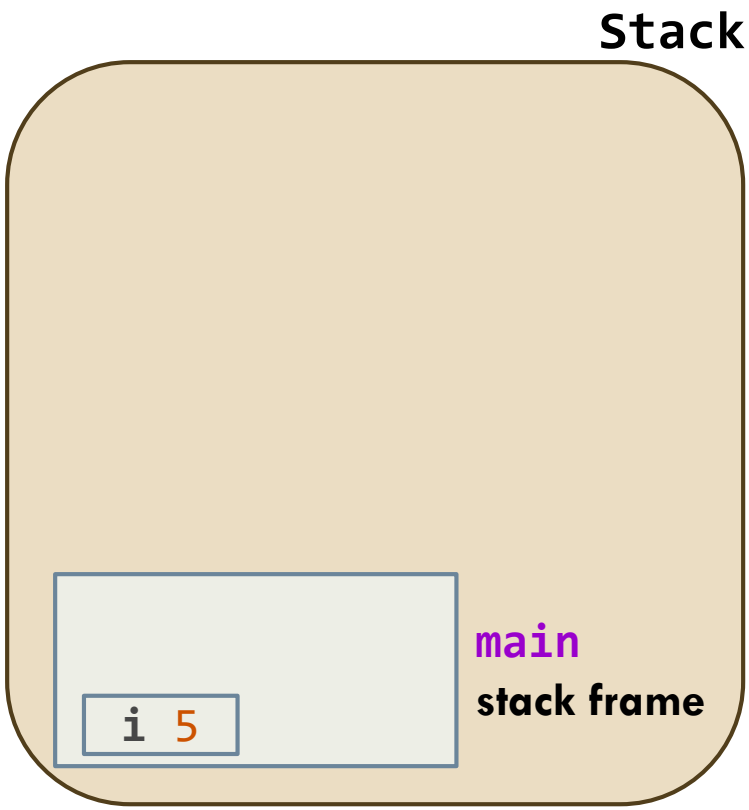
Functions: Pass-by-Value Convention (15/20)


```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

Before call: i is 5
In foo, x is 5
In foo, x is now 10



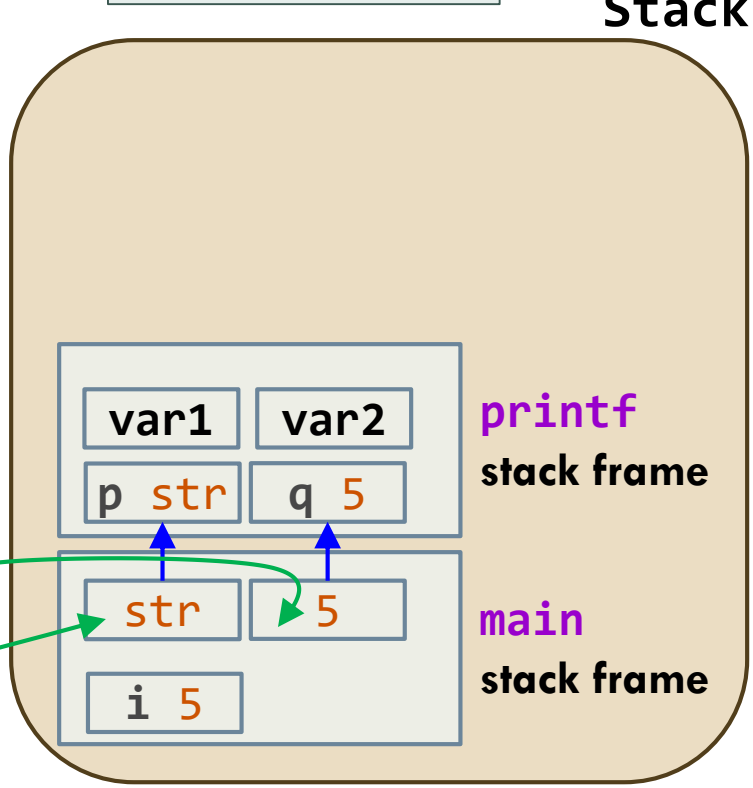
Functions: Pass-by-Value Convention (16/20)


```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
After call: i is 5
```



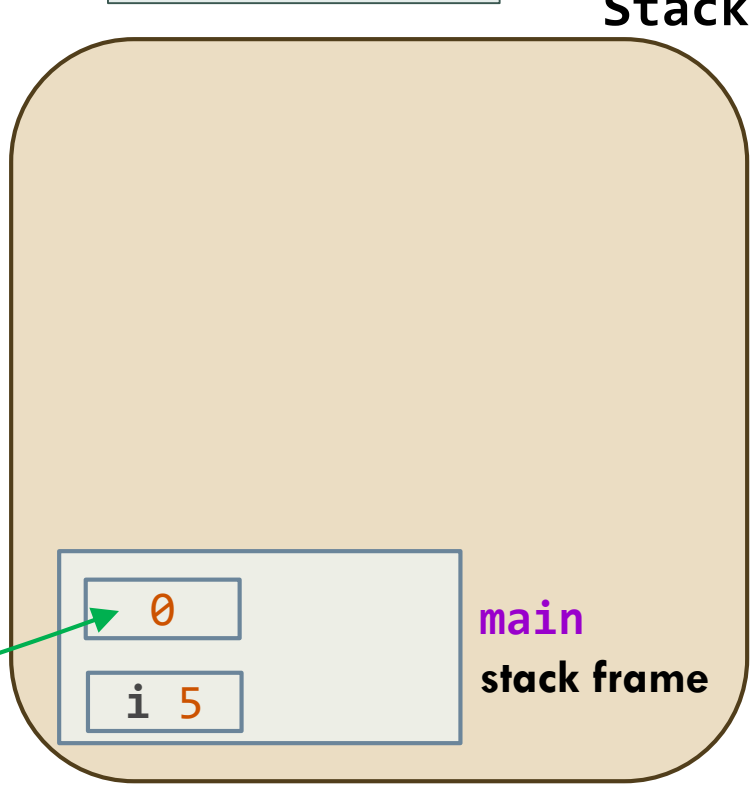
Functions: Pass-by-Value Convention (17/20)


```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
After call: i is 5
```



Functions: Pass-by-Value Convention (18/20)

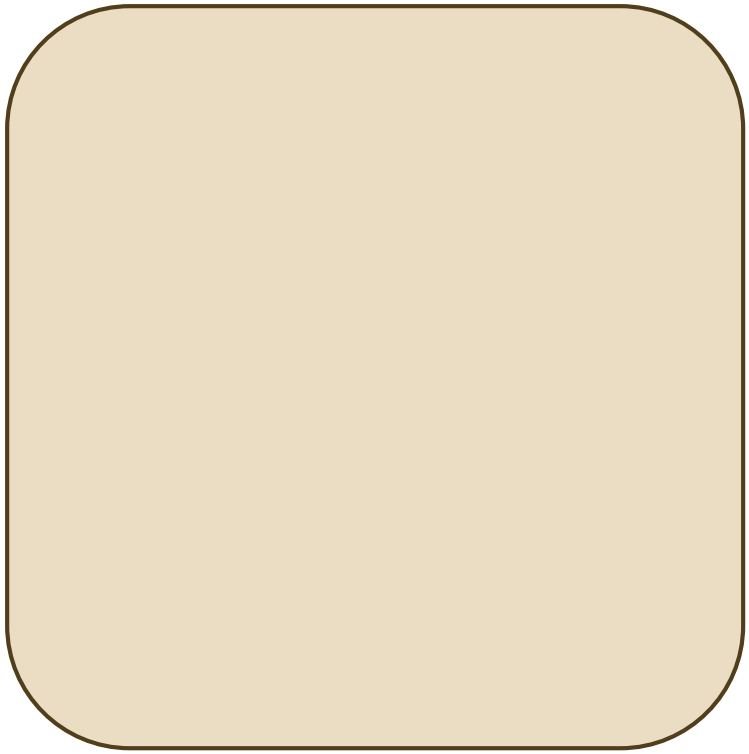

```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

Before call: i is 5
In foo, x is 5
In foo, x is now 10
After call: i is 5

Stack



Functions: Pass-by-Value Convention (19/20)


```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

```
Before call: i is 5
In foo, x is 5
In foo, x is now 10
After call: i is 5
```

Main takeaway:

Inter-function communication uses *pass-by-value* semantics. Using the *stack*, copy of argument `i` is passed to function `foo` to initialize parameter `x`.

Changes made to parameter `x` do not affect argument `i`!!!

Functions: Pass-by-Value Convention (20/20)

- Visualization of program

```c
#include <stdio.h>

void foo(int x) {
  printf("In foo, x is %d\n", x);
  x = 10;
  printf("In foo, x is now %d\n", x);
}

int main(void) {
  int i;
  i = 5;
  printf("Before call: i is %d\n", i);
  foo(i); // call to function foo
  printf("After call: i is %d\n", i);
  return 0;
}
```

Pass-by-Value Convention: Example

- Specify the ordered sequence of function calls made by this program. Write - in order - the functions that are called (including functions `main` and `printf`) and the arguments associated with each of these calls.

Pass-by-Value Convention: Example [Answers]

Function called	Argument 1	Argument 2
main	void	-
printf	"main's local variable x is originally: %d\n"	1
boo	2	-
printf	"boo's local variable x is originally: %d\n"	2
coo	4	-
printf	"coo's local variable x is originally: %d\n"	4
doo	7	-
printf	"doo's local variable x is originally: %d\n"	7
printf	"doo's local variable x is now : %d\n"	11
printf	"coo's local variable x is now : %d\n"	15
printf	"boo's local variable x is now : %d\n"	18
printf	"main's local variable x is now : %d\n"	20

Summary

- Function is encapsulation of algorithm
- Function prototype/declaration
- Function definition
- Function call operator
- Function arguments and parameters
- Call by value semantics
- Role of stack in implementation of call by value semantics
- Tracing functions and identifying arguments