Section 3: Memory and Flow Control

3.1: Control Flow

Control Flow

We use ***control flow*** to model how programs are executed. During execution, we keep track of the ***program location***, which is "*where*" in the code the execution is currently occurring. When a program is "run", the *program location* starts at the beginning of the main function.

There are four types of control flow:

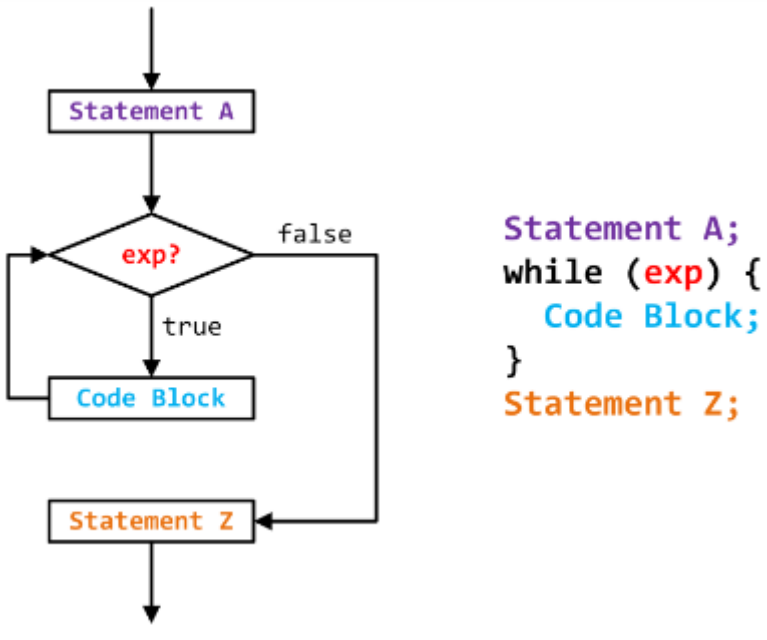
- compound statements(blocks)
- function calls
- conditionals (*i.e.*, if statements)
- iteration (*i.e.*, loops)

while

The syntax for a `while` loop is as follows:

```
while (expression) statement
```

Notice that `while` is similar to `if`: the statement is only executed if the expression is true. The difference is, while **repeatedly** "*loops back*" and executes the statement **until the expression is false.** Like with `if`, `while` always uses a block (`{}`) for a *compound statement*, even if there is only a single statement.



Example:

```
int main(void) {
  int i = 2;
  while (i >= 0) {
    printf("%d\n", i);
    --i;
  }
}
```

Iteration vs. recursion

Using a loop to solve a problem is called `iteration`. *Iteration* is an alternative to *recursion* and is much more common in imperative programming.

The main difference between iteration and recursion is that in in recursion, we use function calls to execute the statements repeatedly inside of the function body. In iteration, on the other hand, we use loops like while to do the same.

Iteration is often about the same time efficiency as recursion. However, sometimes the iterative approach is easier to code and/or understand for a particular problem. The same is true of recursion, which will become clearer when we cover in order tree traversal later in the course.

Example:

```
// recursive_sum(k) finds the sum of the numbers 0...k using an simple recursion
int recursive_sum(int k) {
  if (k <= 0) {
    return 0;
  }
  return k + recursive_sum(k - 1);
```

```
}

// iterative_sum(k) finds the sum of the numbers 0...k
int iterative_sum(int k) {
  int s = 0;
  while (k > 0) {
    s += k;
    --k;
  }
  return s;
}

int main(void) {
  trace_int(recursive_sum(5));
  trace_int(iterative_sum(5));
}
```

Loops can be "nested" within each other.

Example:

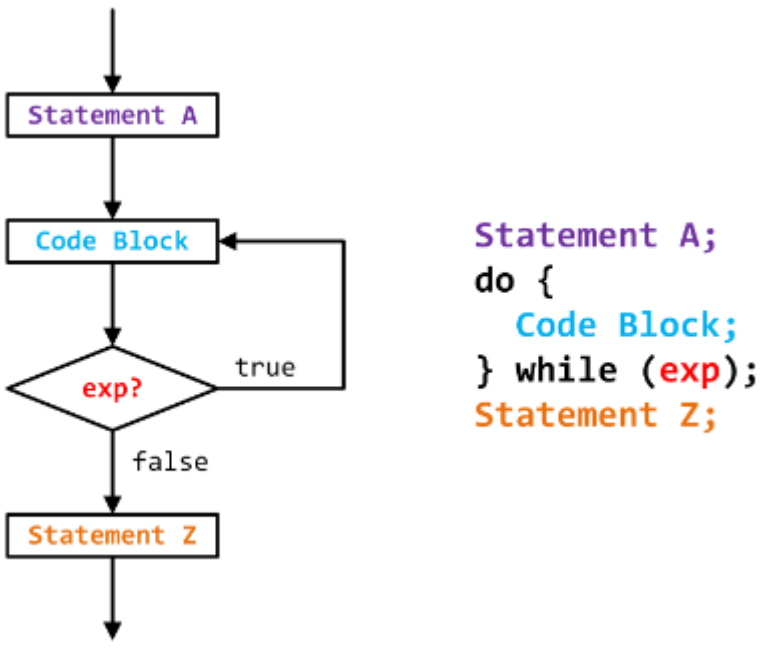
```
int main(void) {
  int i = 5;
  while (i >= 0) {
    int j = i;
    while (j >= 0) {
      printf("*");
      --j;
    }
    printf("CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
    --i;
  }
}
```

do ... while

The syntax for the `do...while` control flow statement is:

```
do statement while (expression);
```

The difference between the while loop and the do...while loop is that in a do...while loop the statement is always executed *at least* once, and the expression is checked at the *end* of the loop instead of the beginning.



Example:

```
int main(void) {
  int i = 0;

  printf("Do..while loop:CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
  // this do...while example
  i = 5;                      // setup
  do {
    printf("%dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", i);        // statement
    --i;                      // update
  } while (i >= 0);           // expression

  printf("while loop:CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
```

```
  // this while example
  i = 5;                        // setup
  while (i >= 0) {              // expression
    printf("%dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", i);         // statement
    --i;                        // update
  }
}
```

break and continue

The `break` control flow statement is useful to exit from the *middle* of a loop. The `break` statement will immediately terminate the current (innermost) loop. As such, the `break` statement is often used with a (purposefully) infinite loop.

Example:

```
int main(void) {

  int n = 0;

  while (1) {
  n = read_int();
  if (n == READ_INT_FAIL) {
    break;
  }
  printf("%dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", n);
  }
}
```

The `continue` control flow statement skips over the rest of the statements in the current block (`{}`) and "continues" with loop.

Example:

```
// odd(n) returns true if a number is odd and false otherwise
bool odd(n) {
  return (n % 2 != 0);
}

int main(void) {

  int n = 0;

  while (1) {
  n = read_int();
  if (n == READ_INT_FAIL) {
    break;
  }
  if (!odd(n)) {
    continue;
  }
  printf("%dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", n);
  }
}
```

for loops

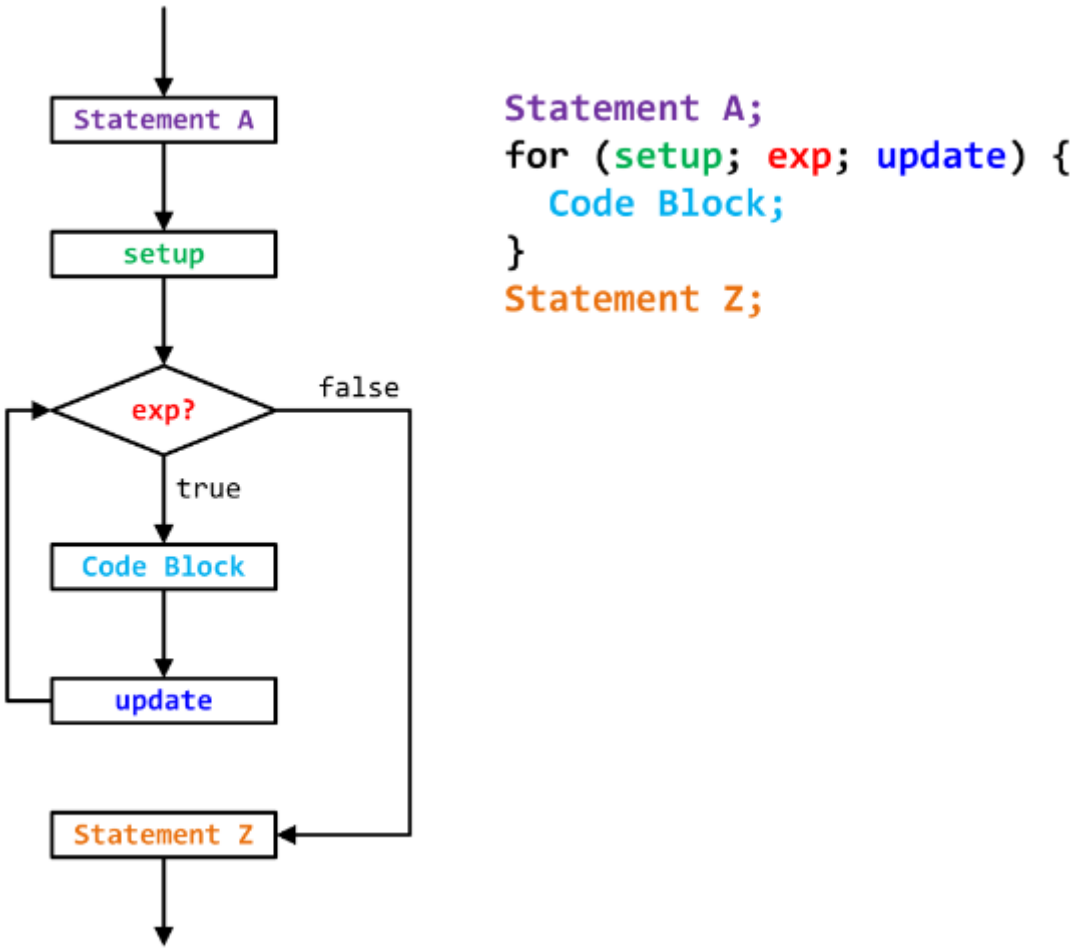
The final control flow statement is `for`, which is often referred to as a "for loop". `for` loops are a "condensed" version of a `while` loop.

The format of a while loop is often of the form:

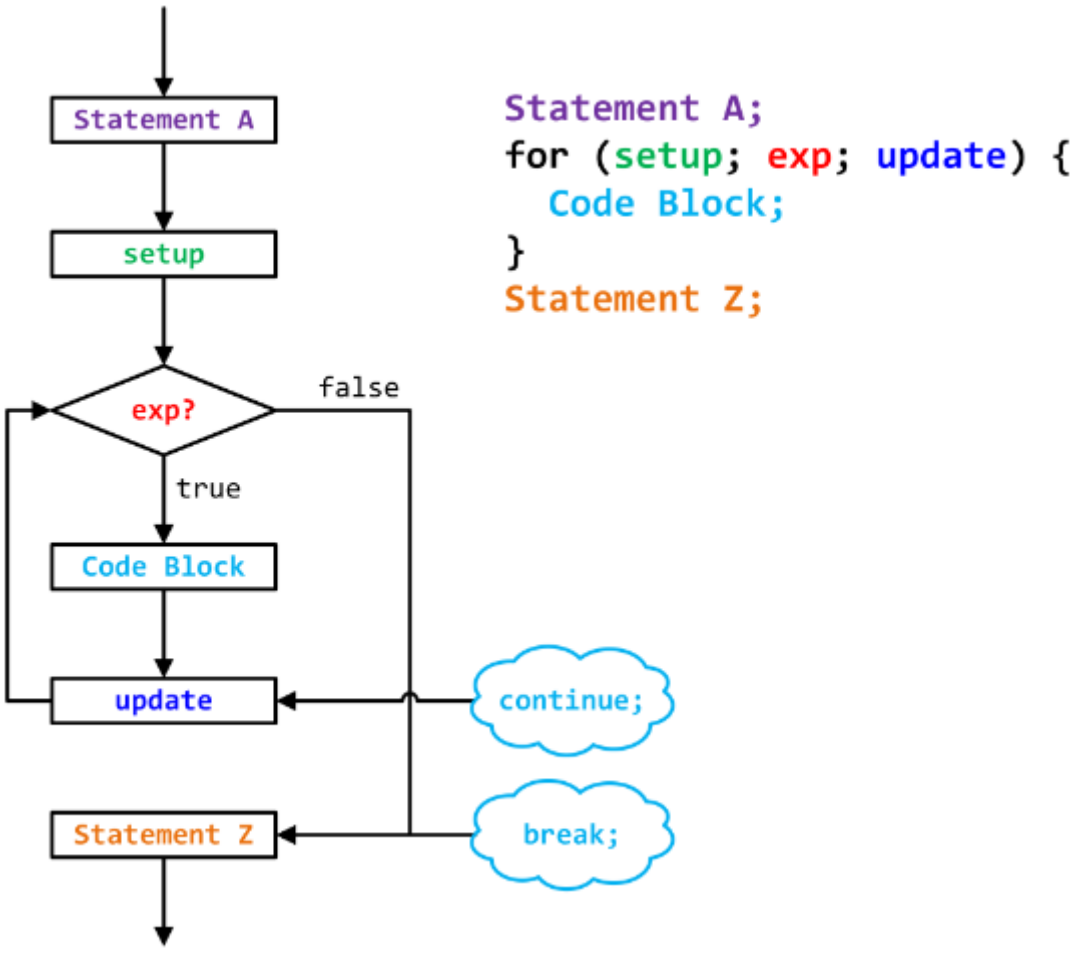
```
while (expression) {
 body statement(s)
 update statement
}
```

which can be rewritten as a single for loop:

```
for (setup; expression; update) {
   body statement(s)
}
```



`break` and `continue` statements can be used in `for` loops just as in `while` loops:



3.2: Memory and Variables

Memory

One bit of storage (in memory) has two possible **states**: 0 or 1. A byte is 8 bits of storage. Each byte in memory is in one of 256 possible states. The smallest accessible unit of memory is a byte. To access a byte of memory, its *position* in memory, which is known as the **address** of the byte, must be known.

Memory addresses are usually represented in *hex*, so with 1MB of memory, the address of the first byte is 0x0, and the address of the last byte is 0xFFFFF.

You can visualize computer memory as a collection of "labeled mailboxes" where each mailbox stores a byte. The *contents* in the below table are arbitrary values.

Address (1 MB of Storage)	Contents (one byte per address)
0x00000	00101001
0x00001	1101101
...	...

0xFFFFE	00010111
0xFFFFF	01110011

Defining Variables

For a **variable definition**, C

- reserves (or "finds") space in memory to `store` the variable.
- "keeps track of" the *address* of that storage location
- stores the initial value of the variable at that location (address).

sizeof

When we define a variable, C reserves space in memory to store its value —— but **how much space** is required? It depends on the **type** of the variable.

The **size operator** (`sizeof`) produces the number of bytes required to store a type (it can also be used on identifiers). `sizeof` looks like a function, but it is an operator.

The size of an integer is 4 bytes (32 bits).

Integer limits

Because C uses 4 bytes (32 bits) to store an int, there are only 2^{32} (4,294,967,296) possible values that can be represented. The range of C int values is -2^{31} -2^{31} ... (2^{31} - 1) or -2,147,483,648 ... 2,147,483,647. In our CS 136 environment, the constants `INT_MIN` and `INT_MAX` are defined with those limit values.

Overflow

If we try to represent values outside of the int limits, ***overflow*** occurs. Never assume what the value of an int will be after an overflow occurs. The value of an integer that has overflowed is **undefined.**

Example:

```
int main(void) {

  int bil = 1000000000;
  int four_bil = bil + bil + bil + bil;
  int nine_bil = 9 * bil;

  trace_int(INT_MIN);
  trace_int(INT_MAX);
  trace_int(bil);
  trace_int(four_bil);
  trace_int(nine_bil);
}
```

The char type

Now that we have a better understanding of what an int in C is, we introduce some additional types. The char type is also used to store integers, but C only allocates one byte of storage for a char (an int uses 4 bytes).

Because of this limited range, chars are rarely used for calculations. As the name implies, they are often used to store **characters.**

Early in computing, there was a need to represent text (*characters*) in memory. The American Standard Code for Information Interchange (ASCII) was developed to assign a numeric code to each character.

The only control character we use in this course is the line feed (10), which is the newline `CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS` character.

```
/*
  32 space  48 0     64 @     80 P      96 `     112 p
  33 !      49 1     65 A     81 Q      97 a     113 q
  34 "      50 2     66 B     82 R      98 b     114 r
  35 #      51 3     67 C     83 S      99 c     115 s
  36 $      52 4     68 D     84 T     100 d     116 t
  37 %      53 5     69 E     85 U     101 e     117 u
  38 &      54 6     70 F     86 V     102 f     118 v
  39 '      55 7     71 G     87 W     103 g     119 w
  40 (      56 8     72 H     88 X     104 h     120 x
  41 )      57 9     73 I     89 Y     105 i     121 y
  42 *      58 :     74 J     90 Z     106 j     122 z
  43 +      59 ;     75 K     91 [     107 k     123 {
  44 ,      60 <     76 L     92 \     108 l     124 |
  45 -      61 =     77 M     93 ]     109 m     125 }
  46 .      62 >     78 N     94 ^     110 n     126 ~
  47 /      63 ?     79 O     95 _     111 o
*/
```

In C, **single** quotes (') are used to indicate an ASCII character.

Note that the `printf` format specifier to display a *character* is "`%c`".

Example:

```
int main(void) {

  char letter_a = 'a';
  char ninety_seven = 97;

  printf("letter_a as a character:   %cCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", letter_a);
  printf("ninety_seven as a char:    %cCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", ninety_seven);

  printf("letter_a in decimal:       %dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", letter_a);
  printf("ninety_seven in decimal:   %dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", ninety_seven);

}

/*Code Output
letter_a as a character:   a
ninety_seven as a char:    a
letter_a in decimal:       97
ninety_seven in decimal:   97*/
```

Because C interprets characters as integers, characters can be used in expressions to avoid having "magic numbers" in your code.

Symbol

C ***symbols*** are constants (often `ints`) with meaningful identifiers ("names") but arbitrary (meaningless) values. We use them to avoid using magic numbers. As with other constants, you want to use **ALL_CAPS** for symbol names.

Example:

```
const int UP = 1;
const int DOWN = 2;

int direction = UP;
```

Floating Point Types

The C float (floating point) type can represent real (non-integer) values.

C also has a double type that is still inexact but has significantly better precision. Just as we use **check-within** with inexact numbers in Racket, we can use a similar technique for testing floating point numbers in c.

A double has more precision than a float because it uses more memory.

3.3: Memory - Structures

Structures

Structures (*compound data*) in C are similar to structures in Racket. The syntax is:

- the keyword struct is before the name of the structure
- the entire structure is enclosed with curly braces
- there is a semi-colon at the very end of the structure.

C is statically typed, structure definitions require the *type* of each field.

When defining a variable of the structure *type*, include the keyword "`struct`" followed by the structure name.

Instead of *selector functions*, C has a ***structure operator*** (`.`) which "selects" the requested field. The syntax is `variablename.fieldname`.

Example:

```
struct posn {       // name of the structure
  int x;            // type and field names
  int y;
};                  // don't forget this ;


int main(void) {
```

```
  struct posn p = {1, 2};

  trace_int(p.x);
  trace_int(p.y);
}
```

Mutation with structures

The assignment operator can be used with structs to copy all of the fields from another `struct`. Individual fields can also be mutated.

The braces (`{}`) are **part of the initialization syntax** and cannot simply be used in assignment.

The *equality* operator (`==`) **does not work with structures**. You have to define your own equality function.

Example:

```
struct posn {       // name of the structure
  int x;            // type and field names
  int y;
};                  // don't forget this ;

// posn_equal(a, b) returns true the structs a and b are equal and
//   false otherwise
bool posn_equal (struct posn a, struct posn b) {
  return (a.x == b.x) && (a.y == b.y);
}

int main(void) {
  struct posn p = {1, 2};

  p.x = 5;                    // VALID MUTATION
  p.y = 6;                    // VALID MUTATION

  // alternatively:
  struct posn new_p ={5, 6};
  p = new_p;                 // VALID MUTATION

  trace_int(p.x);
  trace_int(p.y);

  if (posn_equal (p, new_p)) {
    printf("The structures p and new_p are equal!CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
  }

  printf("The value of p is (%d, %d)CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", p.x, p.y);
}
```

Structures in the memory model

For a structure *definition*, no memory is reserved. Memory is only reserved when a `struct` **variable** is defined.

The amount of space reserved for a `struct` is **at least** the sum of the `sizeof` for each field, but it may be larger.

You **must** use the `sizeof` operator to determine the size of a structure.

Example

```
struct mystruct1 {
  int x;          // 4 bytes
  char c;         // 1 byte
  int y;          // 4 bytes
};

struct mystruct2 {
  char c;         // 1 byte
  char d;         // 1 byte
  int i;          // 4 bytes
};

int main(void) {
  trace_int(sizeof(struct mystruct1));
  trace_int(sizeof(struct mystruct2));
}
```

3.4: Memory - Sections

Sections of memory

In this course we model five sections (or "regions") of memory:

Code
Read-Only Data
Global Data
Heap
Stack

When evaluating C expressions, the intermediate results must be *temporarily* stored. In this course, we are not concerned with this "temporary" storage.

When you program, you write **source code** in a text editor using ASCII characters that are "human readable". To "run" a C program, the *source code* must first be converted into **machine code** that is "machine readable". This machine code is then placed into the **code section** of memory where it can be executed.

Earlier we described how C "reserves space" in memory for a variable definition.

Global variables are available throughout the entire execution of the program, and the space for the global variables is reserved **before** the program begins execution.

- First, the code from the entire program is scanned and all global variables are identified.
- Next, space for each global variable is reserved.
- Finally, the memory is properly initialized.
- This happens **before the `main` function is called**.

All global variables are placed in either the **read-only data** section (**constants**) or the **global data** section (**mutable variables**).

Example:

```c
// global

int gmv = 13;
int n = 0;

// read-only
const int c = 42;
```

Memory Diagram:

Global	
gmv	13
n	0

Read-Only	
c	42

3.5: Memory - Stacks and Recursion in C

Stacks

A **stack** in computer science is similar to a physical stack where items are “stacked” on top of each other. For example, a stack of papers or a stack of plates. Only the *top* item on a stack is “visible” or “accessible”. Items are *pushed* onto the top of the stack and *popped* off of the stack.

Whenever a function is called, we can imagine that it is *pushed* onto a stack, and it is now on the *top* of the stack. If another function is called, it is then *pushed* so it is now on *top*. The call stack illustrates the "history" or "sequence" of function calls that led us to the current function. When a function returns, it is *popped* off of the stack, and the control flow returns to the function now on *top*.

Example:

```
void blue(void) {
// SEE EXAMPLE 3.5.2: CALL STACK 2
  return;
}

void green(void) {
  return;
}

void red(void) {
  green();
  blue();
}

int main(void) {
  red();
}
```

Call Stack (when blue is called)

blue
red
main

green does not appear because it was previously popped before blue was called

When C encounters a return, it needs to know: "where was the program location **before** this function was called?" In other words, it needs to "remember" the program location to "jump back to" when returning. This location is known as the ***return address***.

In the example when the function foo is called, the program location is on line 9 of the function main so we would record **the return address** as:

```
// foo() is just a demo function
void foo(void) {
  printf("inside fooCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
  return;
}

int main(void) {
  printf("inside mainCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
  foo();
  printf("back from fooCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
}
```

Stack frames

The "entries" pushed onto the *call stack* are known as `stack frames`. Each function call creates a *stack* frame (or a "*frame of reference*"). Each *stack frame* contains:

- the **argument values**
- all **local variables** (both mutable variables and constants) that appear within the function *block* (including any sub-blocks)
- the **return address** (the program location within the *calling* function to *return* to)

Example:

```
int pow4(int j) {
    printf("inside pow4CODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
    int k = j * j;
    // Snapshot
    return k * k;
}
```

```c
int main(void) {
    printf("inside mainCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS");
    int i = 1;
    printf("%dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", pow4(i + i));
}
```

```
=========================
pow4:
  j: 2
  k: 4
  return address: main:11
=========================
main:
  i: 1
  return address: OS
=========================
```

Example:

```c
struct foo {
  int x;
  int y;
};

int baz(struct foo qux) {
  return qux.x + 1;
}

int main(void) {
  struct foo bar = {5, 6};
  // Snapshot
  int x = baz(bar);
  return 0;
}
```

```
=========================
main:
  bar:
    .x = 5
    .y = 6
  x: ???
  return address: OS
=========================
```

C **makes a copy** of each argument value and **places the copy in the stack frame**. This is known as the “pass by value” convention.

Whereas space for a *global* variable is reserved *before* the program begins execution, space for a *local* variable is only reserved **when the function is called**. The space is reserved within the newly created stack frame. When the function returns, the variable (and the entire frame) is popped and effectively “disappears”.

We can now model all of the **control flow** when a function is called:

- a *stack frame* is created (“pushed” onto the Stack)
- the current program location is placed in the stack frame as the *return* address
- a **copy** of each of the arguments is placed in the stack frame
- the program location is changed to the start of the new function
- the initial values of local variables are set when their definition is encountered

When a function `returns`:

- the current program location is changed back to the *return address* (which is retrieved from the stack frame)
- the stack frame is removed (“popped” from the Stack memory area)

The return address is a **location within the calling function** and has **nothing** to do with the **location of** any `return` statement(s) in the called function or if one does not exist (e.g., a void function).

There is always one (and only one) return address in a stack frame.

Recursion

In C, each recursive call is simply a new *stack frame* with a separate frame of reference. The only unusual aspect of recursion is that the *return* address is a location within the same function.

Example:

```c
int sum_first(int n) {
  if(n == 0) {   // Snapshot here
    return 0;
  } else {
    return n + sum_first(n - 1);
  }
}

int main(void) {
  int a = sum_first(2);
  //...
}
```

```
==============================
sum_first:
  n: 0
  return address: sum_first:5
==============================
sum_first:
  n: 1
  return address: sum_first:5
==============================
sum_first:
  n: 2
  return address: main:10
==============================
main:
  a: ???
  return address: OS
==============================
```

Stack section

The *call stack* is stored in the ***stack section***, the fourth section of our memory model. We refer to this section as "the stack".

In practice, the "bottom" of the stack (*i.e.*, where the `main` stack frame is placed) is placed at the *highest* available memory address. Each additional stack frame is then placed at increasingly *lower* addresses. The stack "grows" toward lower addresses.

If the stack grows too large, it can "collide" with other sections of memory. This is called "*stack overflow*" and can occur with very deep (or infinite) recursion.

Uninitialized memory

C allows variable definitions without any initialization.

For all **global** variables, C automatically initializes the variable to be zero. Regardless, it is good style to explicitly initialize a global variable to be zero, even if it is automatically initialized.

A **local** variable (on the *stack*) that is uninitialized, has an **arbitrary** initial value.

Example:

```c
int i;  // uninitialized global variable

// weird() is just a demo function
void weird(void) {
  int j = 8675309;
  printf("weird: the value of j is: %dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", j);
}

// mystery() is just a demo function
void mystery(void) {
  int k;  // uninitialized local variable
  printf("mystery: the value of k is: %dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", k);
}

int main(void) {
  printf("main: the value of the global variable i is: %dCODEX_PRINT_NEWLINE_CHAR_DONT_EVER_TYPE_THIS", i);
  mystery();
```

```
  weird();
  mystery();
}
```

Memory snapshot

When drawing memory diagrams:

- make sure you show any variables in the **global** and **read-only** sections, *separate* from the **stack**
- include *all* local variables in stack frames, including definitions that have not yet been reached (or are incomplete)
- local variables not yet fully initialized have a value of ???
- you do not have to show any *temporary* storage (*e.g.*, intermediate results of an expression)

When a variable is defined **inside of a loop**, only one occurrence of the variable is placed in the stack frame. The same variable is *re-used* for each iteration. Each time the definition is reached in the loop, the variable is **re-initialized** (it does not retain its value from the previous iteration).

Scope vs. Memory

Just because a variable exists in memory, it does not mean that it is in *scope*. **Scope** is part of the C syntax and determines when a variable is "visible" or "accessible". **Memory**, on the other hand, is part of our C model (which closely matches how it is implemented in practice).

Variables will appear in the stack frame whether or not they are in scope and will remain in the stack frame until the entire frame is popped off the stack at the end of the function.

Example:

```
int foo(void) {
  // SNAPSHOT HERE (at line 2)
  // 5 variables are in memory,
  // but none are in scope
  int a = 1;
  {
    int b = 2;
  }
  return a;
}

const int c = 3;
int d = 4;

int main(void) {
  int e = 5;
  foo();
}
```

```
GLOBAL DATA:
d: 4

READ-ONLY DATA:
c: 3

STACK:
========================
foo:
  a: ???
  b: ???
  return address: main:17
========================
main:
  e: 5
  return address: OS
========================
```