C Model: Memory & Control Flow

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

Course Notes: Memory Appendix

- 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”).

To call (“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 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 three types of control flow:

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

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.

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

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

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

The `return` control flow statement changes the program location to go back to the **most recent** calling function.

Obviously, C needs to “keep track” of where to go.

We revisit this when we introduce memory later in this section.

---

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

```c
if (expression) {
    statement(s)
} else if (expression) {
    statement(s)
} else if (expression) {
    statement(s)
} else {
    statement(s)
}
```
If an if condition returns, there may be no need for an else.

```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.

```c
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, you must also have `labels` (code locations).

```c
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


<table>
<thead>
<tr>
<th>variable</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>2</td>
</tr>
</tbody>
</table>

⇒ 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.

```c
// 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**.

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

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.

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

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

```c
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`.

```c
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.

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

```c
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.

```c
// only concerned with fun numbers
while (1) {
    n = read_int();
    if (n == READ_INT_FAIL) break;
    if (!is_fun(n)) continue;
    //...
}
```
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) {
    printf("%d\n", i);
    --i; // update
}
```

is equivalent to

```
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 setup statement can be a **definition**.

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

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

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

```c
{
    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”.

```c
for (; i < 100; ++i) {...} // i was setup previously
for (; i < 100;) {...} // same as a while(i < 100)
for (;;) {...} // endless loop
```

You can use the **comma operator** (,) to use more than one expression in the setup and update statements of a **for** loop.

See CP:AMA 6.3 for more details.

```c
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 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.

Accessing memory

The smallest accessible unit of memory is a byte.

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

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 $2^{20} - 1$.

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.

<table>
<thead>
<tr>
<th>address (1 MB of storage)</th>
<th>contents (one byte per address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000</td>
<td>00101001</td>
</tr>
<tr>
<td>0x00001</td>
<td>11001101</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xFFFFE</td>
<td>00010111</td>
</tr>
<tr>
<td>0xFFFFF</td>
<td>01110011</td>
</tr>
</tbody>
</table>

The contents in the above table are arbitrary values.
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

```c
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.

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

**example: variable definition**

```c
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.

<table>
<thead>
<tr>
<th>identifier</th>
<th>type</th>
<th># bytes</th>
<th>starting address</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>int</td>
<td>4</td>
<td>0x5000</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>address</th>
<th>0x5000</th>
<th>0x5001</th>
<th>0x5002</th>
<th>0x5003</th>
</tr>
</thead>
<tbody>
<tr>
<td>contents</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>
**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 $-2,147,483,648 \ldots 2,147,483,647$.

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.

You should 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, you can sometimes avoid overflow.

In CS 251 / CS 230 you learn more about overflow.
```c
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).

### Additional types

Now that we have a better understanding of what an int in C is, we introduce some additional types.
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 $(-128 \ldots 127)$ in our Seashell environment.

Because of this limited range, chars 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 \ldots 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:

```c
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".

```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.

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

// to lowercase(c) converts upper case letters to lowercase letters, everything else is unchanged
```c
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 your application.

`read_char` has a parameter for specifying if whitespace should be ignored.

Symbol type

In C, there is no equivalent to the Racket `symbol` type. To achieve similar behaviour in C, you can define a unique integer for each “symbol”.

```c
const int POP = 1;    // instead of 'pop
const int ROCK = 2;   // instead of 'rock

int my_favourite_genre = POP;
```

It is common to use an alternative naming convention (such as `ALL_CAPS`) when using numeric constants to represent symbols.

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 **enums** 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.

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

Unfortunately, **floats** 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):

```racket
(sqr 2) ; => #i1.4142135623730951
(sqr (sqrt 2)) ; => #i12.000000000000004
```

---

**example 1: inexact floats**

```c
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** placeholder to display a **float** is "\%f".
example 2: inexact floats

```c
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`, you should 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 $6.022 \times 10^{23}$, 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 $(53 + 11)$.

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

```c
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”.

```c
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.

```c
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. You have to print each field of a structure individually:

```c
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:

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

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

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

### sizeof a struct

```c
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.

```c
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:

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

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

```c
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:

<table>
<thead>
<tr>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-Only Data</td>
</tr>
<tr>
<td>Global Data</td>
</tr>
<tr>
<td>Heap</td>
</tr>
<tr>
<td>Stack</td>
</tr>
</tbody>
</table>

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:

```c
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.
The return address

When we encounter a return, we need to know: “what was the address we were at right before this function was called?”

In other words, 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 calling function and a line number (or an arrow) to represent the return address.

In practice, the return address is the address in the machine code immediately 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
- all local variables (both mutable variables and constants) that appear within the function block (including any sub-blocks)
- the return address

The return address is a location from inside the calling function.
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 returns, 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.

```c
int h(int i) {
    int r = 10 * i;
    return r;
}

int g(int y) {
    int c = y * y;
    return c;
}

int f(int x) {
    int b = 2*x + 1;
    int d = g(b + 3) + h(b);
    return d;
}

int main(void) {
    int a = f(2);
    //...
    return 0;
}
```
In `void` functions the `return` is optional, so a `return` automatically occurs when the end of the function block is reached.

```c
void print_size(int n) {
    if (n > 1000000) {
        printf("n is huge\n");
    } else if (n > 10) {
        printf("n is big\n");
    } else {
        printf("n is tiny\n");
    }
}
```

### 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)
- 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 local variables are set when their definition is encountered

### return

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 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.
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.

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

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

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 you to define variables without any initialization.

```c
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.

```c
int g = 0;
```

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

```c
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).

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

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

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 stack frames at specific execution points