Computer Science I

Functions & Pointers

Dr. Chris Bourke
cbourke@cse.unl.edu

Outline

Part I: Introduction

Functions

- A function produces an *output* when given an *input* or *inputs*

Functions

- A function produces an *output* when given an *input* or *inputs*
- In math:

$$f(x) = x^2$$

$$g(x, y) = 2x + 3y$$

Functions

- A function produces an *output* when given an *input* or *inputs*
- In math:

$$f(x) = x^2$$

$$g(x, y) = 2x + 3y$$

- "Outputs":

$$f(3) = 9$$

$$g(2, 4) = 16$$

Functions

- A function produces an *output* when given an *input* or *inputs*
- In math:

$$f(x) = x^2$$

$$g(x, y) = 2x + 3y$$

- "Outputs":

$$f(3) = 9$$

$$g(2, 4) = 16$$

- It can only ever produce at most *one* output

Functions

- A function produces an *output* when given an *input* or *inputs*
- In math:

$$f(x) = x^2$$

$$g(x, y) = 2x + 3y$$

- "Outputs":

$$f(3) = 9$$

$$g(2, 4) = 16$$

- It can only ever produce at most *one* output
- It may take any number of inputs (including none!)

Functions

Introduction
Declaring
Modularity
Pitfalls & Other Issues
How Functions Work
Pointers
Pass By Reference

- A function produces an *output* when given an *input* or *inputs*
- In math:

$$f(x) = x^2$$

$$g(x, y) = 2x + 3y$$

- "Outputs":

$$f(3) = 9$$

$$g(2, 4) = 16$$

- It can only ever produce at most *one* output
- It may take any number of inputs (including none!)
- Functions in code are similar and use familiar "syntax"

Functions in Code

- In code, a *function* is a reusable unit of code that may take input(s) and may produce an output

Functions in Code

- In code, a *function* is a reusable unit of code that may take input(s) and may produce an output
- Familiar functions: `main()`, `printf()`, `sqrt()`

Functions in Code


```
int main(int argc, char **argv) {

  double x = 2.0;
  double y = sqrt(x);

  return 0;
}
```

Functions in Code


```
int main(int argc, char **argv) {

  double x = 2.0;
  double y = sqrt(x);

  return 0;
}
```

- You *call* or *invoke* a function

Functions in Code


```
int main(int argc, char **argv) {

  double x = 2.0;
  double y = sqrt(x);

  return 0;
}
```

- You *call* or *invoke* a function
- Functions can be called inside other functions

Functions in Code


```c
int main(int argc, char **argv) {

  double x = 2.0;
  double y = sqrt(x);

  return 0;
}
```

- You *call* or *invoke* a function
- Functions can be called inside other functions
- Function A ("calling function") calls function B

Functions in Code


```c
int main(int argc, char **argv) {

  double x = 2.0;
  double y = sqrt(x);

  return 0;
}
```

- You *call* or *invoke* a function
- Functions can be called inside other functions
- Function A ("calling function") calls function B
- A function may "return" a value to the calling function

Motivation

- Functions facilitate *code reuse*

Motivation

- Functions facilitate *code reuse*
- Procedural abstraction: designing and using functions allows you to abstract the details of how a block of code or algorithm works

Motivation

- Functions facilitate *code reuse*
- Procedural abstraction: designing and using functions allows you to abstract the details of how a block of code or algorithm works
- Functions *encapsulate* functionality into reusable, abstract code blocks

Motivation

- Functions facilitate *code reuse*
- Procedural abstraction: designing and using functions allows you to abstract the details of how a block of code or algorithm works
- Functions *encapsulate* functionality into reusable, abstract code blocks
- Example: how does `sqrt()` work?

Usefulness

- Standard libraries and external libraries are full of useful functions

Usefulness

- Standard libraries and external libraries are full of useful functions
- Well tested, well designed, efficient and optimized

Usefulness

- Standard libraries and external libraries are full of useful functions
- Well tested, well designed, efficient and optimized
- Functions are fundamental to top-down design

Usefulness

- Standard libraries and external libraries are full of useful functions
- Well tested, well designed, efficient and optimized
- Functions are fundamental to top-down design
- Problem solving: first question you ask is “is this problem already solved?”

Functions in C

- Functions must be declared before they can be used

Functions in C

- Functions must be declared before they can be used
- Declare a function using a *prototype* that specifies the function's *signature*

Functions in C

- Functions must be declared before they can be used
- Declare a function using a *prototype* that specifies the function's *signature*
- Signature:

Functions in C

- Functions must be declared before they can be used
- Declare a function using a *prototype* that specifies the function's *signature*
- Signature:
  - Name of the function (*identifier*)

Functions in C

- Functions must be declared before they can be used
- Declare a function using a *prototype* that specifies the function's *signature*
- Signature:
  - Name of the function (*identifier*)
  - A list of its *parameters* or *arguments*; both the *number* and *type*

Functions in C

- Functions must be declared before they can be used
- Declare a function using a *prototype* that specifies the function's *signature*
- Signature:
  - Name of the function (*identifier*)
  - A list of its *parameters* or *arguments*; both the *number* and *type*
  - The function's *return* type: the type of data the function returns

Prototype Example

```
/**
 *  This function computes the Euclidean distance between
 *  the two points defined by (x1, y1) and (x2, y2)
 */
double euclideanDistance(double x1, double y1, double x2, double y2);
```

Syntax and style:

- Documented using doc-style comments (DRY)

Prototype Example

```
/**
 *  This function computes the Euclidean distance between
 *  the two points defined by (x1, y1) and (x2, y2)
 */
double euclideanDistance(double x1, double y1, double x2, double y2);
```

Syntax and style:

- Documented using doc-style comments (DRY)
- Return type

Introduction
Declaring
Modularity
Pitfalls & Other Issues
How Functions Work
Pointers
Pass By Reference

Prototype Example

```
/**
 *  This function computes the Euclidean distance between
 *  the two points defined by (x1, y1) and (x2, y2)
 */
double euclideanDistance(double x1, double y1, double x2, double y2);
```

Syntax and style:

- Documented using doc-style comments (DRY)
- Return type
- Identifier: use `lowerCamelCasing`

Prototype Example


```
/**
 *  This function computes the Euclidean distance between
 *  the two points defined by (x1, y1) and (x2, y2)
 */
double euclideanDistance(double x1, double y1, double x2, double y2);
```

Syntax and style:

- Documented using doc-style comments (DRY)
- Return type
- Identifier: use `lowerCamelCasing`
- Comma delimited list of parameters and their type

Prototype Example


```
/**
 *  This function computes the Euclidean distance between
 *  the two points defined by (x1, y1) and (x2, y2)
 */
double euclideanDistance(double x1, double y1, double x2, double y2);
```

Syntax and style:

- Documented using doc-style comments (DRY)
- Return type
- Identifier: use `lowerCamelCasing`
- Comma delimited list of parameters and their type
- Prototype ends with a semicolon and has no function *body*

Definition Example

```
double euclideanDistance(double x1, double y1, double x2, double y2) {

  double temp = sqrt( (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2) );
  return temp;
}
```

- Signature is repeated but a function *body* is included

Definition Example

```
double euclideanDistance(double x1, double y1, double x2, double y2) {

  double temp = sqrt( (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2) );
  return temp;
}
```

- Signature is repeated but a function *body* is included
- The `temp` variable is a *local* variable

Nebraska Lincoln

Definition Example


```
double euclideanDistance(double x1, double y1, double x2, double y2) {

  double temp = sqrt( (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2) );
  return temp;
}
```

- Signature is repeated but a function *body* is included
- The `temp` variable is a *local* variable
- Parameter variables are also *local*

Definition Example

```
double euclideanDistance(double x1, double y1, double x2, double y2) {

  double temp = sqrt( (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2) );
  return temp;
}
```

- Signature is repeated but a function *body* is included
- The `temp` variable is a *local* variable
- Parameter variables are also *local*
- Observe: functions call functions

Definition Example

```
double euclideanDistance(double x1, double y1, double x2, double y2) {

  double temp = sqrt( (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2) );
  return temp;
}
```

- Signature is repeated but a function *body* is included
- The `temp` variable is a *local* variable
- Parameter variables are also *local*
- Observe: functions call functions
- Demonstration

Part II: Modularity

Writing modular code, creating libraries

Modularity

- Modularity refers to the degree to which a system's components can be separated and recombined or reused

Modularity

- Modularity refers to the degree to which a system's components can be separated and recombined or reused
- In software, this means separating functionality (functions) into independent interchangeable modules or "libraries"

Modularity

- Modularity refers to the degree to which a system's components can be separated and recombined or reused
- In software, this means separating functionality (functions) into independent interchangeable modules or "libraries"
- Separation allows you to organize very complex programs with thousands or millions of LOCs

Modularity

- Modularity refers to the degree to which a system's components can be separated and recombined or reused
- In software, this means separating functionality (functions) into independent interchangeable modules or "libraries"
- Separation allows you to organize very complex programs with thousands or millions of LOCs
- Programs only "include" the functionality they actually need

Modularity in C

- In C, libraries are separated out into different files

Modularity in C

- In C, libraries are separated out into different files
- Prototypes are placed in a *header* file (ends with `.h`)

Modularity in C

- In C, libraries are separated out into different files
- Prototypes are placed in a *header* file (ends with `.h`)
- Definitions are placed in a *source* file (ends with `.c`)

Modularity in C

- In C, libraries are separated out into different files
- Prototypes are placed in a *header* file (ends with `.h`)
- Definitions are placed in a *source* file (ends with `.c`)
- `stdio.h`, `math.h`

Modularity in C

- In C, libraries are separated out into different files
- Prototypes are placed in a *header* file (ends with `.h`)
- Definitions are placed in a *source* file (ends with `.c`)
- `stdio.h`, `math.h`
- You can then `#include` your own libraries!

Demonstration

- Separate our distance functionality into:

Demonstration

- Separate our distance functionality into:
  - Header: `distance.h`

Demonstration

- Separate our distance functionality into:
  - Header: `distance.h`
  - Source: `distance.c`

Demonstration

- Separate our distance functionality into:
  - Header: `distance.h`
  - Source: `distance.c`
  - Use the same base file name

Demonstration

- Separate our distance functionality into:
  - Header: `distance.h`
  - Source: `distance.c`
  - Use the same base file name
- Include the header file in our main: for *user defined libraries* we generally use `#include "distance.h"`

Demonstration

- Separate our distance functionality into:
  - Header: `distance.h`
  - Source: `distance.c`
  - Use the same base file name
- Include the header file in our main: for *user defined libraries* we generally use `#include "distance.h"`
- Compile separate modules:

Demonstration

- Separate our distance functionality into:
  - Header: `distance.h`
  - Source: `distance.c`
  - Use the same base file name
- Include the header file in our main: for *user defined libraries* we generally use `#include "distance.h"`
- Compile separate modules:
  - Compile our distance library:
    `gcc -c distance.c`
    produces an *object* file, `distance.o`

Demonstration

- Separate our distance functionality into:
  - Header: `distance.h`
  - Source: `distance.c`
  - Use the same base file name
- Include the header file in our main: for *user defined libraries* we generally use
  `#include "distance.h"`
- Compile separate modules:
  - Compile our distance library:
    `gcc -c distance.c`
    produces an *object* file, `distance.o`
  - Compile the entire program together including the library files:
    `gcc distance.o distanceDriver.c`

Demonstration

- Separate our distance functionality into:
  - Header: `distance.h`
  - Source: `distance.c`
  - Use the same base file name
- Include the header file in our main: for *user defined libraries* we generally use `#include "distance.h"`
- Compile separate modules:
  - Compile our distance library:
    `gcc -c distance.c`
    produces an *object* file, `distance.o`
  - Compile the entire program together including the library files:
    `gcc distance.o distanceDriver.c`
- Add additional distance-related functionality

Part III: Pitfalls & Other Issues

Void functions

- Functions are not required to return a value

Void functions

- Functions are not required to return a value
- Functions that do not return a value are called `void` functions

Void functions

- Functions are not required to return a value
- Functions that do not return a value are called `void` functions
- Keyword `void` is used as the return type

Void functions

- Functions are not required to return a value
- Functions that do not return a value are called `void` functions
- Keyword `void` is used as the return type
- Example: `void swap(int a, int b);`

Void functions

- Functions are not required to return a value
- Functions that do not return a value are called `void` functions
- Keyword `void` is used as the return type
- Example: `void swap(int a, int b);`
- Return statement should still be included: `return;`

Void functions

- Functions are not required to return a value
- Functions that do not return a value are called `void` functions
- Keyword `void` is used as the return type
- Example: `void swap(int a, int b);`
- Return statement should still be included: `return;`
- Functions are not required to take inputs

Void functions

- Functions are not required to return a value
- Functions that do not return a value are called `void` functions
- Keyword `void` is used as the return type
- Example: `void swap(int a, int b);`
- Return statement should still be included: `return;`
- Functions are not required to take inputs
- Functions that do not have any inputs are no-arg functions

Void functions

- Functions are not required to return a value
- Functions that do not return a value are called `void` functions
- Keyword `void` is used as the return type
- Example: `void swap(int a, int b);`
- Return statement should still be included: `return;`
- Functions are not required to take inputs
- Functions that do not have any inputs are no-arg functions
- Example: `int foo(void);`

Void functions

- Functions are not required to return a value
- Functions that do not return a value are called `void` functions
- Keyword `void` is used as the return type
- Example: `void swap(int a, int b);`
- Return statement should still be included: `return;`
- Functions are not required to take inputs
- Functions that do not have any inputs are no-arg functions
- Example: `int foo(void);`
- Better practice to omit `void`:
  `int foo();`

- Some languages support *Function Overloading*

Function Overloading

- Some languages support *Function Overloading*
- *Function Overloading*: multiple functions may share the same identifier (name) as long as they differ in the number or type of parameters

Function Overloading

- Some languages support *Function Overloading*
- *Function Overloading*: multiple functions may share the same identifier (name) as long as they differ in the number or type of parameters
- C does *not* support function overloading

Function Overloading

- Some languages support *Function Overloading*
- *Function Overloading*: multiple functions may share the same identifier (name) as long as they differ in the number or type of parameters
- C does *not* support function overloading
- Consequence: functions that do the same thing but with different types all need unique names

Function Overloading

- Some languages support *Function Overloading*
- *Function Overloading*: multiple functions may share the same identifier (name) as long as they differ in the number or type of parameters
- C does *not* support function overloading
- Consequence: functions that do the same thing but with different types all need unique names
- Example: `abs(int)`, `fabs(double)`, `labs(long)`

Function Overloading

- Some languages support *Function Overloading*
- *Function Overloading*: multiple functions may share the same identifier (name) as long as they differ in the number or type of parameters
- C does *not* support function overloading
- Consequence: functions that do the same thing but with different types all need unique names
- Example: `abs(int)`, `fabs(double)`, `labs(long)`
- Must take care when naming functions so as not to *pollute the namespace*

Missing Return Statements

- Common mistake: forgetting to include the `return` statement

Missing Return Statements

- Common mistake: forgetting to include the `return` statement
- Omission is still syntactically correct; but will not give the intended results

Missing Return Statements

- Common mistake: forgetting to include the `return` statement
- Omission is still syntactically correct; but will not give the intended results
- Can be easily avoided using the `-Wall` flag

Missing Return Statements

- Common mistake: forgetting to include the `return` statement
- Omission is still syntactically correct; but will not give the intended results
- Can be easily avoided using the `-Wall` flag
- Demonstration

Common Compilation Failures

- Must use `#include` and you *only* ever include header files

Common Compilation Failures

- Must use `#include` and you *only* ever include header files
- Prototypes should always be included

Common Compilation Failures

- Must use `#include` and you *only* ever include header files
- Prototypes should always be included
- Prototype and definition signatures *must* match

Common Compilation Failures

- Must use `#include` and you *only* ever include header files
- Prototypes should always be included
- Prototype and definition signatures *must* match
- Demonstration

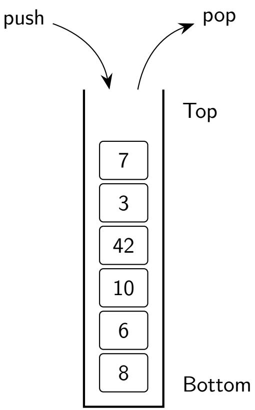
Part IV: How Functions Work

How Functions Work

- Programs have a *program stack*

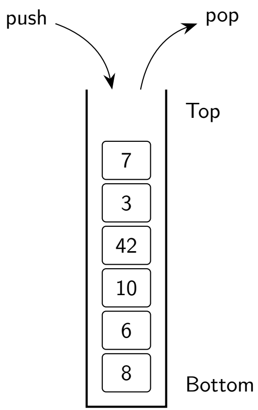
How Functions Work

- Programs have a *program stack*
- Stack: Last-In First-Out (LIFO)



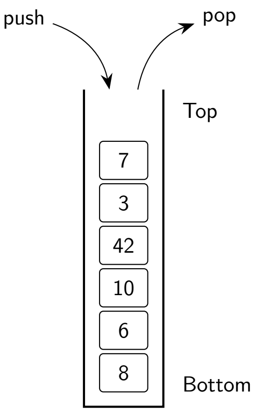
How Functions Work

- Programs have a *program stack*
- Stack: Last-In First-Out (LIFO)
- Push: add something to the top



How Functions Work

- Programs have a *program stack*
- Stack: Last-In First-Out (LIFO)
- Push: add something to the top
- Pop: remove something from the top



Program Stack

Introduction
Modularity
Pitfalls & Other Issues
How Functions Work
Pointers
Pass By Reference

- At the start of a program, a program call stack is created

Nebraska Lincoln

Program Stack

- At the start of a program, a program call stack is created
- At the bottom, the program code is loaded

Program Stack

- At the start of a program, a program call stack is created
- At the bottom, the program code is loaded
- Global variables and static content are added

Program Stack

- At the start of a program, a program call stack is created
- At the bottom, the program code is loaded
- Global variables and static content are added
- As functions are called, a new stack frame is created

Program Stack

- At the start of a program, a program call stack is created
- At the bottom, the program code is loaded
- Global variables and static content are added
- As functions are called, a new stack frame is created
- Each frame contains: parameters, local variables, etc.

Program Stack

- At the start of a program, a program call stack is created
- At the bottom, the program code is loaded
- Global variables and static content are added
- As functions are called, a new stack frame is created
- Each frame contains: parameters, local variables, etc.
- When a function returns, the frame is popped from the top of the call stack

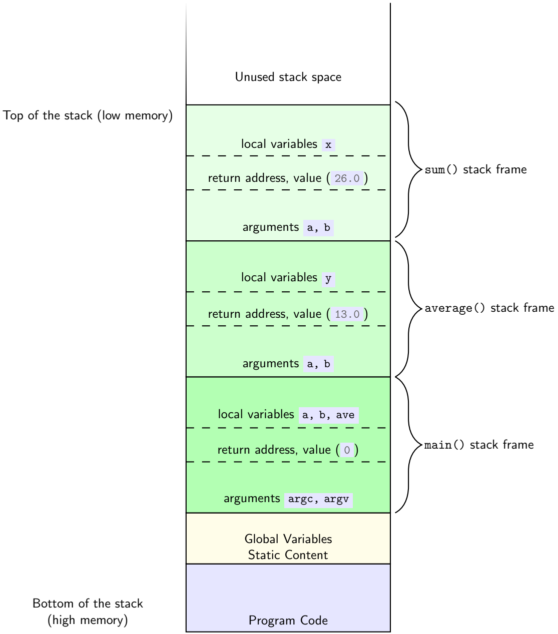
Program Stack

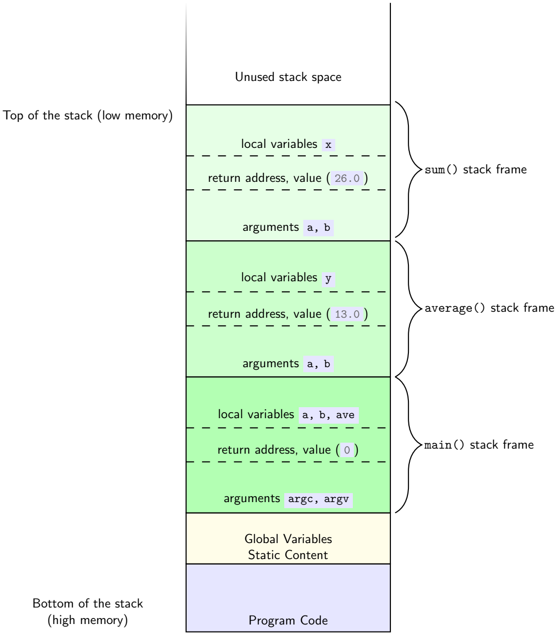
- At the start of a program, a program call stack is created
- At the bottom, the program code is loaded
- Global variables and static content are added
- As functions are called, a new stack frame is created
- Each frame contains: parameters, local variables, etc.
- When a function returns, the frame is popped from the top of the call stack
- Data and variables in each frame is distinct and separate

Program Stack

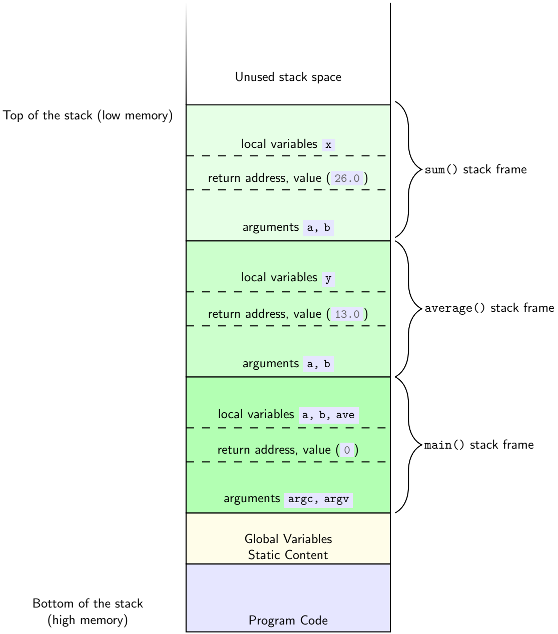

```c
double sum(double a, double b) {
  double x = a + b;
  return x;
}

double average(double a, double b) {
  double y = sum(a, b) / 2.0;
  return y;
}

int main(int argc, char **argv) {
  double a = 10.0, b = 16.0;
  double ave = average(a, b);
  printf("average = %f\n", ave);
  return 0;
}
```

- Stack orientation

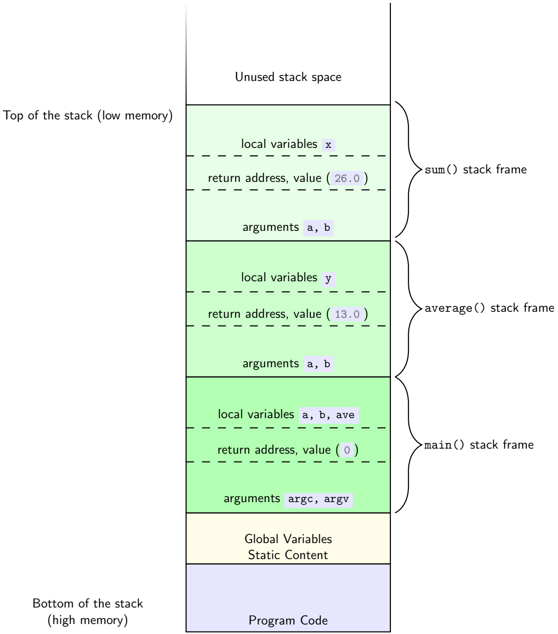




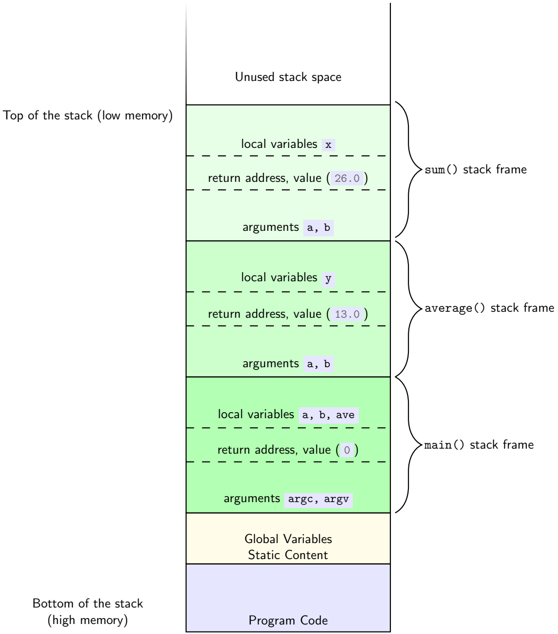
- Stack orientation
- Each function has its own stack frame



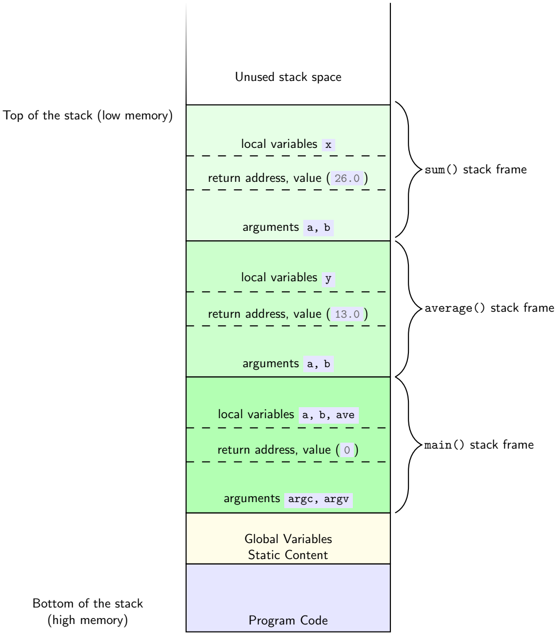
- Stack orientation
- Each function has its own stack frame
- Parameters/arguments are *passed by value* to the function



- Stack orientation
- Each function has its own stack frame
- Parameters/arguments are *passed by value* to the function
- The values stored in the variables are copied into variables in a new stack frame



- Stack orientation
- Each function has its own stack frame
- Parameters/arguments are *passed by value* to the function
- The values stored in the variables are copied into variables in a new stack frame
- As functions return, stack frames are removed



- Stack orientation
- Each function has its own stack frame
- Parameters/arguments are *passed by value* to the function
- The values stored in the variables are copied into variables in a new stack frame
- As functions return, stack frames are removed
- Live demonstration

Demonstration

Consider the following code that attempts to swap two values:

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

Part V: Pointers

Memory

- Every piece of data in a computer is stored in memory

Memory

- Every piece of data in a computer is stored in memory
- Memory has both an *address* and *contents*

Address	Contents
0x7fff58310b87	0x01
0x7fff58310b86	0x32
0x7fff58310b85	0x7c
0x7fff58310b84	0xff
0x7fff58310b83	
0x7fff58310b82	
0x7fff58310b81	
0x7fff58310b80	
0x7fff58310b7f	
0x7fff58310b7e	
0x7fff58310b7d	
0x7fff58310b7c	3.14159265359
0x7fff58310b7b	
0x7fff58310b7a	
0x7fff58310b79	
0x7fff58310b78	32,321,231
0x7fff58310b77	
0x7fff58310b76	
0x7fff58310b75	
0x7fff58310b74	1,458,321
.	.

Pointers

- In C, you can access memory *contents* with variables

Pointers

- In C, you can access memory *contents* with variables
- You can access memory *addresses* with *pointers*

Pointers

- In C, you can access memory *contents* with variables
- You can access memory *addresses* with *pointers*
- A pointer is a memory reference that "points" to some memory address

Pointers

- In C, you can access memory *contents* with variables
- You can access memory *addresses* with *pointers*
- A pointer is a memory reference that "points" to some memory address
- Syntax:

```c
//regular variable:
int a = 10;

//an integer pointer variable:
int *ptrToA;
```

Pointers


```
int *ptrToA;
```

- `ptrToA` is a pointer variable that can point to a memory location that stores an integer

Pointers


```
int *ptrToA;
```

- `ptrToA` is a pointer variable that can point to a memory location that stores an integer
- An *uninitialized pointer* may point to anything:

```
int *ptrToA;
```

- `ptrToA` is a pointer variable that can point to a memory location that stores an integer
- An *uninitialized pointer* may point to anything:
  - Non-existant memory location

Pointers


```
int *ptrToA;
```

- `ptrToA` is a pointer variable that can point to a memory location that stores an integer
- An *uninitialized pointer* may point to anything:
  - Non-existant memory location
  - Memory location that doesn't belong to us

Pointers


```
int *ptrToA;
```

- `ptrToA` is a pointer variable that can point to a memory location that stores an integer
- An *uninitialized pointer* may point to anything:
  - Non-existant memory location
  - Memory location that doesn't belong to us
  - `0xDEADBEEF`

```
int *ptrToA;
```

- `ptrToA` is a pointer variable that can point to a memory location that stores an integer
- An *uninitialized pointer* may point to anything:
  - Non-existant memory location
  - Memory location that doesn't belong to us
  - `0xDEADBEEF`
- Best practice to initialize pointers

Pointers


```
int *ptrToA;
```

- `ptrToA` is a pointer variable that can point to a memory location that stores an integer
- An *uninitialized pointer* may point to anything:
  - Non-existant memory location
  - Memory location that doesn't belong to us
  - `0xDEADBEEF`
- Best practice to initialize pointers
- The `NULL` pointer is a special pointer that indicates "nothing"

Pointers


```
int *ptrToA;
```

- `ptrToA` is a pointer variable that can point to a memory location that stores an integer
- An *uninitialized pointer* may point to anything:
  - Non-existant memory location
  - Memory location that doesn't belong to us
  - `0xDEADBEEF`
- Best practice to initialize pointers
- The `NULL` pointer is a special pointer that indicates "nothing"
- Initialization:

  ```
  int *ptrToA = NULL;
  ```

Introduction
Modularity
Pitfalls & Other Issues
How Functions Work
Pointers
Pass By Reference

- Pointer variables can be created for any type of variable.

```
int *ptrToA = NULL;
double *ptrToB = NULL;
char *ptrToC = NULL;
```

Pointers
Other Types

- Pointer variables can be created for any type of variable.

```
int *ptrToA = NULL;
double *ptrToB = NULL;
char *ptrToC = NULL;
```

- You can also test for nullity:

```
if(ptrToA == NULL) {
  printf("Error: ptrA points to nothing!\n");
}
```

- Pointer variables can be created for any type of variable.

```
int *ptrToA = NULL;
double *ptrToB = NULL;
char *ptrToC = NULL;
```

- You can also test for nullity:

```
if(ptrToA == NULL) {
  printf("Error: ptrA points to nothing!\n");
}
```

- Null pointer check

- Need to be able to make a pointer *point* to something

Using Pointers
Referencing Operator

- Need to be able to make a pointer *point* to something
- Usual assignment operator works, but *both sides must match*

Using Pointers
Referencing Operator

- Need to be able to make a pointer *point* to something
- Usual assignment operator works, but *both sides must match*
- Referencing operator, `&` gives the *memory address* of a regular variable

Using Pointers

Referencing Operator

- Need to be able to make a pointer *point* to something
- Usual assignment operator works, but *both sides must match*
- Referencing operator, `&` gives the *memory address* of a regular variable

```c
int a = 42;
int *ptrToA = NULL;

//make ptrToA point to a:
ptrToA = &a;
```

Using Pointers

Referencing Operator

- Need to be able to make a pointer *point* to something
- Usual assignment operator works, but *both sides must match*
- Referencing operator, `&` gives the *memory address* of a regular variable

```
int a = 42;
int *ptrToA = NULL;

//make ptrToA point to a:
ptrToA = &a;
```

```
double b = 10.5;
double *ptrToB = NULL;
ptrToB = &b;
```

Pitfalls

- Types of pointer variables and what they point to *need to match*:

```
int a = 42;
double *dblPtr = NULL;
dblPtr = &a; //WRONG
```

Pitfalls

- Types of pointer variables and what they point to *need to match*:

```c
int a = 42;
double *dblPtr = NULL;
dblPtr = &a; //WRONG
```

- Pointers *must be assigned a valid memory address*

```c
int a = 42;
int *ptrToA = NULL;
ptrToA = a;  //WRONG!!
ptrToA = 10; //SO WRONG!!
```

Using Pointers
Dereferencing Operator

- Once a pointer points to something, we need a way to get to the memory *contents*

- Once a pointer points to something, we need a way to get to the memory *contents*
- The *dereferencing operator*, `*` "changes" a pointer into a regular variable

- Once a pointer points to something, we need a way to get to the memory *contents*
- The *dereferencing operator*, `*` "changes" a pointer into a regular variable

```
int a = 42;
int *ptrToA = &a;

int b = *ptrToA + 10;
```

Using Pointers

Dereferencing Operator

- Once a pointer points to something, we need a way to get to the memory *contents*
- The *dereferencing operator*, `*` "changes" a pointer into a regular variable

```c
int a = 42;
int *ptrToA = &a;

int b = *ptrToA + 10;
```

- You can also *change* the contents

```c
*ptrToA = 43;
```

- Pointer declaration:

```
int *ptr = NULL;
```

Summary

- Pointer declaration:

```
int *ptr = NULL;
```

- Pointer initialization and referencing operator:

```
ptr = &a;
```

Summary

- Pointer declaration:
  ```
  int *ptr = NULL;
  ```
- Pointer initialization and referencing operator:
  ```
  ptr = &a;
  ```
- Dereferencing operator: `*ptr = 43;`

Summary

- Pointer declaration:
  ```
  int *ptr = NULL;
  ```
- Pointer initialization and referencing operator:
  ```
  ptr = &a;
  ```
- Dereferencing operator: `*ptr = 43;`
- Demonstration

Summary

- Pointer declaration:
  ```
  int *ptr = NULL;
  ```
- Pointer initialization and referencing operator:
  ```
  ptr = &a;
  ```
- Dereferencing operator: `*ptr = 43;`
- Demonstration
- Why pointers?

Summary

- Pointer declaration:
  `int *ptr = NULL;`
- Pointer initialization and referencing operator:
  `ptr = &a;`
- Dereferencing operator: `*ptr = 43;`
- Demonstration
- Why pointers?
- So we can *pass variables by reference*!!

Part VI: Pass By Reference

Pass By Reference

- Recall our swap function:

```c
void swap(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
  return;
}
```

Pass By Reference

- Recall our swap function:

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

- Failed because `a` and `b` were passed by value

Pass By Reference

- Recall our swap function:

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

- Failed because `a` and `b` were passed by value
- Values were copied to separate local parameter variables

Pass By Reference

- Recall our swap function:

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

- Failed because `a` and `b` were passed by value
- Values were copied to separate local parameter variables
- Using pointers, we can pass *references* to the variables instead

Pass By Reference

- Parameters become pointer variables instead:

Pass By Reference

- Parameters become pointer variables instead:

```c
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
  return;
}
```

Pass By Reference

- Parameters become pointer variables instead:

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

- You've actually seen this before!

Pass By Reference

- Parameters become pointer variables instead:

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

- You've actually seen this before!
- `scanf("%d", &x);`

Pass By Reference

- Parameters become pointer variables instead:

```c
void swap(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
  return;
}
```

- You've actually seen this before!
- `scanf("%d", &x);`
- Demonstration

- Passing by reference means that *pointers* to variables are passed instead of copies

- Passing by reference means that *pointers* to variables are passed instead of copies
- Functions can make changes that are *visible* to the calling function

Pass By Reference

Summary

Introduction
Modularity
Pitfalls & Other Issues
How Functions Work
Pointers
Pass By Reference

- Passing by reference means that *pointers* to variables are passed instead of copies
- Functions can make changes that are *visible* to the calling function
- Consequences:

- Passing by reference means that *pointers* to variables are passed instead of copies
- Functions can make changes that are *visible* to the calling function
- Consequences:
    - Functions can have *side effects*

- Passing by reference means that *pointers* to variables are passed instead of copies
- Functions can make changes that are *visible* to the calling function
- Consequences:
  - Functions can have *side effects*
  - Functions can "return" multiple values via pass-by-reference variables

Introduction
Modularity
Pitfalls & Other Issues
How Functions Work
Pointers
Pass By Reference

Demonstration:

1. Modify one of our distance functions to utilize pass-by-reference
2. Implement a new function to calculate a line graph:

$$y = mx + b$$

Given $(x_1, y_1), (x_2, y_2)$, compute:

$$m = \frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

$$b = y_1 - mx_1$$