C Memory Model

Readings: CP:AMA 1, 7.1, 7.2, 7.3, 7.6, 11.1, 11.2, 14.1, 14.2, Appendix E

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

A brief history of C

C was developed by Dennis Ritchie in 1969-73 to make the Unix operating system more portable.

It was named “C” because it was a successor to “B”, which was a smaller version of the language BCPL.

C was specifically designed to give programmers “low-level” access to memory. It was also designed to be easily translatable into “machine code”.

Today, thousands of popular programs, and portions of all of the popular operating systems are written in C.

Motivation

You are probably aware that computers measure their memory capacity in ***bytes***, and that a byte is eight ***bits***.

A *bit* is a ***binary digit*** that is either 0 or 1.

To have a better understanding of the C memory model, we provide a *brief* introduction to working with *bits* and *bytes*.

A **bit** is short for **b**inary dig**it**.

Decimal notation

In ***decimal*** representation (also known as ***base 10***) there are **ten** distinct ***digits*** (0123456789).

When we write the number 7305 in base 10, we interpret it as:
$7305 = 7 \times 10^3 + 3 \times 10^2 + 0 \times 10^1 + 5 \times 10^0$.

With 4 decimal digits, we can represent 10^4 $(10,000)$ different possible values $(0000 \ldots 9999)$.

> The reason we use base 10 is because we have 10 fingers.

Binary notation

In ***binary*** representation (also known as ***base 2***) there are **two** distinct digits (01).

When we write the number 1011010 in binary, we interpret it as:

$1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
$= 64 + 16 + 8 + 2 = 90$ (in base 10).

With 4 binary digits, we can represent 2^4 (16) different possible values $(0000 \ldots 1111)$ or $(0 \ldots 15)$ in base 10.

In CS 251 / CS 230 you will learn more about binary notation.

Hexadecimal notation

In ***hexadecimal (hex)*** representation (also known as ***base 16***) there are **sixteen** distinct digits (0123456789ABCDEF).

When we write the number 2A9F in hex, we interpret it as:

$$2 \times 16^3 + 10 \times 16^2 + 9 \times 16^1 + 15 \times 16^0$$
$$= 8192 + 2560 + 144 + 15 = 10911 \text{ (in base 10).}$$

The reason *hex* is so popular is because it is very easy to switch between binary and hex representation. Each hex digit corresponds to exactly 4 bits.

Conversion table

Dec	Bin	Hex	Dec	Bin	Hex
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Binary/Hex conversion

To distinguish between 1234 (decimal) and 1234 (hex), we prefix hex numbers with “0x” (0x1234).

Here are some simple hex / binary conversions:

0x1234 = 0001 0010 0011 0100

0x2A9F = 0010 1010 1001 1111

0xFACE = 1111 1010 1100 1110

C recognizes hex numbers:

```
const int num = 0x2A9F; //same as 10911
```

Memory Capacity

Early in computing it became obvious that working with individual bits can be quite tedious.

It was much more convenient to work with 8 bits at a time.

A group of 8 bits was called a **byte**.

Each byte can have 2^8 (256) possible values.

Each byte value can be represented by 2 hex digits: (00...FF)

With today's computers, we can have large memory capacities:

- 1 KB = 1 kilobyte = 1024 bytes
- 1 MB = 1 megabyte = 1024 KB = 1,048,576 bytes*
- 1 GB = 1 gigabyte = 1024 MB = 1,073,741,824 bytes*

* Manufacturers often use different measurements to make their capacities look more impressive. A terabyte (TB) drive is almost always 1,000,000,000,000 bytes instead of 1,099,511,627,776.

Sometimes they measure in bits instead of bytes to look more impressive. Network speeds are often measured in *bits per second* instead of bytes per second.

Primary Memory

Modern computers have ***primary memory*** in addition to ***secondary storage*** (hard drives, solid state drives, flash drives, DVDs etc.).

The characteristics of *secondary storage* devices vary, but *in general*, they have the following properties:

Primary Memory:

- very fast (nanoseconds)
- medium capacity (≈GB)
- high cost ($) / byte
- harder to remove
- erased on power down

Secondary Storage:

- (≈20x-1000x) slower
- large capacity (≈TB)
- low cost ($) / byte
- removable or portable
- persistent after power down

Programs can only “run” in primary memory.

When you “launch” a program, it is copied from secondary storage to primary memory before it is “run”.

In this course, we are always referring to *primary* memory, which is also known as ***Random Access Memory (RAM)*** because you can access any individual byte directly, and you can access the memory in any order you desire (randomly).

Traditional secondary storage devices (hard drives) are faster if you access data *sequentially* (not randomly).

Primary memory became known as **RA**M to distinguish it from sequential access devices.

This terminology is becoming outdated, as solid state drives and flash drives use random access. Also, modern RAM can be faster when accessed sequentially (in "bursts").

Regardless, when you encounter the term "RAM", you should interpret it as *"primary memory"*.

Accessing Memory

To access a byte of memory, you have to know it's *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.

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.

Declaring constants in C

Now that we understand memory addresses, we can see our first example of how C exposes the memory model to us.

When we *declare* a global constant, C reserves space in memory to ***store*** that constant. For example:

```
const int g = 42;
```

C “keeps track of” the *address* of `g` in memory.

When the constant `g` appears in an expression, C “fetches” the value from that memory address.

Address operator

In C, the ***address operator*** (&), produces the address of an identifier. In the example below, &g is the *address of* g.

The printf placeholder to display an address is "%p".

```
const int g = 42;

int main(void) {
  printf("the value of g is:   %d\n", g);
  printf("the address of g is: %p\n", &g);
}
```

```
the value of g is:   42
the address of g is: 0x805b7a0
```

sizeof an int

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

In C, 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.

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

```
printf("the size of an int is: %d\n", sizeof(int));
printf("the size of g is:      %d\n", sizeof(g));
```

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

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), and the size of an `int` is usually the word size.

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`.

Integer limits

If C uses 32 bits to represent an `int`, there are only 2^{32} (4,294,967,296) possible values that can be represented.

Because we want to represent **negative** values, the actual range of integers in C is: $-2^{31} \ldots (2^{31} - 1)$ (-2,147,483,648 . . . 2,147,483,647).

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

Overflow

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

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 will not be 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 and negative binary numbers.

Example: overflow

```
const int bil = 1000000000;
const int four_bil = bil + bil + bil + bil;
const 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 easily handle arbitrarily large numbers such as `(expt 2 1000)`.

Why didn't we 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 `int` type uses *four bytes* to represent an integer.

The `char` type uses only **one byte** to represent an integer.

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

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

> Even though they are often used to store *characters*, "`byte`" might have been a better choice than `char`.

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.

For example, capital A is 65, while lower case a is 97. A space is 32.

ASCII was developed when *teletype* machines were popular, so some of the codes are teletype "control characters". For example, 7 is a "bell" noise, and 10 is a line feed (our newline `\n` in linux).

Only codes 32 . . . 126 are "printable" characters.

```
/*
  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
*/
```

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 will "translate" `'a'` into 97.

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

```
const char letter_a = 'a';
const 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"`.

```
const char letter_a = 'a';
const 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) will convert upper case letters
//   to lowercase letters, everything else is unchanged
char to_lowercase(char c) {
  return ((c >= 'A') && (c <= 'Z')) ?
         c - 'A' + 'a' : c;
}
```

Floats: inexact numbers in C

In Racket, there were ***inexact numbers***. In the teaching languages, they appeared with an `#i` prefix:

```
(sqrt 2)
(sqr (sqrt 2))
```

```
#i1.4142135623730951
#i2.0000000000000004
```

In C, the `float` type is used for *inexact numbers*.

They are even **more inexact** than in Racket.

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

Example 1: inexact floats

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

```
const float penny = 0.01;

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

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

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

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
```

`float`s are a common source of bugs and errors in C.

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

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

The `double` type uses 64 bits (53 + 11), which improves the precision but is still inexact.

You will learn more about `float`s and their inexact nature in CS 251 / 230, and in detail in CS 370 / 371.

Machine code

When you write your program, you write ***source code*** in a text editor, using ASCII characters that are "human readable".

At the heart of each computer is a ***processor***. The *processor* can not ***execute*** source code directly, it can only execute ***machine code***.

To "run" a program in C, the *source code* must first be converted into *machine code*. This is known as ***compiling***.

> Each processor has its own unique machine code language, although some processors are compatible (e.g.: Intel and AMD).

When the following source code is *compiled* (on an Intel machine):

```
int sum_first(int n) {
  return (n <= 1) ? 1 : n + sum_first(n-1);
}
```

It generates the following machine code (shown as bytes):

```
55 89 E5 83 EC 18 83 7D 08 01 7E 13 8B 45 08 83 E8 01 89 04 24 E8
E6 FF FF FF 03 45 08 EB 05 B8 01 00 00 00 C9 C3
```

For insight, the start of the Intel ADD instruction is **03**, which appears in the machine code above.

You will learn how to compile code in CS 241.

When source code is compiled, the identifiers (names) disappear. In the machine code, only *addresses* are used.

The machine code generated for this function:

```
int sum_first(int n) {
  return (n <= 1) ? 1 : n + sum_first(n-1);
}
```

is identical to the code generated for this function:

```
int asdfasdfasdf(int xkcd) {
  return (xkcd <= 1) ? 1 : xkcd + asdfasdfasdf(xkcd-1);
}
```

One of the most significant differences between C and Racket is that C is *compiled*, while Racket is ***interpreted***.

An *interpreter* reads source code and "translates" it into machine code **while the program is running**. JavaScript and Python are popular interpreted languages.

Another approach that Racket also supports is to compile source code into an intermediate language (*"bytecode"*) that is not machine specific. A *virtual machine* "translates" the bytecode into machine code while the program is running. Java and C# use this approach, which is faster than translating source code directly.

Code in memory

To *execute* machine code, it must first be placed into memory.

For each function, C "keeps track of" the *address* of where the function code **starts** in memory.

```
int sum_first(int n) {
   return (n <= 1) ? 1 : n + sum_first(n-1);
}

int main(void) {
   printf("The address of main:       %p\n", &main);
   printf("The address of main:       %p\n", main); // Same!
   printf("The address of sum_first:  %p\n", sum_first);
}
```

```
The address of main:       0x804d720
The address of main:       0x804d720
The address of sum_first:  0x804d300
```

Surprisingly, the values of `main` and `&main` are equivalent. This is because functions in C are not "values", so C provides the address of the function instead.

The choice of notation (`main` *vs.* `&main`) is a matter of style. We will use `main` as it is cleaner, but `&main` is common in practice because it reminds the reader that it is an *address*.

Whenever the code is *compiled*, the addresses of the functions and the global constants may change. A small change in the source code can result in different address values.

Sections of memory

Our memory model contains five different ***sections*** of memory. We have already seen how programs use two of the sections:

The **code section** is where the *machine code* is placed, and the **read-only section** is where global *constants* are placed.

We will introduce the **stack section** next. The **global data** and **heap** sections are introduced in future modules.

Sections (also known as "regions") are combined into memory ***segments***. When you try to access memory outside of a segment, a *segmentation fault* occurs (more on this in CS 350).

Memory sections (so far)

Racket semantic model

Our semantic model of Racket was based on *substitution 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)) 5)
=> (+ 2 (my-sqr 4) 5)
=> (+ 2 (* 4 4) 5)
=> (+ 2 16 5)
=> 23
```

Control flow

In C, we use ***control flow*** to model how programs are executed.

During execution, the program is in a specific ***state***, which includes the ***program location*** within the code of *"where"* the execution is occurring.

When a program is "run", the *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 251).

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

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

int main(void) {
  const 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 we call, we need to "remember" the program location to "jump back to" when we `return`. This location is known as the ***return address***.

In practice, the *return address* is an address within the machine code section that corresponds to the *next* instruction after the function call.

In this course, we will use the name of the function and sometimes line numbers and/or an arrow to represent the return address.

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”).

In addition to the *return address*, each *stack frame* also contains the **argument values** and any **local constants** that appear within the function *block* (including any sub-blocks).

> The compiler may also reserve additional space within the stack frame to store *temporary* (or *intermediate*) expression values. This is discussed in CS 241.

When a *global* constant is declared, space for the constant is reserved within the *read only section*. This space is available for the entire execution of the program.

When a *local* constant is declared, space for the constant is only reserved **when the function is called**. Space is reserved for the entire stack frame, which includes the space for the constant. When the function `return`s, the space for the frame (including the constant) “disappears”.

As with Racket, **before** a function can be called, all of the arguments **must be values**. The calling function **makes a copy** of each argument value for the stack frame.

Example: stack frames

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

Example: Recursion

```
1  int sum_first(int n) {
2    const int r =
3      (n <= 1) ? 1 :
4      n + sum_first(n-1);
5    ⇒ return r;
6  }
7
8  int main(void) {
9    const int a = sum_first(3);
10   //...
11 }
```

```
sum_first:
  n: 1
  r: 1
  return address: sum_first:4
================================
sum_first:
  n: 2
  r: ???
  return address: sum_first:4
================================
sum_first:
  n: 3
  r: ???
  return address: main:9
================================
main:
  a: ???
  return address: OS
```

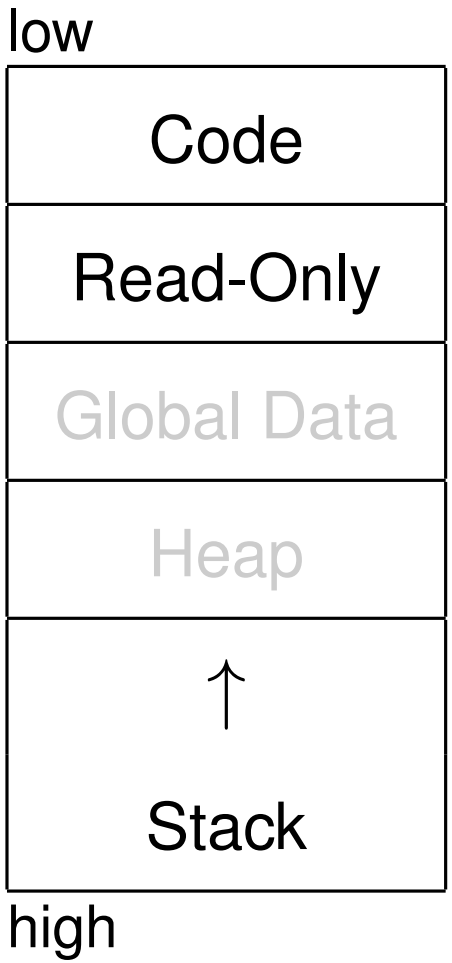
Stack section

The *call stack* is stored in the ***stack section***, the third section of our model. We will simply 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.

Memory sections (so far)



```c
const int r = 42;

int main(void) {

  const int s = 23;

  printf("the address of main is:   %p\n", main); // CODE
  printf("the address of    r is:   %p\n", &r);   // READ-ONLY
  printf("the address of    s is:  %p\n",  &s);   // STACK
}
```

```
the address of main is:   0x804d2d0
the address of    r is:   0x805b8a0
the address of    s is:  0xbff357c0
```

To illustrate the stack “growing down”, we can display the address of the parameter `n` in each recursive call:

```
int sum_first(int n) {
  printf("the address of n is: %p\n", &n);
  return (n <= 1) ? 1 : n + sum_first(n-1);
}
```

```
the address of n is: 0xbff62960
the address of n is: 0xbff62880
the address of n is: 0xbff627a0
the address of n is: 0xbff626c0
the address of n is: 0xbff625e0
the address of n is: 0xbff62500
the address of n is: 0xbff62420
the address of n is: 0xbff62340
the address of n is: 0xbff62260
the address of n is: 0xbff62180
...
```

Pointers

We have seen the *address operator* (&), which obtains the *address* of an identifier, "exposing" the underlying memory model.

In C, there is also a *type* for **storing an address**: a ***pointer***.

A *pointer* is *declared* by placing a *star* (`*`) *before* the identifier (name). The `*` is part of the declaration syntax, not the identifier.

```
const int i = 42;
const int *p = &i;    // p "points at" i
```

The *type* of `p` is an *"int pointer"*, which is written as: `int *`.

For *each type* (e.g.: `int`, `char`) there is a corresponding *pointer type* (e.g.: `int *`, `char *`).

The **value** of a pointer is an **address**.

```
const int i = 42;
const int *p = &i;
const int *q = p;

printf("address of i  (&i) = %p\n", &i);
printf("value of p     (p) = %p\n",  p);
printf("value of q     (q) = %p\n",  q);
```

```
address of i  (&i) = 0xf004
value of p     (p) = 0xf004
value of q     (q) = 0xf004
```

To make working with pointers easier in the notes, we will use shorter, simplified (“fake”) addresses.

C mostly ignores white space, so these are equivalent:

```
const int *p = &i;      // style A
const int * p = &i;     // style B
const int* p = &i;      // style C
```

There is some debate over which is the best style. Proponents of style B & C argue it's clearer that the type of `p` is an "`int *`".

However, *in the declaration* the `*` "belongs" to the `p`, not the `int`, and so style A is used in this course and in CP:AMA.

This is especially clear if multiple declarations are used:

```
const int i = 42, j = 23;
const int *pi = &i, *pj = &j; // VALID
const int* pi = &i, pj = &j;  // INVALID: pj is not a pointer
```

sizeof a pointer

In most k-bit systems, memory addresses are k bits long (and so pointers require k bits to store an address).

In our 32-bit RunC environment, the `sizeof` a pointer is always 32 bits (4 bytes).

The `sizeof` a pointer is **always the same size**, regardless of the type of data stored at that address.

In RunC, our programs can not access more than 2^{32} (4 GB) of memory.

Note: `sizeof(int *)` and `sizeof(char *)` are both 4 bytes.

```
const int i = 42;
const char c = 'c';
const int *pi = &i;
const char *pc = &c;

printf("sizeof(i)      = %d\n", sizeof(i));
printf("sizeof(c)      = %d\n", sizeof(c));
printf("sizeof(pi)     = %d\n", sizeof(pi));
printf("sizeof(pc)     = %d\n", sizeof(pc));
printf("sizeof(int *)  = %d\n", sizeof(int *));
printf("sizeof(char *) = %d\n", sizeof(char *));
```

```
sizeof(i)      = 4
sizeof(c)      = 1
sizeof(pi)     = 4
sizeof(pc)     = 4
sizeof(int *)  = 4
sizeof(char *) = 4
```

Indirection operator

The ***indirection operator*** (∗), also known as the *dereference operator*, is the **inverse** of the *address operator* (&).

∗p will produce the **value** of what pointer p "points at".

```
const int i = 42;
const int *p = &i;      // pointer p points at i
const int j = *p;       // integer j is 42
```

The value of ∗&i is simply the value of i.

Example: indirection

```
const int i = 42;
const int *p = &i;

printf("address of i                (&i) = %p\n",  &i);
printf("value of i                   (i) = %d\n\n", i);

printf("address of p                (&p) = %p\n",  &p);
printf("value of p                   (p) = %p\n",   p);
printf("value of what p points at (*p) = %d\n",  *p);
```

```
address of i                (&i) = 0xf004
value of i                   (i) = 42

address of p                (&p) = 0xf008
value of p                   (p) = 0xf004
value of what p points at (*p) = 42
```

The `*` symbol is used in three different ways in C:

- as the *multiplication operator* between expressions

```
const int k = i * i;
```

- in pointer *declarations* and pointer *types*

```
const int *p = &i;
const int s = sizeof(int *);
```

- as the *indirection operator* for pointers

```
const int j = *p;
```

`(*x * *x)` is a confusing, but valid, C expression.

Pointers to pointers

In the following code, "`pi` points at `i`", but what if we want a pointer to "point at `pi`"?

```
const int i = 42;
const int *pi = &i;       // pointer pi points at i
```

In C, we can declare a **pointer to a pointer**:

```
const int **ppi = &pi;    // pointer ppi points at pi
```

C will allow any number of pointers to pointers, but in practice more than two levels of "pointing" is very rare.

```
const int i     = 42;
const int *pi   = &i;
const int **ppi = &pi;

address of i                                  (&i) = 0xf004
value of i                                     (i) = 42

address of pi                                (&pi) = 0xf008
value of pi                                    (pi) = 0xf004
value of what pi points at                    (*pi) = 42

address of ppi                              (&ppi) = 0xf00C
value of ppi                                  (ppi) = 0xf008
value of what ppi points at                  (*ppi) = 0xf004
value of what ppi points at points at (**ppi) = 42
```

(**x * **x) is a confusing, but valid, C expression.

Pointer parameters

Function parameters and return types can also be pointers.

For now, while we are only working with `const`ants, any parameters and return pointer types must include the `const` keyword.

```
int my_add_with_ptrs(const int *a, const int *b) {
  return *a + *b;
}

const int *ptr_to_max(const int *a, const int *b) {
  return *a >= *b ? a : b;
}
```

The use of `const` with pointers will be discussed further in the next module.

Pointer parameters

```
int my_add_with_ptrs(const int *a, const int *b) {
  return *a + *b;
}

const int *ptr_to_max(const int *a, const int *b) {
  return *a >= *b ? a : b;
}

int main(void) {
  const int x = 3;
  const int y = 4;
  const int s = my_add_with_ptrs(&x, &y);   // note the &
  const int *p = ptr_to_max(&x, &y);        // note the &

  assert (s == 7);
  assert (p == &y && *p == 4);
}
```

A function must **never** return an address within its stack frame.

```
int *bad_idea(int n) {
  return &n;              // NEVER do this
}

int *bad_idea2(int n) {
  const int a = n*n;
  return &a;              // NEVER do this
}
```

As soon as the function `return`s, the stack frame "disappears", and all memory within the frame should be considered **invalid**.

The NULL pointer

`NULL` is a special pointer value to represent that the pointer points to "nothing", or is "invalid". Some functions return a `NULL` pointer to indicate an error. `NULL` is essentially "zero", but it is good practice to use `NULL` in code to improve communication.

If you try to *dereference* a `NULL` pointer, your program will crash.

Most functions should have the precondition that pointer parameters are not `NULL`.

```
assert (p != NULL);   // because NULL is not true...
assert (p);           // this is equivalent and common
```

`NULL` is defined in the `stdlib` module (and several other modules).

Void pointers

The ***void pointer*** **type** (`void *`) is used to store an address where the type of the data is *unknown*, or it does not matter.

Later, we will see functions use *void pointers* to manage memory directly without knowing the type of the data.

We introduce it now because many confuse the `NULL` pointer **value** (a pointer to invalid memory), with the *void pointer* **type** (a type of pointer where the type is unknown).

You **cannot dereference** an address that is stored in a *void pointer*.

The *void pointer* type is the closest C has to a "generic" type.

Function pointers

A ***function pointer*** is used to store the address of a function.

Function pointers can be passed as arguments to functions in a manner *"similar"* to Racket.

The significant difference is that in Racket, functions are *values*, while in C functions are *pointers*. In C, *new* functions cannot be generated and function pointers can only point to functions that already exist.

Also, in C the **type** of the function must be known, which includes the *return type* and all of the *parameter types*. This makes function pointer *declarations* a little messy.

The syntax to declare a function pointer with name `id` is:

```
return_type (*id)(param1_type, param2_type, ...)
```

Consider the following example:

```
int add1(int i) {
  return i + 1;
}

int main(void) {
  int (*fp)(int) = add1;
  const int four = add1(3);
  const int five = fp(4);
  //...
}
```

Examples: function pointer declarations

```
int functionA(int i) {...}
int (*fpA)(int) = functionA;

char functionB(int i, int j) {...}
char (*fpB)(int, int) = functionB;

int functionC(int *ptr, int i) {...}
int (*fpC)(int *, int) = functionC;

int *functionD(int *ptr, int i) {...}
int *(*fpD)(int *, int) = functionD;

struct posn functionE(struct posn p, int i) {...}
struct posn (*fpE)(struct posn , int) = functionE;
```

Example: function pointers

```
// maxval(i,f,g) will return the maximum of f(i) and g(i)

int maxval(int i, int (*f)(int), int (*g)(int)) {
  const int fval = f(i);
  const int gval = g(i);
  return fval > gval ? fval : gval;
}

int ten(int i) {
  return 10 * i;
}

int sqr(int i) {
  return i * i;
}

int main(void) {
  assert( maxval(5,  ten, sqr) ==  50);
  assert( maxval(11, ten, sqr) == 121);
}
```

Aside from illustrating the memory model and using function pointers, pointers may not seem very useful yet.

They will become much more important when we introduce mutation and dynamic memory in upcoming modules.

One additional (small) reason to use pointers is to reduce stack overhead when parameters are *structures*.

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 constant is *declared*:

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

If you need to know how much memory is reserved for a structure, you **must** use the `sizeof` operator.

The `sizeof(struct posn)` is 8.

C may reserve more memory than necessary to store a `struct`.

The `sizeof` may even depend on the *order* of the fields:

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

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

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

Passing structures as arguments

Recall that when a function is called, a **copy** of each argument value is placed into the stack frame.

For structures, the *entire* structure is copied into the frame. For large structures, this can be quite time consuming (inefficient).

```
struct bigstruct {
  int a; int b; int c; ... int y; int z;
};
```

Large structures also increase the size of the stack frame. This can be *especially* problematic with recursive functions, and may even cause a *stack overflow* to occur.

Passing pointers to structures

To avoid the previous problem, function parameters are often *pointers* to structures.

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

int sqr_dist(const struct posn *p1, const struct posn *p2) {
  const int xdist = (*p1).x - (*p2).x;
  const int ydist = (*p1).y - (*p2).y;
  return xdist * xdist + ydist * ydist;
}
```

In the above code, the parentheses `( )` in the expression `(*p1).x` are **required** because the structure operator (`.`) has higher precedence over the indirection operator (`*`).

Writing the expression `(*ptr).field` is a little awkward, and occurs common enough in C that there is an *additional* selection operator.

The ***right arrow selection operator*** (`->`) combines the indirection and the selection operators.

`ptr->field` is equivalent to `(*ptr).field`.

```
int sqr_dist(const struct posn *p1, const struct posn *p2) {
  const int xdist = p1->x - p2->x;
  const int ydist = p1->y - p2->y;
  return xdist * xdist + ydist * ydist;
}
int main(void) {
  const struct posn p1 = {2,4};
  const struct posn p2 = {5,8};
  assert(sqr_dist(&p1,&p2) == 25);
}
```

Goals of this module

You should have a basic understanding of bits and bytes, and how to use decimal, binary and hexadecimal notation.

You should understand why C has limits on integers and why overflow occurs.

You should be comfortable with the `char` type and how characters are represented in ASCII.

You should have a basic understanding of the inexact nature of the `float` type and why it should be avoided.

You should understand that C source code is compiled into machine code before it can be executed.

You should understand how C execution is modelled with control flow, as opposed to the substitution model of Racket.

You should understand the 3 areas of memory seen so far: code, read-only and the stack, and be able to identify which section of memory an identifier belongs to.

You should understand how and why stack frames are used, the components of a stack frame (return address, parameters, local constants), and be able to construct a stack frame for a function.

You should understand how C makes copies of arguments for the stack frame.

You should understand how to declare and deference pointers.

You should understand how structures are represented in the memory model, and why parameters are often pointers to structures.

You should be comfortable with the three new operators (& * ->).

You should be able to trace the execution of small programs by hand, and produce a memory model diagram and/or draw stack frames at specific execution points