C Model: Memory & Control Flow

Readings: CP:AMA 1, 7.1, 7.2, 7.3, 7.6, 11.1, 11.2, Appendix E

Course Notes: Appendix A.3

- the ordering of topics is different in the text
- some portions of the above sections have not been covered yet

Models of computation

In CS 135, we modelled the computational behaviour of Racket with substitutions (the “stepping rules”).

To *apply* a function, all arguments are evaluated to values and then we substitute the *body* of the function, replacing the parameters with the argument values.

```
(define (my-sqr x) (* x x))

(+ 2 (my-sqr (+ 3 1)))
=> (+ 2 (my-sqr 4))
=> (+ 2 (* 4 4))
=> (+ 2 16)
=> 18
```

In this course, we model the behaviour of C with

- **memory** and
- **control flow**.

Memory review

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.

Review Appendix A.3.

Accessing memory

The smallest accessible unit of memory is a byte.

To access a byte of memory, you have to know its *position*, which is known as the ***address*** of the byte.

For example, if you have 1MB of memory (RAM), the *address* of the first byte is 0 and the *address* of the last byte is 1048575 ($2^{20} - 1$).

Note: 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 the computer memory as a collection of “labeled mailboxes” where each mailbox stores a byte.

address (1 MB of storage)	contents (one byte per address)
`0x00000`	`00101001`
`0x00001`	`11001101`
...	...
`0xFFFFE`	`00010111`
`0xFFFFF`	`01110011`

In the above table, the contents are arbitrary examples.

Defining variables

When C encounters a variable **definition**, it

- 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).

For example, with the definition

```
int n = 0;
```

C reserves space (an address) to store `n`, “keeps track of” the address `n`, and stores the value `0` at that address.

A variable *definition* reserves space but a *declaration* does not.

In our CS 135 substitution model, a variable is a “name for a value”.

When a variable appears in an expression, the name is *substituted* for its value.

In our new model, a variable is a “name for a location” where a value is stored.

When a variable appears in an expression, C “fetches” the contents at its address to obtain the value stored there.

sizeof

When we define a variable, C reserves space in memory to store that variable – but **how much space?**

It depends on the **type** of the variable.

It may also depend on the *environment* (the machine and compiler).

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.

```
int n = 0;
printf("the size of an int is: %zd\n", sizeof(int));
printf("the size of n is:      %zd\n", sizeof(n));
```

```
the size of an int is: 4
the size of n is:      4
```

In our `Seashell` environment, each `int`eger is 4 bytes (32 bits).

The placeholder for a size is `"%zd"` (the type is `size_t`).

In C, the size of an `int` depends on the machine (processor) and/or the operating system that it is running on.

Every processor has a natural ***"word size"*** (*e.g.,* 32-bit, 64-bit). Historically, the size of an `int` was the word size, but most modern systems use a 32-bit `int` to improve compatibility.

In C99, the `inttypes` module (`#include <inttypes.h>`) defines many types (*e.g.,* `int32_t`, `int16_t`) that specify *exactly* how many bits (bytes) to use.

In this course, you should only use `int`, and there are always 32 bits in an `int`.

example: variable definition

```
int n = 0;
```

For this variable definition C reserves (or “finds”) 4 consecutive bytes of memory to store `n` (*e.g.*, addresses `0x5000...0x5003`) and then “keeps track of” the first (or “*starting*”) address.

identifier	type	# bytes	starting address
n	`int`	4	`0x5000`

C updates the contents of the 4 bytes to store the initial value (0).

address	`0x5000`	`0x5001`	`0x5002`	`0x5003`
contents	`00000000`	`00000000`	`00000000`	`00000000`

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 $-2^{31} \ldots (2^{31} - 1)$ or -2,147,483,648 . . . 2,147,483,647.

If you `#include <limits.h>`, the constants `INT_MIN` and `INT_MAX` are defined with those limit values.

`unsigned int` variables represent the values $0 \ldots (2^{32} - 1)$ but we do not use them in this course.

Overflow

If we try to represent values outside of the integer limits, ***overflow*** occurs.

For example, when you add one to 2,147,483,647 the result is -2,147,483,648.

By carefully specifying the order of operations, you can sometimes avoid overflow.

You are not responsible for calculating overflow, but you should understand why it occurs and how to avoid it.

In CS 251 / CS 230 you will learn more about overflow.

example: overflow

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

printf("the value of 1 billion is: %d\n", bil);
printf("the value of 4 billion is: %d\n", four_bil);
printf("the value of 9 billion is: %d\n", nine_bil);
```

```
the value of 1 billion is: 1000000000
the value of 4 billion is: -294967296
the value of 9 billion is: 410065408
```

Racket can handle arbitrarily large numbers, such as `(expt 2 1000)`.

Why did we not have to worry about overflow in Racket?

Racket does not use a fixed number of bytes to store numbers. Racket represents numbers with a *structure* that can use an arbitrary number of bytes (imagine a *list* of bytes).

There are C modules available that provide similar features (a popular one is available at `gmplib.org`).

The char type

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).

There are only 2^8 (256) possible values for a `char` and the range of values is (-128 ... 127) in our `Seashell` environment.

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

ASCII

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.

Upper case A is 65, while lower case a is 97. A space is 32.

ASCII was developed when *teletype* machines were popular, so the characters 0 . . . 31 are teletype “control characters” (*e.g.,* 7 is a “bell” noise).

The only control character we use in this course is the line feed (10), which is the newline `\n` 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
*/
```

ASCII worked well in English-speaking countries in the early days of computing, but in today's international and multicultural environments it is outdated.

The **Unicode** character set supports more than $100,000$ characters from all over the world.

A popular method of ***encoding*** Unicode is the `UTF-8` standard, where displayable ASCII codes use only one byte, but non-ASCII Unicode characters use more bytes.

C characters

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

For example, `'a'` is equivalent to 97 and `'z'` is 122.

C “translates” `'a'` into 97.

In C, there is **no difference** between the following two variables:

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

Always use **single** quotes with characters:

`"a"` is **not** the same as `'a'`.

example: C characters

The `printf` placeholder to display a *character* is `"%c"`.

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

printf("letter_a as a character:   %c\n", letter_a);
printf("ninety_seven as a char:    %c\n", ninety_seven);

printf("letter_a in decimal:       %d\n", letter_a);
printf("ninety_seven in decimal:   %d\n", ninety_seven);
```

```
letter_a as a character:   a
ninety_seven as a char:    a

letter_a in decimal:       97
ninety_seven in decimal:   97
```

Character arithmetic

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

```
bool is_lowercase(char c) {
  return (c >= 'a') && (c <= 'z');
}

// to_lowercase(c) converts upper case letters to
//   lowercase letters, everything else is unchanged
char to_lowercase(char c) {
  if ((c >= 'A') && (c <= 'Z')) {
    return c - 'A' + 'a';
  } else {
    return c;
  }
}
```

Structures in the memory model

When a structure **type** is *defined*, no memory is reserved:

```
struct posn {
  int x;
  int y;
};
```

Memory is only reserved when a **variable** is defined.

```
struct posn p1 = {3,4};
```

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

```
struct mystruct {
  int x;
  char c;
  int y;
};

struct mystruct s = {3, 'a', 4};

printf("sizeof(struct mystruct) = %zd\n", sizeof(struct mystruct))
printf("sizeof(s) =               %zd\n", sizeof(s));
```

```
sizeof(struct mystruct) = 12
sizeof(s) =               12
```

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

The size may depend on the *order* of the fields:

```
struct s1 {                struct s2 {
  char c;                    char c;
  int i;                     char d;
  char d;                    int i;
};                         };
```

```
printf("The sizeof s1 is: %zd\n", sizeof(struct s1));
printf("The sizeof s2 is: %zd\n", sizeof(struct s2));
```

```
The sizeof s1 is: 12
The sizeof s2 is: 8
```

C may reserve more space for a structure to improve *efficiency* and enforce *alignment* within the structure.

Floating point numbers in C

The C `float` (floating point) type can represent real (non-integer) values and has a much larger range than integers.

```
float pi = 3.14159;
float avagadro = 6.022e23;   // 6.022*10^23
```

Unfortunately, `float`s are susceptible to precision errors.

C's `float` type is similar to **inexact numbers** in Racket (which appear with an `#i` prefix in the teaching languages):

```
(sqrt 2)           ; => #i1.4142135623730951
(sqr (sqrt 2))     ; => #i2.0000000000000004
```

example 1: inexact floats

```
const float penny = 0.01;

float add_pennies(int n) {
  if (n == 0) {
    return 0;
  } else {
    return penny + add_pennies(n-1);
  }
}

int main(void) {
  float dollar = add_pennies(100);
  printf("the value of one dollar is: %f\n", dollar);
}
```

```
the value of one dollar is: 0.999999
```

The `printf` placeholder to display a `float` is `"%f"`.

example 2: inexact floats

```
const float bil = 1000000000;
const float bil_and_one = bil + 1;

printf("a float billion is:     %f\n", bil);
printf("a float billion + 1 is: %f\n", bil_and_one);
```

```
a float billion is:     1000000000.000000
a float billion + 1 is: 1000000000.000000
```

In the previous two examples, we highlighted the precision errors that can occur with the `float` type.

C also has a `double` type that is still inexact but has significantly better precision.

Just as we used `check-within` for inexact numbers in Racket, use a similar technique for testing in C.

Assuming that the precision of a `double` is perfect or “good enough” can be a serious mistake and introduce errors.

Unless you are explicitly told to use a `float` or `double`, you should not use them in this course.

Just as we might represent a number in decimal as 6.022×10^{23}, a `float` uses a similar strategy.

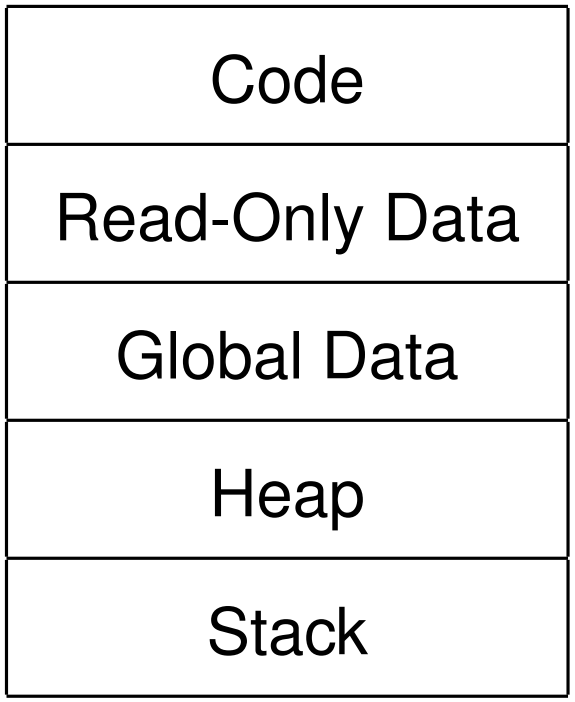
A `float` in our `Seashell` environment uses 32 bits: 24 bits for the *mantissa* and 8 bits for the *exponent*.

A `double` uses 64 bits (53 + 11).

`float`s and their internal representation are discussed in CS 251 / 230 and in detail in CS 370 / 371.

Sections of memory

In this course we model five ***sections*** (or “regions”) of memory:



Other courses may use alternative names.

The **heap** section is introduced in Section 10.

Sections are combined into memory ***segments***, which are recognized by the hardware (processor).

When you try to access memory outside of a segment, a **segmentation fault** occurs (more on this in CS 350).

When evaluating C expressions, the intermediate results must be *temporarily* stored.

```
  a = f(3) + g(4) - 5;
```

In the above expression, C must temporarily store the value returned from `f(3)` “somewhere” before calling `g`.

In this course, we do not discuss this “temporary” storage, which is covered in CS 241.

The code section

When you write your 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.

Converting source code into machine code is known as ***compiling***. It is briefly discussed in Section 13 and covered extensively in CS 241.

The read-only & global data sections

Earlier we described how C reserves space in memory for a variable definition. For example:

```c
int n = 0;
```

The location of the reserved space depends on whether the variable is **global** or **local**. First, we discuss global variables.

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

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 (all of the modules) 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**.

We discuss local variables and the **stack** section after control flow.

The read-only and global memory sections are created and initialized when the code is compiled.

Control flow

In our C model, 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.

In hardware, the *location* is known as the ***program counter***, which contains the *address* within the machine code of the current instruction (more on this in CS 241).

```c
int g(int x) {
  return x + 1;
}

int f(int x) {
  return 2 * x + g(x);
}

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

When a function is called, the program location “jumps” to the start of the function. The `return` keyword “returns” the location *back* to the calling function.

The return address

For each function call, we need to "remember" the program location to "jump back to" when we `return`. This location is known as the ***return address***.

In this course, we use the name of the function and a line number (or an arrow) to represent the return address.

In practice, the *return address* is the address of the machine instruction following the function call.

The call stack

Suppose the function `main` calls `f`, then `f` calls `g`, and `g` calls `h`.

As the program flow jumps from function to function, we need to "remember" the "history" of the return addresses. When we `return` from `h`, we jump back to the return address in `g`. The "last called" is the "first returned".

This "history" is known as the ***call stack***. Each time a function is called, a new entry is *pushed* onto the stack. Whenever a `return` occurs, the entry is *popped* off of the stack.

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**
- any **local variables** that appear within the function *block* (including any sub-blocks), and
- the *return address*.

As with Racket, **before** a function can be called, all of the **arguments must be values**.

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.

Space for a *global* variable is reserved before the program begins execution.

Space for a *local* variable is reserved **when the function is called**. The space is reserved within the newly created stack frame.

When the function `return`s, the variable (and the entire frame) “disappears”.

In C, local variables are known as *automatic* variables because they are “automatically” created when needed. There is an `auto` keyword in C but it is rarely used.

example: stack frames

```
 1 int h(int i) {
 2   int r = 10 * i;
 3   return r;
 4 }
 5
 6 int g(int y) {
 7   int c = y * y;
 8   ⇒ return c;
 9 }
10
11 int f(int x) {
12   int b = 2*x + 1;
13   return g(b + 3) + h(b);
14 }
15
16 int main(void) {
17   int a = f(2);
18   //...
19 }
```

```
=========================
g:
  y: 8
  c: 64
  return address: f:13
=========================
f:
  x: 2
  b: 5
  return address: main:17
=========================
main:
  a: ???
  return address: OS
=========================
```

Recursion in C

Now that we understand how stack frames are used, we can see how *recursion* works in C.

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.

In this example, we will also see control flow with the `if` statement.

example: recursion

```
1  int sum_first(int n) {
2    ⇒if (n == 0) {
3      return 0;
4    } else {
5      return n + sum_first(n-1);
6    }
7  }
8
9  int main(void) {
10   int a = sum_first(2);
11   //...
12 }
```

```
==============================
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

In most situations, mutable variables *should* be initialized, but C will allow you to define variables without any initialization.

```
int i;
```

For all **global** variables, C will automatically initialize 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.

```
int g = 0;
```

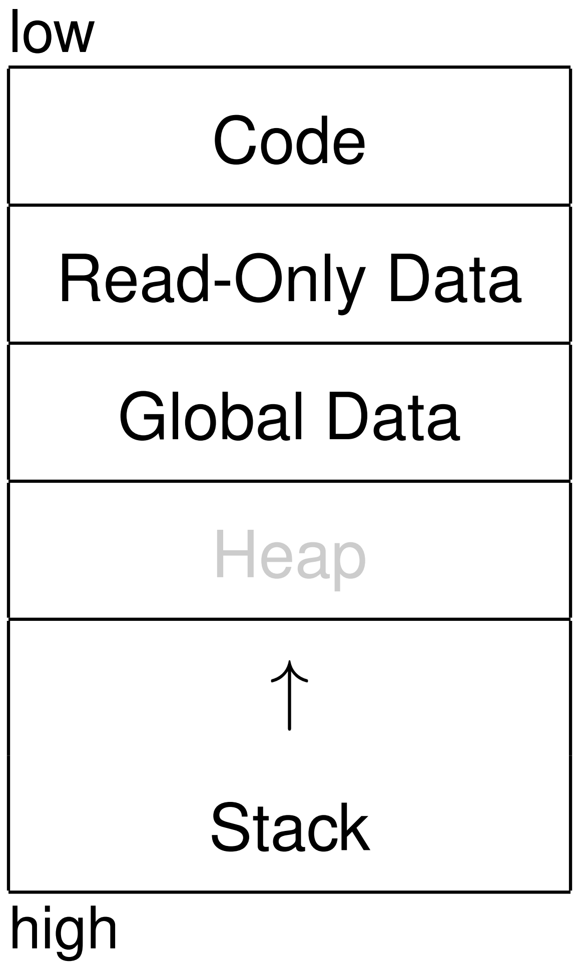
A **local** variable (on the *stack*) that is uninitialized has an **arbitrary** initial value.

```
void mystery(void) {
  int k;
  printf("the value of k is: %d\n", k);
}
```

`Seashell` gives you a warning if you access an uninitialized variable.

In the example above, the value of `k` will likely be a leftover value from a previous stack frame.

Memory sections (so far)



Model

We now have the tools to model the behaviour of a C program.

Any any moment of execution, a program is in a specific *state*, which is the combination of:

- the current *program location*, and
- the current contents of the *memory*.

To properly interpret a program's behaviour, we must keep track of the program location and all of the memory contents.

For the remainder of this Section we will discuss the **control flow** mechanisms in C.

Calling a function

Calling a function is control flow. When a function is called:

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

return

We have already seen the `return` control flow statement.

When a function `return`s:

- 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 value (for non-`void` functions) is stored in a *temporary* memory area we are not discussing in this course. This will be discussed further in CS 241.

if statement

We briefly introduced the `if` control flow statement in Section 03.

We now discuss `if` in more detail.

The syntax of `if` is

```
if (expression) statement
```

where the `statement` is only executed `if` the `expression` is true (non-zero).

```
if (n < 0) printf("n is less than zero\n");
```

Remember: the `if` statement does not produce a value. It only controls the flow of execution.

The `if` statement only affects whether the *next* statement is executed. To execute more than one statement when the expression is true, braces (`{}`) are used to insert a compound statement block (a sequence of statements) in place of a single statement.

```
if (n <= 0) {
  printf("n is zero\n");
  printf("or less than zero\n");
}
```

Using braces is **strongly recommended** *even if there is only one statement*. It makes the code easier to follow and less error prone. *(In the notes, we omit them only to save space.)*

```
if (n <= 0) {
  printf("n is less than or equal to zero\n");
}
```

The `if` statement also supports an `else` statement, where a second statement (or block) is executed if the expression is false.

```
if (expression) {
  statement(s)
} else {
  statement(s)
}
```

`else`s can be combined with more `if`s for multiple conditions.

```
if (expression) {
  statement(s)
} else if (expression) {
  statement(s)
} else if (expression) {
  statement(s)
} else {
  statement(s)
}
```

If there is an `if` condition to `return`, there may not be a need for an `else` block. In this simple example, there is not much difference but in larger examples it can reduce the number of indentation levels required.

```c
int sum(int k) {
  if (k <= 0) {
    return 0;
  } else {
    return k + sum(k-1);
  }
}

// Alternate equivalent code

int sum(int k) {
  if (k <= 0) return 0;
  return k + sum(k-1);
}
```

Braces are sometimes necessary to avoid a “dangling” `else`.

```
if (y > 0)
  if (y != 5)
    printf("you lose");
else
  printf("you win!");  // when does this print?
```

The C `switch` control flow statement (see CP:AMA 5.3) has structure a similar to `else if` and `cond`, but very different behaviour.

A `switch` statement has “fall-through” behaviour where more than one branch can be executed.

In our experience, `switch` is very error-prone for beginner programmers.

Do not use `switch` in this course.

The C `goto` control flow statement (CP:AMA 6.4) is one of the most disparaged language features in the history of computer science because it can make *"spaghetti code"* that is hard to understand.

Modern opinions have tempered and most agree it is useful and appropriate in some circumstances.

To use `goto`s, you must also have *labels* (code locations).

```
if (k < 0) goto mylabel;
//...
mylabel:
//...
```

Do not use `goto` in this course.

Looping

With mutation, we can control flow with a method known as ***looping***.

```
  while (expression) statement
```

`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`, you should always use braces (`{}`) for a *compound statement*, even if there is only a single statement.

example: while loop

variable	value
i	2

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

OUTPUT:
```

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.

```
// recursion
int sum(int k) {
  if (k <= 0) {
    return 0;
  }
  return k + sum(k-1);
}
```

```
// iteration
int sum(int k) {
  int s = 0;
  while ( k > 0) {
    s += k;
    --k;
  }
  return s;
}
```

When first learning to write loops, you may find that your code is very similar to using *accumulative recursion*.

```
int accsum(int k, int acc) {
  if (k == 0) return acc;
  return accsum(k-1, k + acc);
}

int recursive_sum(int k) {
  return accsum(k, 0);
}
```

```
int iterative_sum(int k) {
  int acc = 0;
  while (k > 0) {
    acc += k;
    --k;
  }
  return acc;
}
```

Looping is a very “imperative” programming method. Without mutation (side effects), the while loop condition would not change, causing an “endless loop”.

Loops can be “nested” within each other.

```
int i = 5;
while (i >= 0) {
  int j = i;
  while (j >= 0) {
    printf("*");
    --j;
  }
  printf("\n");
  --i;
}
```

```
******
*****
****
***
**
*
```

Changing parameter values

Earlier, we saw this example of an iterative function:

```
int sum(int k) {
  int s = 0;
  while (k > 0) {
    s += k;
    --k;
  }
  return s;
}
```

In this code, we mutate `k` within the loop, which may seem odd because `k` is a parameter.

Remember that a **copy** of each argument is passed to the function, so the function `sum` is free to mutate its own copy of `k`.

while errors

A simple mistake with `while` can cause an “endless loop” or “infinite loop”. Each of the following code will produce an endless loop.

```
while (i >= 0)              // missing {}
  printf("%d\n", i);
  --i;

while (i >= 0); {           // extra ;
  printf("%d\n", i);
  --i;
}

while (i = 100) { ... }     // assignment typo

while (1) { ... }           // constant true expression
```

Do while

The `do` control flow statement is very similar to `while`.

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

The difference is that `statement` is always executed *at least* once, and the `expression` is checked at the *end* of the loop.

```
int i = 0;
bool success;            // an uninitialized var (rare!)
do {
  ++i;
  success = guess_pin(i);
} while (!success);
```

break

The `break` control flow statement is useful when you want to exit from the *middle* of a loop.

`break` immediately terminates the current (innermost) loop.

`break` is often used with a (purposefully) infinite loop.

```
while (1) {
  // stuff
  if (early_exit_condition) break;
  // more stuff
}
```

continue

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

```
// only concerned with fun numbers
int i = 0;
while (i <= 9999) {
  ++i;
  if (!is_fun(i)) continue;
  //...
}
```

for loops

The final control flow statement we introduce 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:

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

which can be re-written as a single `for` loop:

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

for vs. while

Recall the `for` syntax.

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

This `while` example

```c
i = 100;                     // setup
while (i >= 0) {             // expression
  printf("%d\n", i);
  --i;                       // update
}
```

is equivalent to

```c
for (i = 100; i >= 0; --i) {
  printf("%d\n", i);
}
```

Most `for` loops follow one of these forms (or “idioms”).

```
// Counting up from 0 to n-1
for (i = 0; i < n; ++i) {...}

// Counting up from 1 to n
for (i = 1; i <= n; ++i) {...}

// Counting down from n-1 to 0
for (i = n-1; i >= 0; --i) {...}

// Counting down from n to 1
for (i = n; i > 0; --i) {...}
```

It is a common mistake to be “off by one” (*e.g.*, using < instead of <=). Sometimes re-writing as a `while` is helpful.

In C99, the “initialization” statement can be a *definition* instead.

This is very convenient for defining a variable that only has *local (block) scope* within the `for` loop.

```
for (int i = 100; i >= 0; --i) {
  printf("%d\n", i);
}
```

For the above `for` loop, the equivalent `while` loop would have an extra block.

```
{
  int i = 100;
  while (i >= 0) {
    printf("%d\n", i);
    --i;
  }
}
```

You can omit any of the three components of a `for` statement.

If the expression is omitted, it is always "true".

```
for (; i < 100; ++i) {...}  // i already initialized

for (;;) {...}              // endless loop
```

You can use the *comma operator* (`,`) to use more than one expression in the "init" and "update" statements of the `for` loop. See CP:AMA 6.3 for more details.

```
for (i = 1, j = 100; i < j; ++i, --j) {...}
```

A `for` loop is *not always* equivalent to a `while` loop.

The only difference is when a `continue` statement is used.

In a `while` loop, `continue` jumps back to the expression.

In a `for` loop, the “update” statement is executed before jumping back to the expression.

Goals of this Section

At the end of this section, you should be able to:

- explain why C has limits on integers and why overflow occurs
- use the `char` type and explain how characters are represented in ASCII
- describe the `float` and `double` types
- explain how C execution is modelled with memory and control flow, as opposed to the substitution model of Racket

- describe the 4 areas of memory seen so far: code, read-only data, global data and the stack
- identify which section of memory an identifier belongs to
- explain a stack frame and its components (return address, parameters, local variables)
- explain how C makes copies of arguments for the stack frame

- use the introduced control flow statements, including (`return`, `if`, `while`, `do`, `for`, `break`, `continue`)
- re-write a recursive function with iteration and *vice versa*
- trace the execution of small programs by hand, and draw the stack frames at specific execution points