C Model: Memory & Control Flow

Readings: CP:AMA 6.1–6.4, 7.1–7.3, 7.6, Appendix E

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

The primary goal of this section is to be able to model how C programs execute.

Models of computation

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

- all arguments are evaluated to values **before** a function can be called ("applied")
- to call ("apply") a function, 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 two complimentary mechanisms:

- **control flow**
- **memory**

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.

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).

Types of control flow

In this course, we explore four types of control flow:

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

Compound statements (blocks)

We have already seen compound statements (blocks) where a **sequence** of statements (and definitions) are executed **in order**.

```
int main(void) {
  trace_int(1 + 1);     // first
  assert(3 > 2);        // second
  int i = 7;            // third (i is now in scope)
  printf("%d\n", i);    // fourth
  return 0;             // fifth
}
```

Function Calls

When a function is called, the program location "jumps" *from* the current location *to* the start of the function.

The `return` control flow statement changes the program location to go *back* to the **most recent** calling function (where it "jumped from").

Obviously, C needs to "keep track" of where to go. We revisit this when we introduce memory later in this section.

example: function call flow

```
void blue(void) {
  printf("three\n");
  return;
}

void green(void) {
  blue();
  printf("four\n");
  return;
}
```

```
void red(void) {
  printf("two\n");
  green();
  printf("five\n");
  return;
}

int main(void) {
  printf("one\n");
  red();
  printf("six\n");
}
```

There is a supplemental video for this slide online.

Conditionals (if)

We introduced the `if` control flow statement in Section 02. 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).

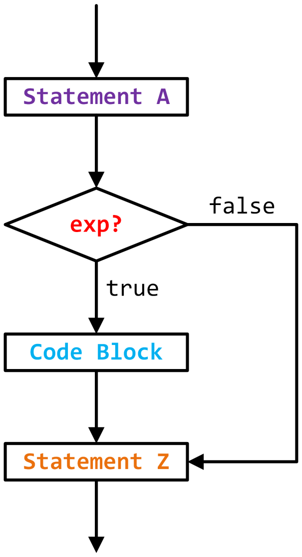
```
if (n < 0) printf("n is less than zero\n");
```

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 conditionally execute **more** than one statement, use a *compound statement* (block).

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

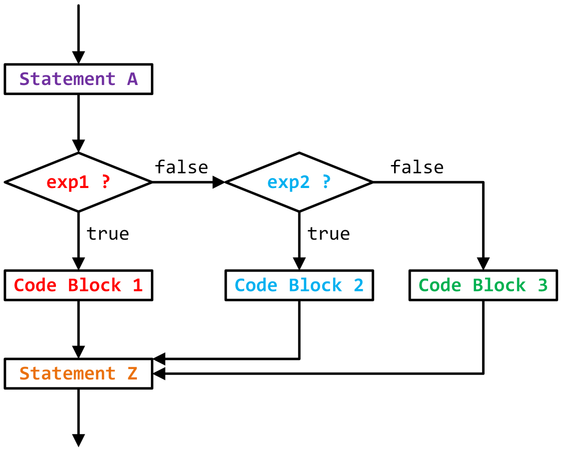
> **Using a block with every `if` is strongly recommended *even if there is only one statement*. It is good style: it makes code easier to follow and less error prone.**
>
> ```
> if (n <= 0) {
>   printf("n is less than or equal to zero\n");
> }
> ```
>
> ***(In the notes, we occasionally omit them to save space.)***



```
Statement A;
if (exp) {
  Code Block;
}
Statement Z;
```

As we have seen, the `if` statement can be combined with `else` statement(s) for multiple conditions.

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



```
Statement A;
if (exp1) {
  Code Block 1;
} else if (exp2) {
  Code Block 2;
} else {
  Code Block 3;
}
Statement Z;
```

If an `if` block contains a `return` statement, there may be no need for an `else` block.

```
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 != 7)
    printf("you lose");
else
  printf("you win!");  // when does this print?
```

The C `switch` control flow statement (see CP:AMA 5.3) has a similar structure 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, *labels* (code locations) are required.

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

Do not use `goto` in this course.

Looping

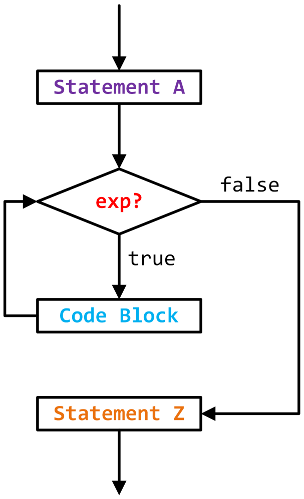
With mutation, we can control flow with a method known as ***looping***.

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



```
Statement A;
while (exp) {
  Code Block;
}
Statement Z;
```

example: while loop

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

```
OUTPUT:
2
1
0
```

There is a supplemental video for this slide online.

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

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

Tracing tools

The provided *tracing tools* can be used to help you understand your control flow and "see" what is happening in your program.

This can help you **debug** your code.

The tracing tools do **not** interfere with your I/O testing.

On your assignments, never `printf` any unnecessary output as it may affect your correctness results.

Always use our tracing tools to help debug your code.

example: tracing tools

```
int sum(int k) {
  trace_msg("sum called");
  int s = 0;
  trace_msg("loop starting");
  while (k > 0) {
    trace_int(k);
    s += k;
    trace_int(s);
    --k;
  }
  trace_msg("loop ended");
  return s;
}

int main(void) {
  trace_int(sum(3));
}
```

```
>> "sum called"
>> "loop starting"
>> k => 3
>> s => 3
>> k => 2
>> s => 5
>> k => 1
>> s => 6
>> "loop ended"
>> sum(3) => 6
```

while errors

A simple mistake with `while` can cause an “endless loop” or “infinite loop”. Each of the following examples are endless loops.

```
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
                            // (this may be on purpose...)
```

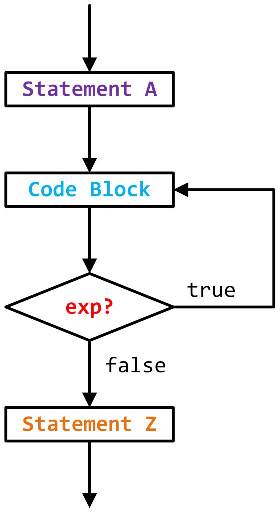
do ... while

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.

```
do {
  printf("try to guess my number!\n");
  guess = read_int();
} while (guess != my_number && guess != READ_INT_FAIL);
```



```
Statement A;
do {
  Code Block;
} while (exp);
Statement Z;
```

break

The `break` control flow statement is useful to exit from the *middle* of a loop. `break` immediately terminates the current (innermost) loop.

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

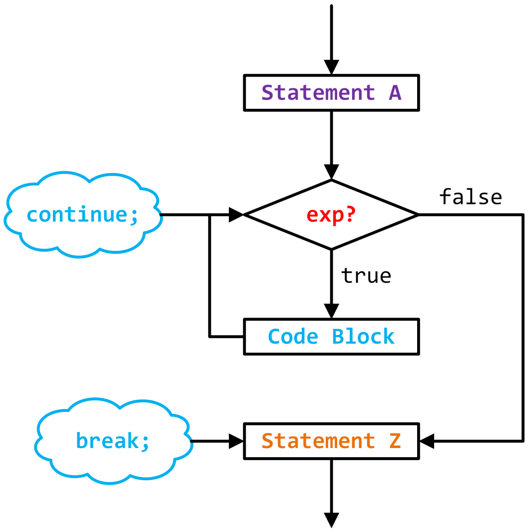
```
while (1) {
  n = read_int();
  if (n == READ_INT_FAIL) {
    break;
  }
  //...
}
```

`break` only terminates loops. You cannot `break` out of an `if`.

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
while (1) {
  n = read_int();
  if (n == READ_INT_FAIL) {
    break;
  }
  if (!is_fun(n)) {
    continue;
  }
  //...
}
```



```
Statement A;
while (exp) {
  Code Block;
}
Statement Z;
```

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.

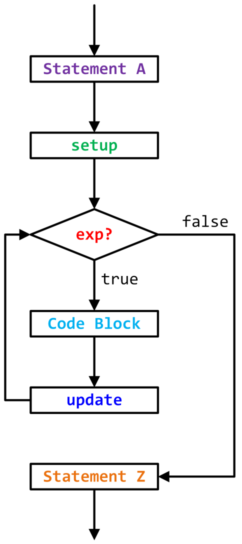
```
for (setup; expression; update) { body statement(s) }
```

This `while` example

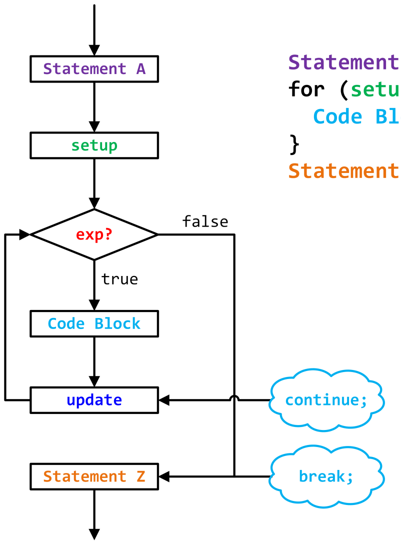
```
i = 100;                   // setup
while (i >= 0) {           // expression
  printf("%d\n", i);
  --i;                     // update
}
```

is equivalent to

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



```
Statement A;
for (setup; exp; update) {
  Code Block;
}
Statement Z;
```



```
Statement A;
for (setup; exp; update) {
  Code Block;
}
Statement Z;
```

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 *setup* can be a **definition**.

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);
}
```

The equivalent `while` loop would have an extra block.

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

Any of the three components of a `for` statement can be omitted.

If the expression is omitted, it is always “true”.

```
for (; i < 100; ++i) {...}  // i was setup previously

for (; i < 100;) {...}      // same as a while(i < 100)

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

The *comma operator* (`,`) allows for multiple sub-expressions in the *setup* and *update* statements of a `for` loop. Do not use it in this course. 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.

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.

In this course, we will usually be dealing with *bytes* and not individual *bits*.

Accessing memory

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.

For example, if you have 1 MB 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 1 MB 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.

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

The *contents* in the above table are arbitrary values.

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).

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.

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

When a variable appears in an expression, a *substitution* occurs and the name is *replaced* by 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 its value – but **how much space** is required?

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;
trace_int(sizeof(int));
trace_int(sizeof(n));
```

```
sizeof(int) => 4
sizeof(n) => 4
```

In this course, the size of an `integer` is 4 bytes (32 bits).

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, 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.

In our CS 136 environment, 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.

In the `read_int` function we provide, the value of the constant `READ_INT_FAIL` is actually `INT_MIN`, so the smallest value of `int` that can be successfully read by our `read_int` function is $-$2,147,483,647.

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.

By carefully specifying the order of operations, sometimes overflow can be avoided.

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

example: overflow

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

trace_int(bil);
trace_int(four_bil);
trace_int(nine_bil);
```

```
bil => 1000000000
four_bil => -294967296
nine_bil => 410065408
```

Remember, do not try to "deduce" what the value of an `int` will be after overflow – its behaviour is **undefined**.

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

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).

There are only 2^8 (256) possible values for a `char` and the range of values is ($-128 \ldots 127$) 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` format specifier 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;
  }
}
```

Reading characters from input

In Section 03, we used the `read_int` function to read integers from input.

We have also provided `read_char` for reading characters.

When reading `int` values, we ignored whitespace in the input.

When reading in characters, you **may** or **may not** want to ignore whitespace characters, depending on the application.

`read_char` has a parameter for specifying if whitespace is ignored.

Symbol type

There is no C equivalent of the Racket `'symbol` type.

C ***symbols*** are constants (often `int`s) with meaningful identifiers (“names”) but arbitrary (meaningless) values.

Use `ALL_CAPS` for symbol names.

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

int direction = UP;
```

We have provided some tools for working with C “symbols” on your assignments.

In C, there are ***enumerations*** (`enum`, CP:AMA 16.5) which allow you to create your own `enum` types and help to facilitate defining constants with unique integer values.

Enumerations are an example of a C language feature that we do *not* introduce in this course.

After this course, we would expect you to be able to read about `enum`s in a C reference and understand how to use them.

If you would like to learn more about C or use it professionally, we recommend reading through all of CP:AMA *after* this course is over.

Floating point types

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

```
float pi = 3.14159;
float avogadro = 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

```
float penny = 0.01;
float money = 0;

for (int n = 0; n < 100; ++n) {
  money += penny;
}

printf("the value of one dollar is: %f\n", money);
```

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

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

example 2: inexact floats

```
float bil = 1000000000;
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 use `check-within` with inexact numbers in Racket, we can use a similar technique for testing in floating point numbers 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`, do not use them in this course.

Floats in memory

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

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

A 32 bit `float` uses 24 bits for the *mantissa* and 8 bits for the *exponent*.

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

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

Structures

Structures (*compound data*) in C are similar to structures in Racket.

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

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

Do not forget the last semicolon (`;`) in the structure definition.

The structure *type* includes the keyword "`struct`". For example, the type is "`struct posn`", not just "`posn`". This can be seen in the definition of `p` below.

```
struct posn p = {3, 4};    // note the use of {}

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

```
p.x => 3
p.y => 4
```

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

The syntax is `variablename.fieldname`

C99 supports an alternative way to initialize structures:

`struct posn p = { .y = 4, .x = 3};`

This prevents you from having to remember the "order" of the fields in the initialization.

Any omitted fields are automatically zero, which can be useful if there are many fields:

`struct posn p = {.x = 3}; //.y = 0`

Mutation with structures

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

```
struct posn p = {1, 2};
struct posn q = {3, 4};

p = q;
p.x = 23;

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

```
p.x => 23
p.y => 4
```

The braces (`{}`) are **part of the initialization syntax** and can not simply be used in assignment. Instead, just mutate each field.

On rare occasions, you may want to define a new `struct` so you can mutate "all at once".

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

p = {5, 6};                 // INVALID

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

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

The *equality* operator (==) does not work with structures.

You have to define your own equality function.

```
bool posn_equal (struct posn a, struct posn b) {
  return (a.x == b.x) && (a.y == b.y);
}
```

Also, `printf` only works with elementary types. Print each field of a structure individually:

```
struct posn p = {3, 4};
printf("The value of p is (%d, %d)\n", p.x, p.y);
```

```
The value of p is (3, 4)
```

Structures in the memory model

For a structure *definition*, no memory is reserved:

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

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

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

sizeof a struct

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

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

```
trace_int(sizeof(struct mystruct));
```

```
sizeof(struct mystruct) => 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;
};                               };
```

```
trace_int(sizeof(struct s1));
trace_int(sizeof(struct s2));
```

```
sizeof(struct s1) => 12
sizeof(struct s2) => 8
```

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

Sections of memory

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

Code
Read-Only Data
Global Data
Heap
Stack

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).

Temporary results

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 are not concerned with this "temporary" storage.

Temporary storage is discussed in CS 241.

The code section

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.

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:

```
int n = 0;
```

The location of memory depends on whether the variable is **global** or **local**.

First, we discuss global variables.

All 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 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**.

The read-only and global memory sections are created and initialized at compile time.

example: read-only & global data

```
// global

int gmv = 13;
int n = 0;

// read-only

const int c = 42;
```

Memory Diagram:

Global	
gmv	13
n	0

Read-Only	
c	42

Function Calls (revisited)

Recall the control flow for function calls:

- When a function is called, the program location “jumps” *from* the current location *to* the start of the function
- `return` changes the program location to go *back* to the **most recent** calling function (where it “jumped from”)
- C needs to track where it “jumped from” so it knows where to `return` to

We model function calls with a ***stack***, and store the information in the **stack section** of memory.

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.

The call 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 `return`s, it is *popped* off of the stack, and the control flow returns to the function now on *top*.

example: call stack

```
void blue(void) {
  return;
}

void green(void) {
  // DRAW DIAGRAM
  return;
}

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

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

Call Stack
(when green is called)

green
red
main

When green returns:
∗ green is popped
∗ red is now on top
∗ control flow returns to red

example: call stack 2

```
void blue(void) {
  // DRAW DIAGRAM
  return;
}

void green(void) {
  return;
}

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

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

The return address

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 `return`ing.

This location is known as the ***return address***.

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

The *operating system* calls the `main` function, so that is shown as `"OS"`.

example:return address

```
 1 void foo(void) {
 2   printf("inside foo\n");
 3   return;
 4 }
 5
 6 int main(void) {
 7   printf("inside main\n");
 8   foo();
 9   printf("back from foo\n");
10 }
```

When `foo` is called, the program location is on line 8 of the function `main` so we would record the **return address** as:

```
main: 8
```

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 in the *calling* function to *return to*)

example: stack frames

```
 1 int pow4(int j) {
 2   printf("inside pow4\n");
 3   int k = j * j;
 4   // DRAW DIAGRAM
 5   return k * k;
 6 }
 7
 8 int main(void) {
 9   printf("inside main\n");
10   int i = 1;
11   printf("%d\n", pow4(i + i));
12 }
```

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

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.

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 `return`s, the variable (and the entire frame) is popped and effectively “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.

Calling a function

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

return

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 is discussed further in CS 241.

Return vs. return address

Beginners often confuse the return address and `return` statements.

Remember, The return address is a **location from the calling function**.

It has **nothing** to do with the **location of** any `return` statement(s), 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

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 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 allows variable definitions without any initialization.

```
int i;
```

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.

```
int g = 0;
```

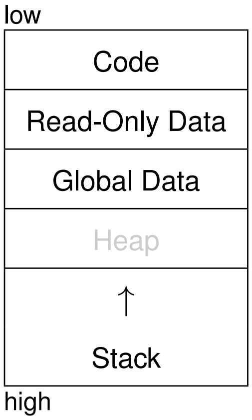
A **local** variable (on the *stack*) that is uninitialized has an **arbitrary** initial value.

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

`Seashell` gives you a warning if you obtain the value of 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)



low

Code

Read-Only Data

Global Data

Heap

↑

Stack

high

Memory snapshot

You may be asked to draw a memory diagram (including the call stack) at a particular moment in the code execution.
For example, “draw the memory when line 19 is reached”.

- 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).

```
for (int j = 0; j < 3; ++j) {
  int k = 0;
  k = k + j;
  trace_int(k);
}
```

```
k => 0
k => 1
k => 2
```

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 is part of our C model (which closely matches how it is implemented in practice).

example: snapshot and scope

```
1  int foo(void) {
2    // SNAPSHOT HERE: five
3    // variables are in memory,
4    // but none are in scope
5    int a = 1;
6    {
7      int b = 2;
8    }
9    return a;
10 }
11
12 const int c = 3;
13 int d = 4;
14
15 int main(void) {
16   int e = 5;
17   foo();
18 }
```

```
READ-ONLY DATA:
c: 3
GLOBAL DATA:
d: 4

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

Model

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

At 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.

Goals of this Section

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

- use the introduced control flow statements, including (`return`, `if`, `while`, `do`, `for`, `break`, `continue`)
- re-write a recursive function with iteration and *vice versa*
- explain why C has limits on integers and why overflow occurs
- use the `char` type and explain how characters are represented in ASCII
- use structures in C

- 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
- model the execution of small programs by hand, and draw the memory snapshot (including stack frames) at specific execution points