An Introduction and Review of C

CP:AMA Readings: 2.2, 2.3, 2.7, 4.1, 5.1, 6.1–6.4, 9.1

CHTP Readings: 2–4

- the ordering of topics is different in the texts

The primary goal of this section is to give a basic overview of programming in C.

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 and be easily translatable into “machine code”.

Thousands of popular programs and portions of **all** of the popular operating systems are written in C.

In this course, we use the C99 standard (from 1999).

gcc -Wall

Comments

In C, any text on a line **after** `// is a comment.`

Any text between `/*` and `*/` is also a comment.

`/* ... */` can extend over multiple lines and can comment out large sections of code.

```
// C comment (one-line only)

/* This is a
   multi-line comment */
```

C's multi-line comment cannot be "nested":

```
/* this /* nested comment is an */ error */
```

Expressions

C expressions use traditional algebraic notation: (*e.g.*, `3 + 3`).

Use parentheses to specify the **order of operations**
(normal arithmetic rules apply).

7 6

`1 + (3 * 2)` $\Rightarrow$ `7`

`(1 + 3) * 2` $\Rightarrow$ `8`

4 * 2 → 8

Operators

In addition to the traditional mathematical ***operators*** (*e.g.,* +, -, $*$), C also has *non-mathematical* operators (*e.g.,* data operators).

With over 40 operators in total, the order of operations is complicated (see CP:AMA Appendix A).

C does not have an *exponentiation* operator (*e.g.,* x^n).

Confusingly, the *"bitwise exclusive or"* operator (^) looks like an exponentiation operator. (We can ignore it in this course.)

In C, each operator is either *left* or *right* associative to further clarify any ambiguity (see CP:AMA 4.1).

The multiplication operators are *left*-associative:

`4 * 5 / 2` is equivalent to `(4 * 5) / 2`.

The distinction in this particular example is important in C.

The / operator

When working with integers, the C division operator (/) truncates (rounds toward zero) any intermediate values.

20 / 2 => 10

```
(4 * 5) / 2    ⇒   10

4 * (5 / 2)    ⇒    8

  -5 / 2       ⇒   -2
```

2

-(5/2) => -2

5/2 => 2.5
in math

5/2 => 2
in C

Remember, use parentheses to clarify the order of operations.

C99 standardized the "(round toward zero)" behaviour.

The % operator

The C **modulo** operator (%) produces the **remainder** after integer division.

```
9 % 2   ⇒   1
9 % 3   ⇒   0
9 % 5   ⇒   4
```

9/2 → 4R1

9/3 → 3R0

9/5 → 1R4

The value of `(a % b)` is equal to: `a - (a / b) * b`.

> It is often best to avoid using % with negative integers.

> `(i % j)` has the same sign as `i` (see CP:AMA 4.1).

C identifiers

Every function and variable requires an ***identifier*** (or “name”).

C identifiers must start with a letter, and can only contain letters, underscores and numbers.

For example: `hst_rate`, `trace_int`, `quick_sort`

`underscore_style` is a popular style for C projects.

In other languages (*e.g.*, Java) `camelCaseStyle` is popular.

In practice, it is important to use the recommended style for the language and/or follow the project (or corporate) style guide.

Anatomy of a function definition

```
int my_add(int a, int b) {
  return a + b;
}
```

- braces (`{}`) indicate the beginning/end of a function ***block***
- `return` keyword, followed by an expression, followed by a semicolon (`;`)
- parameters (`a`, `b`) are separated by a comma
- the function and parameter **types** are specified (*i.e.*, `int`)

We'll cover functions in more depth in Section 2.

Static type system

C uses a ***static type system***: all types **must** be known **before** the program is run and the type of an identifier **cannot change**.

```
int my_add(int a, int b) {
  return a + b;
}
```

The `return` type of `my_add` is an `int` (appears *before* `my_add`).

The parameters `a` and `b` are also both `int`s.

Function terminology

We ***call*** a function by ***passing*** it ***arguments***.

A function ***returns*** a value.

`my_add(1, 2)` $\Rightarrow$ `3`

We *call* `my_add` and *pass* it the *arguments* 1 and 2.

`my_add(1, 2)` *returns* 3.

Function documentation

It is good style to provide a purpose for every function that provides a brief description of **what** the function does (not *how* it does it).

Add a **requires** comment if appropriate.

```
// my_divide(x, y) evaluates x/y using
//   integer division
// requires: y is not 0

int my_divide(int x, int y) {
  return x / y;
}
```

Due to lack of space a purpose statement will not be given for all functions in these notes.

Whitespace

C mostly ignores whitespace.

```
// The following three functions are equivalent

int my_add(int a, int b) {                   // GOOD
  return a + b;
}

int my_add(int a,int b){return a+b;}         // BAD

int my_add(int a, int                        // RIDICULOUSLY
b){return a+                                 // BAD
b ; }
```

The course staff and markers may not follow your code if it is poorly formatted.

Coding style

```
int my_add(int a, int b) {
   return a + b;
}
```

if (...) {
if (...) {
//code

- a block start (open brace {) appears at the end of a line
- a block end (close brace }) is aligned with the line that started it, and appears on a line by itself
- indent the contents of a block using either spaces or tabs consistently
- add a space after commas and around arithmetic operators

When there are a large number of parameters, a large expression or a long purpose, continue (indented) on the following line.

```
// my_super_long_function(a, b, c, d, e, f, g) does some
//   amazing things with those parameters...

int my_super_long_function(int a, int b, int c, int d,
                           int e, int f, int g) {
  return a * b + b * c + c * d + d * e + e * f +
         f * g + g * a;
}
```

The "best" way to style code (*e.g.,* block formatting) is a matter of taste and is often a topic of debate.

The style described here is widely accepted for C (and C++) projects (*e.g.,* it conforms to the Google style guide).

Entry point

Typically, a program is “run” (or “launched”) by an Operating System (OS) through a shell or another program such as `CLion`.

The OS needs to know where to **start** running the program. This is known as the ***entry point***. → main

In C, the entry point is a special function named `main`.

> Every C program must have one (and only one) `main` function.

main

`main` has no parameters† and an `int` return type.

```
int main(void) {
  //...
  return 0;          // success!
}
```

The `return` value communicates to the OS the "error code"

(also known as the "exit code", "error number" or `errno`).

A successful program `return`s **zero** (no error code).

† `main` has *optional* parameters.

`main` is a special function and does not require an explicit `return` value.

The default value is success (zero) and zero is `return`ed *automatically* if it is not present.

```
int main(void) {
  //...
  return 0;          // this is optional
}
```

Boolean expressions

In C, Boolean expressions do not produce “true” or “false”.

They produce either:

- zero (`0`) for “false”, or
- one (`1`) for “true”.

#include<stdbool.h>

false

true

Comparison operators

The ***equality*** **operator** in C is == (note the **double** equals).

```
(3 == 3)   ⇒   1 (true)
(2 == 3)   ⇒   0 (false)
```

The ***not equal*** **operator** is `!=`.

```
(2 != 3)   ⇒   1 (true)
```

The operators <, <=, > and >= behave exactly as expected.

```
(2 < 3)    ⇒   1 (true)
(2 >= 3)   ⇒   0 (false)
```

Always use a *double* == for equality, not a *single* =.

The accidental use of a *single* = instead of a *double* == for equality is one of the most common programming mistakes in C.

This can be a serious bug (we will revisit this).

It is such a serious concern that it warrants an extra slide as a reminder.

Logical Operators

The Logical operators are: ! (not), && (and), || (or):

```
!(3 == 3)                  ⇒  0
(3 == 3) && (2 == 3)       ⇒  0
(3 == 3) && !(2 == 3)      ⇒  1
(3 == 3) || (2 == 3)       ⇒  1
```

&& |

0

[1 && 0] || 1

1 & 0 & 1

f() && g()

C ***short-circuits*** and stops evaluating an expression when the value is known.

```
(a != 0) && (b / a == 2)
```

F

does not generate an error if a is `0`.

> A common mistake is to use a single & or | instead of && or ||.

All non-zero values are true

Operators that *produce* a Boolean value (*e.g.,* ==) will always produce `0` or `1`.

> Operators (or functions) that *expect* a Boolean value (*e.g.,* &&) will consider **any non-zero value** to be "**true**".
>
> **Only zero (`0`) is "false"**.

T && T → T

```
(2 && 3)        ⇒   1
(0 || -2)       ⇒   1
!5              ⇒   0
```

F || T → T

!T → F

You are not expected to "memorize" the order of operations.

When in doubt (or to add clarity) **add parentheses**.

negation	!
multiplicative	* / %
additive	+ -
comparison	< <= >= >
equality	== !=
and	&&
or	||

bool type

The `bool` *type* is an integer that can only have a value of 0 or 1.

In order to use the `bool` type you need to `#include` the header file `stdbool.h` at the top of your source file.

0 == false

1 == true

```
#include <stdbool.h>

bool is_even(int n) {
  return (n % 2) == 0;
}

bool my_negate(bool v) {
  return !v;
}
```

Assertions

The `assert` function can be used to test functions if you `#include` the header file `assert.h`.

```
assert(my_add(1, 2) == 3);
```

T → assert passes

`assert(exp)` **stops** the program and displays a message if the expression `exp` is false (zero).

If `exp` is true (non-zero), it does "nothing" and continues to the next line of code.

```c
// A simple program with built-in testing

#include <assert.h>

int my_add(int a, int b) {
  return a + b;
}

int main(void) {
  assert(my_add(0, 0) == 0);
  assert(my_add(1, 1) == 2);
  assert(my_add(-2, 1) == -1);
}
```

You are expected to test your own code. Using `assert` is one way to do this.

Function requirements

The `assert` function is also very useful for **verifying function requirements**.

```
// my_divide(x, y) ....
// requires: y is not 0

int my_divide(int x, int y) {
  assert(y != 0);                // assert(y) also works
  return x / y;
}
```

In the slides, we often omit `assert`s to save space.

Multiple requirements

With multiple requirements, it is better to have several small `assert`s.

It makes it easier to determine which assertion failed (which requirement was not met).

```
// my_function(x, y, z) ....
// requires: x is positive
//           y < z

int my_function(int x, int y, int z) {
  assert((x > 0) && (y < z));    // OK

  assert(x > 0);                 // BETTER
  assert(y < z);
  //...
}
```

Text output

To display text output in C, we use the `printf` function.

```
// My first program with text output

#include <stdio.h>

int main(void) {
  printf("Hello, World");
}
```

```
Hello, World
```

int x;

printf("...", x);

%f

%lld

%u

%c

%s

```c
int main(void) {
  printf("Hello, World");
  printf("C is fun!");
}
```

```
Hello, WorldC is fun!
```

The ***newline*** character (\n) is necessary to properly format output to appear on multiple lines.

```c
printf("Hello, World\n");
printf("C is\nfun!\n");
```

```
Hello, World
C is
fun!
```

\n

\r

The first parameter of `printf` is a `"string"`.

To output values, use a ***format specifier*** (the **f** in `printf`) within the string and provide an **additional argument**.

For an integer in "**d**ecimal format" the format specifier is `"%d"`.

```
printf("2 plus 2 is: %d\n", 2 + 2);
```

```
2 plus 2 is: 4
```

In the output, the format specifier is **replaced** by the additional argument value.

Strings will be covered in Section 6.

There can be multiple format specifiers, each requiring an additional argument.

```
printf("%d plus %d is: %d\n", 2, 10 / 5, 2 + 2);
2 plus 2 is: 4
```

To output a percent sign (%), use two (%%).

```
printf("I am %d%% sure you should watch your", 100);
printf("spacing!\n");

I am 100% sure you should watch yourspacing!
```

" Hello" world "

Similarly,

- to print a backslash (\), use two (\\)
- to print a quote ("), add an extra backslash (\")

Functions with side effects

Consider the two functions below:

```
int sqr(int n) {
  return n * n;
}

int noisy_sqr(int n) {
  printf("Yo! I'm squaring %d!\n", n);
  return n * n;
}
```

Both functions `return` the same value.

However, `noisy_sqr` does **more** than `return` a value.

In addition to returning a value, it **also produces output**.

`noisy_sqr` has a ***side effect***.

Side effects and state

In general, a programming *side effect* is when the ***state*** of something "changes".

State refers to the value of some data (or "information") **at a moment in time**.

Consider the following "real world" example: You have a blank piece of paper, and then you write your name on that paper.

You have *changed the state* of that paper: at one moment it was blank, and in the next it was "autographed".

In other words, the *side effect* of writing your name was that you *changed the state* of the paper.

I/O terminology

In the context of I/O, be careful with terminology.

```
int sqr(int n) {
  return n * n;
}
```

Informally, someone might say:

"if you input 7 into `sqr`*, it outputs 49"*.

This is **poor terminology**: `sqr` does not read input and does not print any output.

Instead, say:

*"if 7 is **passed** to* `sqr`*, it **returns** 49"*.

```c
int noisy_sqr(int n) {
  printf("Yo! I'm squaring %d!\n", n);
  return n * n;
}
```

For `noisy_sqr`, say:

*"if 7 is **passed** to `noisy_sqr`, it **outputs** a message and **returns** 49".*

It is common for beginners to confuse **output** (*e.g.*, via `printf`) and the **return value**.

Ensure you understand the correct terminology and **read your assignments carefully**.

Variables

Variables store values.

To define a variable in C, we need (in order):

- the **type** (*e.g.*, `int`)
- the **identifier** (“name”)
- the **initial value**

```
int my_variable = 7;      // definition
```

The equal sign (=) and semicolon (`;`) complete the syntax.

Mutation

When the value of a variable is changed, it is known as ***mutation***.

Mutation is another form of **side effect**.

In C, mutation is achieved with the ***assignment operator*** (=).

```
int main(void) {
  int m = 5;          // definition (with initialization)
  printf("m => %d\n", m);
  m = 6;              // mutation!
  printf("m => %d\n", m);
  m = -1;             // more mutation!
  printf("m => %d\n", m);
}
```

```
m => 5
m => 6
m => -1
```

More assignment operators

The *compound* addition assignment operator (+=) combines the addition (+) and assignment (=) operator.

x = x+2;

```
x += 2;            // x = x + 2;
```

Additional compound operators include: `-=`, `*=`, `/=`, `%=`.

There are also *increment* and *decrement* operators that increase or decrease a variable by one.

x+=1;

use x

use x

x+=1;

```
++x;          x++;          // x += 1;
--x;          x--;          // x -= 1;
```

> x++ produces the “old” value of x and then increments x.
>
> ++x increments x and then produces the “new” value of x.

Control flow

A program is mostly a sequence of statements to be executed.

Control flow is used to change the order of execution of the statements.

For example, the `return` statement "controls the flow" by halting the execution of a function and `return`ing to the caller.

> There can be more than one `return` in a function, but only one value is ever returned.
>
> The function stops when the first `return` is executed.

Conditionals

The `if` control flow statement allows us to have functions with conditional behaviour.

```
int my_abs(int n) {
  if (n >= 0) {           // note: the () are required
    return n;
  } else {
    return -n;
  }
}
```

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

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.)

else if

If there are more than two possible results, use `else if`.

```
// in_between(x, lo, hi) determines if lo <= x <= hi
// requires: lo <= hi

bool in_between(int x, int lo, int hi) {
  assert(lo <= hi);
  if (x < lo) {
    return false;
  } else if (x > hi) {
    return false;
  } else {
    return true;
  }
}
```

switch {
case 0 :
break
case 1 :
case 2 :
default;

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?
```

Looping (iteration)

We can also 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.

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>=0)
;
{
printf(...);
--i;
}
```

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

while (1) { ... }                 // constant true expression
                                  // (this may be on purpose)
```

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) {
  int n = scanf("%d", &n);
  if (n == 0) {
    break;
  }
  //...
}
```

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

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
}
```

setup

body

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) {...}
```

0 -> n-1

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

1 -> n

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

n-1 -> 0

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

n -> 1

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
```

while(1)

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.

short 2 bytes

char 1 byte

long 8 bytes

int 4 byte

2.7 B

float 4 bytes

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

Double 8 bytes

Accessing memory

1024 × 1024 Bytes

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 ***hexadecimal*** (and prefixed with 0x), so with 1 MB of memory, the address of the first byte is 0x0, and the address of the last byte is 0xFFFFF.

1, 2, 3, ..., 9, A, B, C, D, E, F

10 11 12 13 14 15

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`

1 byte

4 byte

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

```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.

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.

int ⇒ 4 bytes

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

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

```
int n = 0;
printf("sizeof(int) => %d\n", sizeof(int));
printf("sizeof(n) => %d\n", sizeof(n));
```

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

> In these notes, the size of an `int`eger 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.

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`

1

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

`unsigned int` variables represent the values $0 \ldots (2^{32} - 1)$.

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.

example: overflow

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

printf("     bil => %d\n", bil);
printf("four_bil => %d\n", four_bil);
printf("nine_bil => %d\n", 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**.

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 either ($-128 \ldots 127$) or ($0 \ldots 255$) depending on the 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.

B is 66 b is 98

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
*/
```

$(c \geq \text{'A'} \ \&\& \ c \leq \text{'Z'})$ $(\text{'f'} - \text{'a'}) + \text{'A'}$

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. '\t'

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;
  }
}
```

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.

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
```

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`, `for`, `break`)
- explain why C has limits on integers and why overflow occurs
- use the `char` type and explain how characters are represented in ASCII
- print output with `printf`