The C Programming Language

Rob Hackman

Winter 2025

University of Alberta

Table of Contents

The C Programming Language

Originally developed at Bell Labs[1] by Dennis Ritchie and Ken Thompson.

C was developed for use rewriting the Unix kernel, and has been used for many kernels written since.

The C programming language is a high-level, compiled, and statically typed language.

[1] Yes, *that* Bell... sort of

High-level Languages

Programs that run on our computer must be files made up of specific *machine-code* instructions our CPU was designed to understand — these instructions are not very human readable.

High-level languages — languages we've created to make programming easier, but are removed from the actual machine code your computer's CPU executes.

Python and C are both high-level languages, however if high-level and low-level are instead thought of as a spectrum then C is much lower than Python on that spectrum.

Compiled Languages

Compiled languages — compiling is one way we take high-level languages and make them usable on our CPU, the compiler generates a low-level machine code program from our high-level code. We call these files *binary files* or *binaries* as machine code is all binary.

Python is *interpreted* — we used another program (an interpreter) to execute our Python programs. The interpreter was the actual program executing, using our Python programs as inputs.

In C we will use a compiler, `gcc`[1], which will generate an executable binary file from our C programs

[1] Really when we invoke `gcc` we'll be executing four tools, a preprocessor, compiler, assembler, and linker

Statically Typed languages

Statically typed languages — in these languages the programmer must specifie the type of all variables when declaring them, and this type cannot change.

Python was *dynamically typed* which meant it allowed for changing the types variables referred to, and one never had to specify which type they meant to store with a variable.

```
# Python Code
pythonVar = 5 # pythonVar stores an int
pythonVar = "Hello" # Now pythonVar stores a string
```

```
// C code
int cVar = 5; // cVar stores an int
cVar = "Hello"; // Not what you think...
```

Program's Entry Point

When asked to execute a program your operating system must *load* that program into memory, and load particular information for execution.

One important piece of information is where to begin executing this program — this is called the *entry point*.

The compiler defines in the entry point in our binary file based on our C code. Our entry point in C will always be the function `main`.

Thus, unlike Python, all our programs will require a `main` function.

First C Program

This program may seem simple, but there's actually a lot to unpack here.

```c
#include <stdio.h>

int main(int argc, char **argv) {
  printf("Hello, world!\n");
  return 0;
}
```

First C Program

```c
#include <stdio.h>
```

This line is an `include` *preprocessor directive*. We'll talk about the preprocessor shortly — for now think of this like an `import` statement in Python in that we will also get access to the library functions inside.

The `stdio` library is C's standard libray of I/O functions. It is from this library that we have access to the function `printf`

First C Program

```
int main(int argc, char **argv) { ... }
```

This is a declaration *and* definition of our function `main`. We have begun our definition of our `main` function by opening up the *function body* with curly braces.

The `int` at the beginning is the return type of our `main` function

Our main function has two parameters `argc` and `argv`. `argc` stores how many command line arguments our program received when it was ran. `argv` is how we access the values passed as command line arguments.

First C Program

```
return 0;
```

This line is a return statement. Much like Python this ends our function's execution and produces that value to the caller of the function.

But if `main` is our entry point to the
program who is the caller of this function receiving the return value?

First C Program

```c
return 0;
```

This line is a return statement. Much like Python this ends our function's execution and produces that value to the caller of the function.

But if `main` is our entry point to the program who is the caller of this function receiving the return value?

The operating system! When the call to `main` which began the program returns the value produced is the exit status of our program!

First C Program

```c
printf("Hello, world!\n");
```

Line 4 is calling the `printf` function which will be our main function for printing to the screen.

`printf` requires we provide a string as the first argument, and `"Hello, world!\n"` here is called a string literal — however C does not have a "string" type like Python... so what does it have?

C Types

The types built-in to C are not classes like Python - they are simply numbers stored a particular way.

The types we'll concern ourselves with, for now, are:

- `int` — for representing integers
- `char` — for representing characters
- `float` — for representing floating point numbers

We'll learn a lot more about the types in C soon.

Compiling and Running our Program

Let's save our program from the previous slides in the file `hello_world.c`

In order to run our program we must first compile it to machine code. Let's use the program `gcc` to do so.

```
$ gcc hello_world.c
```

By default `gcc` names our newly compiled program `a.out`, so we can run it by following up with the command

```
$ ./a.out
```

Note we didn't have to make `a.out` executable before being able to run it. The executable binary files produced by `gcc` already have the executable bit set.

Compiling and Running our Program

If we want gcc to give our program a different name we can provide the -o command line argument followed by the name we'd like our newly compiled program to have:

```
$ gcc hello_world.c -o hello_world
$ ./hello_world
```

Warning: Be careful not to ask gcc to output to a filename you already have in use! gcc will overwrite any file already existing at that filepath.

How Numbers are Stored in C

All values in our computer are stored in binary as 1's and 0's[1].

In Python this is hidden away and Python creates its own data structures to abstract these details away from the programmer

In C one must be aware of how numbers are really stored in our computers.

[1]Well, we choose to view it as 1's and 0's, really how it is physically stored depends on what device in our computer its stored in.

Table of Contents

Number Systems - Base 10

Consider what you know about decimal (base-10) integers. What does the number 2357 really mean?

Number Systems - Base 10

Consider what you know about decimal (base-10) integers. What does the number 2357 really mean?

$$2 \times 10^3$$

Number Systems - Base 10

Consider what you know about decimal (base-10) integers. What does the number 2357 really mean?

$$2 \times 10^3 + 3 \times 10^2$$

Number Systems - Base 10

Consider what you know about decimal (base-10) integers. What does the number 2357 really mean?

$$2 \times 10^3 + 3 \times 10^2 + 5 \times 10^1$$

Number Systems - Base 10

Consider what you know about decimal (base-10) integers. What does the number 2357 really mean?

$$2 \times 10^3 + 3 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$$

Number Systems - Base 10

Consider what you know about decimal (base-10) integers. What does the number 2357 really mean?

$$2 \times 10^3 + 3 \times 10^2 + 5 \times 10^1 + 7 \times 10^0$$

This is what the 10 in base-10 means. It is also why our digits only go from 0 to 9, to go above 9 would mean we have 10 of the previous power of 10 which would “roll over” to then next digit.

Number Systems - Base 8

We could instead represent numbers where our base is 8 (octal). This means our digits would only go from 0 to 7. Consider the following octal number 472_8[1]. The value it represents is:

$$4 \times 8^2 + 7 \times 8^1 + 2 \times 8^0$$

The equivalent number in base-10 would be 314.

[1] The subscripted 8 just indicates this is a base-8 number.

Number Systems - Generic Radices

We can select any integer b to be a base for a number system. The base in number systems is also called the *radix*. Then our digits[2] would be from 0 to $b - 1$ and each digit in our numbers in that base would successively represent (right to left) how many of each power b^0, b^1, b^2, ... b^n we have.

[1]Digit comes from the latin word for fingers, chosen since the median number of human fingers is 10. There isn't really a commonly used generic term for "digits" in other bases!

Number Systems - Binary

Binary is the name given to the base-2 number system. Our digits in binary then are only 0 and 1 and are called *bits*. The binary number 1011_2 then represents:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$$

Number Systems - Binary

Since our computers are built of circuits which hold electricity we use a simple scheme for holding values — whether voltage is high or low.

Since we only have these two options we find it very useful to encode numbers in our computer's circuits in binary. We can choose a simple mapping, for example a high voltage represents a 1 and a low voltage represents a 0.

Then with several high and low voltages in an order we can represent numbers.

Note: We find it useful in computer science to group bits together, we call 8-bits a *byte*.

Integers in our Computer - Unsigned Binary

Unsigned integers in your computer are stored as binary numbers, encoded exactly as one would interpret any base-2 number.

However we do not have infinite circuits on our computer. As such we must bound the size of our integers. Commonly integers are 64 or 32 bits long.

Knowledge Check: What's the largest number you could represent in base-10 with 6 digits? What about 15 digits? What's the largest number you could represent in binary with 12 bits? What about with 32 bits?

Unsigned Binary - Addition

Addition is performed on our binary numbers the same way we perform addition on decimal numbers.

$$\begin{array}{r} 1011\ \ 1101 \\ +\ \ 0001\ \ 0101 \\ \hline \end{array}$$

Unsigned Binary - Addition

Addition is performed on our binary numbers the same way we perform addition on decimal numbers.

$$\begin{array}{r} \textcolor{red}{1} \\ 1011\ \ 1101 \\ +\ \ 0001\ \ 0101 \\ \hline 0 \end{array}$$

Unsigned Binary - Addition

Addition is performed on our binary numbers the same way we perform addition on decimal numbers.

$$
\begin{array}{rr}
 & {\color{red}1} \\
 & 1011\ \ 1101 \\
+ & 0001\ \ 0101 \\
\hline
 & 10
\end{array}
$$

Unsigned Binary - Addition

Addition is performed on our binary numbers the same way we perform addition on decimal numbers.

$$
\begin{array}{rr}
 & \ \ \textcolor{red}{1\ 1} \\
 & 1011\ \ 1101 \\
+ & 0001\ \ 0101 \\
\hline
 & 010
\end{array}
$$

Unsigned Binary - Addition

Addition is performed on our binary numbers the same way we perform addition on decimal numbers.

$$\begin{array}{rr}
 & \color{red}{1}\ \color{red}{1}\ \ \color{red}{1} \\
 & 1011\ \ 1101 \\
+ & 0001\ \ 0101 \\
\hline
 & 0010
\end{array}$$

Unsigned Binary - Addition

Addition is performed on our binary numbers the same way we perform addition on decimal numbers.

$$\begin{array}{rr} & 11\ \ 1\ \ \ 1\ \ \\ & 1011\ \ 1101 \\ + & 0001\ \ 0101 \\ \hline & 1\ \ 0010 \end{array}$$

Unsigned Binary - Addition

Addition is performed on our binary numbers the same way we perform addition on decimal numbers.

$$
\begin{array}{ccccccccc}
 & & \color{red}{1} & \color{red}{1} & \color{red}{1} & \color{red}{1} & & \color{red}{1} & \\
 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\
+ & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
 & & & 0 & 1 & 0 & 0 & 1 & 0
\end{array}
$$

Unsigned Binary - Addition

Addition is performed on our binary numbers the same way we perform addition on decimal numbers.

$$\begin{array}{rr} & \color{red}{111\ \ 1\ \ 1} \\ & 1011\ \ 1101 \\ + & 0001\ \ 0101 \\ \hline & 101\ \ 0010 \end{array}$$

Unsigned Binary - Addition

Addition is performed on our binary numbers the same way we perform addition on decimal numbers.

$$\begin{array}{cccccccccc}
 & & \color{red}{1} & \color{red}{1} & \color{red}{1} & & \color{red}{1} & & \color{red}{1} & \\
 & 1 & 0 & 1 & 1 & & 1 & 1 & 0 & 1 \\
+ & 0 & 0 & 0 & 1 & & 0 & 1 & 0 & 1 \\
\hline
 & 1 & 1 & 0 & 1 & & 0 & 0 & 1 & 0
\end{array}$$

Unsigned Binary - Addition

Addition is performed on our binary numbers the same way we perform addition on decimal numbers.

$$
\begin{array}{r}
111\ \ 1\ \ 1\ \ \\
1011\ \ 1101 \\
+\ 0001\ \ 0101 \\
\hline
1101\ \ 0010
\end{array}
$$

What happens if we have to carry over from the most significant digit? — Overflow, the extra bit “carries over” but since we are 8-bit fixed-width it has no where to go and is simply lost.

Data in our Computer

Everything in your computer is represented in binary - not just numbers. How?

Data in our Computer

Everything in your computer is represented in binary - not just numbers. How?

Simple answer - beauty is in the eye of the beholder.

Data in our Computer

Everything in your computer is represented in binary - not just numbers. How?

Simple answer - beauty is in the eye of the beholder.

We can make up any mapping we desire of collections of bits to any other data. Then our programs reading those bits can choose to interpret them that way.

Integers in our Computer - Signed Binary

What about negative integers? If we were writing a binary number on paper we could just slap a negative sign in front of it, e.g. -1011_2

We don't have such a luxury inside our computers. So what can we do? Remember our integers are fixed size - let's work with 8-bit integers for now

Remember - beauty is in the eye of the beholder

Integers in our Computer - Signed Binary

What about negative integers? If we were writing a binary number on paper we could just slap a negative sign in front of it, e.g. -1011_2

We don't have such a luxury inside our computers. So what can we do? Remember our integers are fixed size - let's work with 8-bit integers for now

Remember - beauty is in the eye of the beholder

First Idea: stop using the left-most bit to represent part of the value of our number, instead let the left-most bit be a *sign-bit* where a 1 in that location signifies that the number is negative, a 0 in that location means it is a positive number.

First Attempt Signed Binary

If we use the left-most bit to represent sign instead of part of our value then some numbers in our signed 8-bit binary look like so:

$$\begin{aligned} 1_{10} &= 0000\ \ 0001 \\ -1_{10} &= 1000\ \ 0001 \\ 17_{10} &= 0001\ \ 0001 \\ -17_{10} &= 1001\ \ 0001 \end{aligned}$$

First Attempt Signed Binary - Problem 1

If our only change to implement signed binary is using the left-most bit to represent sign, we have a couple problems.

First Attempt Signed Binary - Problem 1

If our only change to implement signed binary is using the left-most bit to represent sign, we have a couple problems.

First, what do the following two numbers represent?

0000 0000

1000 0000

First Attempt Signed Binary - Problem 1

If our only change to implement signed binary is using the left-most bit to represent sign, we have a couple problems.

First, what do the following two numbers represent?

0000 0000

1000 0000

The represent 0_{10} and -0_{10} — not ideal

First Attempt Signed Binary - Problem 2

We have a worse problem... consider the following addition with our representation:

$$\begin{array}{rr} & 1000\ \ 0001 \\ + & 0000\ \ 0001 \\ \hline & 1000\ \ 0010 \end{array}$$

First Attempt Signed Binary - Problem 2

We have a worse problem... consider the following addition with our representation:

$$\begin{array}{r} 1000\ \ 0001 \\ +\ \ 0000\ \ 0001 \\ \hline 1000\ \ 0010 \end{array}$$

This shows $-1 + 1 = -2$ which is a *big* problem.

Second Attempt Signed Binary - Ones' Complement

When the sign bit is not set, treat the number as a normal binary integer.

To make a positive number into its negative equivalent, set the sign bit and flip the rest of the bits. This representation is called *Ones' Complement.*

$$\begin{aligned} 1_{10} &= 0000\ \ 0001 \\ -1_{10} &= 1111\ \ 1110 \\ 17_{10} &= 0001\ \ 0001 \\ -17_{10} &= 1110\ \ 1110 \end{aligned}$$

Second Attempt Signed Binary - Ones' Complement

When the sign bit is not set, treat the number as a normal binary integer.

To make a positive number into its negative equivalent, set the sign bit and flip the rest of the bits. This representation is called *Ones' Complement.*

$$\begin{aligned}
1_{10} &= 0000\ \ 0001 \\
-1_{10} &= 1111\ \ 1110 \\
17_{10} &= 0001\ \ 0001 \\
-17_{10} &= 1110\ \ 1110
\end{aligned}$$

Addition works, still has the double zero problem.

Signed Binary in our Computers - Two's Complement

When the sign bit is not set, treat the number as a normal binary number.

To make a positive number into its negative equivalent, set the sign bit, flip the rest of the bits, **and** add one to it. This representation is called *Twos' Complement.*

$$\begin{aligned} 1_{10} &= 0000\ \ 0001 \\ -1_{10} &= 1111\ \ 1111 \\ 17_{10} &= 0001\ \ 0001 \\ -17_{10} &= 1110\ \ 1111 \end{aligned}$$

Signed Binary in our Computers - Two's Complement

When the sign bit is not set, treat the number as a normal binary number.

To make a positive number into its negative equivalent, set the sign bit, flip the rest of the bits, **and** add one to it. This representation is called *Twos' Complement.*

$$\begin{aligned} 1_{10} &= 0000\ \ 0001 \\ -1_{10} &= 1111\ \ 1111 \\ 17_{10} &= 0001\ \ 0001 \\ -17_{10} &= 1110\ \ 1111 \end{aligned}$$

Addition works, doesn't have two zero problem. This is how our computers store signed integers.

Numbers in C

The C standard does not specify what size type `int` must be. Most implementations have a 32-bit default `int` type.

Numbers, by default, in C are signed. For integers that means they're represented using 32-bit twos' complement by default. You can specified the signedness of your types with the type modifiers `signed` and `unsigned`.

```c
int x; // signed 32-bit int
signed int y; // signed 32-bit int
unsigned int z; // unsigned 32-bit int
```

Characters in C

Characters in C are also just numbers. All our built-in types are. Type `char` is one byte long. The encoding used to map numbers to displayable characters is *ASCII*

Character literals in C are denoted with single quotes. Unlike Python single quotes cannot be used for string literals, only for individual characters.

```c
#include <stdio.h>
int main(int argc, char ** argv) {
  char c1 = 'H';
  char c2 = 101;
  char c3 = c1+36;
  char c4 = 'z' - 11;
  printf("%c%c%c%c%c\n", c1, c2, c3, c3, c4);
}
```

Table of Contents

Formatted Strings

Unlike Python's `print` function, C's `printf` function cannot print out variables of any type. However the string passed as `printf's` first argument can have *formatting specifiers* in it, subsequently provided arguments will be placed instead of those specifiers.
Some specifiers:

- `%d` — for *decimal (base 10)* integer values
- `%f` — for floating point values
- `%c` — for character values
- `%s` — for string values (???)

Formatted Strings - Example

```c
#include <stdio.h>

int main(int argc, char **argv) {
  int x = 10;
  float z = 25.537;
  char c = '^';
  printf("x: %d\nz: %f\nc: %c\n", x, z, c);
  printf("Two significant digits z: %.2f\n", z);
}
```

Table of Contents

Conditionals

Like Python, C uses the `if` keyword for conditionals. Syntax is much Python but parentheses must surround the condition expression, and curly braces are used rather than indentation.

```
if (expr) {
  ...
}
```

While Loops

Like Python, C has `while` loops. Again the condition expression must be in parentheses, and curly braces enclose the code to conditionally execute.

```
while (expr) {
  ...
}
```

Condition Expressions

Unlike Python, C does not have `bool` type. You can access a type named `bool` in the `stdbool` library, but it's just another name for `int`.

In C zero is considered `false`, and any non-zero value is considered `true`.

Boolean expressions evaluate to `1` or `0`.

In C the logical and, or, and negation operators are `&&`, `||`, and `!` respectively.

Sample Condition

Here's a useful way to apply knowledge we've already learned:

```
if (c >= 'a' && c <= 'z') {
  c = c - 'a' + 'A';
}
```

Knowledge Check: What does `c >= 'a' && c <= 'z'` check for us?
What does `c = c - 'a' + 'A'` accomplish? What did we need to remember to write that?

For Loops

For loops in C are very different than those in Python, which we'll call range-based for loops.

Range-based for loops do not exist in C, instead for loops are like while loops, except with a component where we may initialize our loop variable(s), and a component where we may place our update statements to take place at the end of each iteration.

```
for (init_stmts; cond; update_stmts) {
  ...
}
```

For Loop Format

```
for (init_stmts; cond; update_stmts) { ... }
```

- `init_stmts` — statements separated by commas, run once before your loop begins. Use for declaring, or setting, variables for your loop.
- `cond` — Your loop condition, evaluated before entering loop body each iteration.
- `update_stmts` — statements, separated by commas, run after the loop body each iteration. Use for updating your loop variables.

Sample For Loops

```
for (int i = 0; i < 10; ++i) {
  printf("%d\n", i);
}

  int x;
  int y;
  for (x=0,y=0; x*y<100; x=x+2, y=y+3) {
    printf("(%d, %d)\n", x, y);
  }
```

Code blocks and scope

Curly braces in C form a *scope*. Variables defined within curly braces are only accessible within the scope they were defined in or scopes nested within.

```c
int main(int argc, char ** argv) {
  int x = 5;
  {
    int y = 10;
    {
      int z = y * x;
    } // z no longer exists!
  } // y no longer exists!
  printf("%d\n", y); // Won't compile!
}
```

Table of Contents

Writing and Using Functions

We've already seen how we defined main — other functions are defined similarly.

Here is a function that has two `int` parameters and returns an `int`.

```c
int gcd(int a, int b) {
  while (b != 0) {
    int tmp = a%b;
    a = b;
    b = tmp;
  }
  return a;
}
```

Function calls

Syntax for calling functions is almost exactly the same as in Python.

```c
int main() {
  printf("gcd(16,8)=%d\n", gcd(16,8));
  printf("gcd(153,96)=%d\n", gcd(153,96));
}
```

Practice

Practice Problem: Write a function `print_binary` that has one `unsigned int` parameter and prints out the binary representation of that number.

Functions and Scope

Consider the following program in Python

```python
def times2(x):
  x = x*2

y = 10
times2(y)
print(y)
```

What prints? Why?

Functions and Scope

Consider an analagous program in C

```c
void times2(x) {
  x = x*2;
}

int main(int argc, char **argv) {
  int y = 10;
  times2(y);
  printf("%d\n", y);
}
```

The output is the same — but let's dig deeper as to why.

Table of Contents

Memory - why do we care?

The entirety of your program, and data it uses, must be loaded into RAM for it to be able to execute on your computer.

In C we'll need to understand the basics of how our programs memory is laid out in order to understand certain concepts and behaviours of the language.

Memory - What is it?

Memory is just a bunch of “boxes” which can store values. Each box is a byte, and we can store values therein.

Memory - What is it?

Memory is just a bunch of “boxes” which can store values. Each box is a byte, and we can store values therein.

Your physical RAM receives *addresses* — these are how individual bytes are referenced in your RAM when the CPU asks to load/store data to/from RAM.

Memory - What is it?

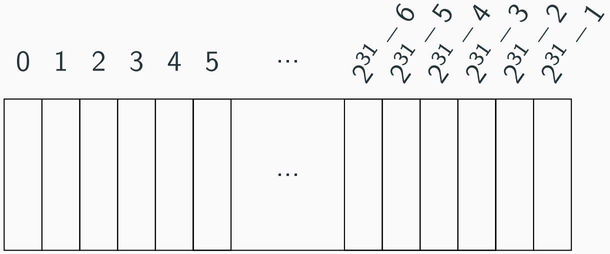
Memory is just a bunch of “boxes” which can store values. Each box is a byte, and we can store values therein.

Your physical RAM receives *addresses* — these are how individual bytes are referenced in your RAM when the CPU asks to load/store data to/from RAM.

Typically RAM is grouped into *words* which are groups of sequential bytes. Typical word sizes are 4 and 8 bytes (32 and 64 bits).

Memory - Addresses

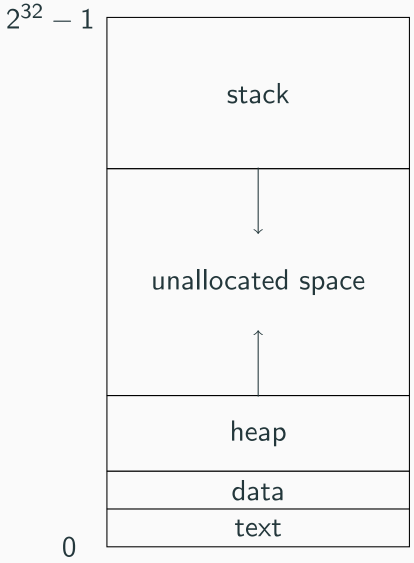
Each byte has an address, addresses start at 0 and go up by one each sequential byte.



Memory - What your program sees

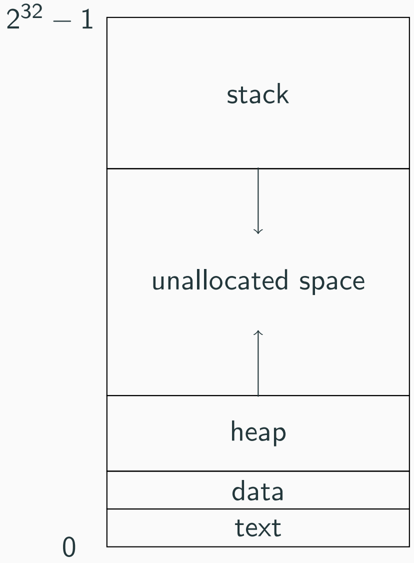
Your program never sees real addresses — the operating system lets your program pretend it has all of memory to itself. As far as your program is concerned its memory starts at address 0 and continues on.

Your program's memory is divided into four segments



Memory Segments

- text — where the code of your program is stored
- data — where global and static data is stored
- heap — where dynamically allocated data is stored
- stack — where statically allocated data is stored



data, heap, and stack expanded

Perhaps a more understandable description of the data stored in these segments is:

- data — stores data whose lifetime that is the entirety of your program. **Known** at compile time.
- heap — stores data whose lifetime is dictated by the programmer. **Not known** at compile time.
- stack — stores data whose lifetime is tied to their enclosing scope. **Known** at compile time.

So far we've only discussed data with lifetimes tied to their scope.

Function calls and the stack

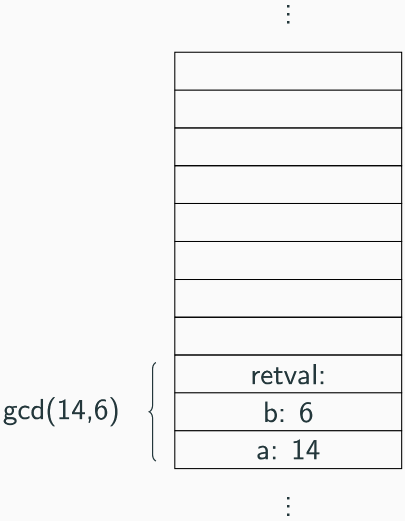
When a function is called a portion of the stack is allocated to that function call. This portion of the stack is called a *stackframe*.

Space for parameters and local variables of that function call is allocated in the functions stackframe. When the function returns the stackframe is deallocated.

Important: This means each call of a function gets its own memory — all of its data is its own. It also means that data goes away when the function ends!

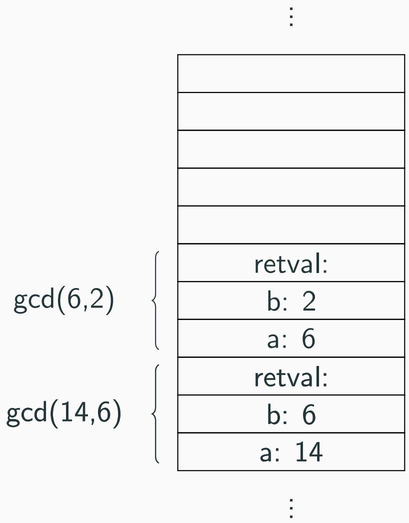
gcd - recursive

```
int gcd(int a, int b) {
  if (b == 0) return a;
  retval = gcd(b, a%b);
  return retval;
}


int main() {
  gcd(14, 6);
}
```



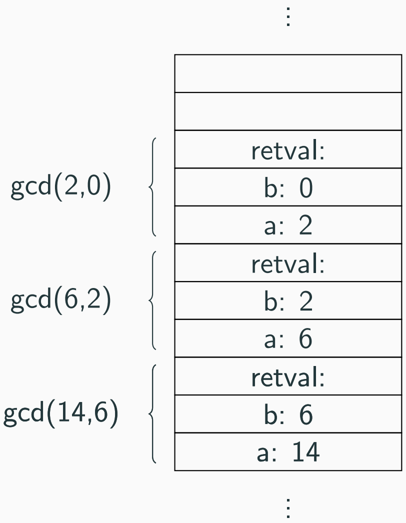
gcd - recursive

```
int gcd(int a, int b) {
  if (b == 0) return a;
  retval = gcd(b, a%b);
  return retval;
}


int main() {
  gcd(14, 6);
}
```



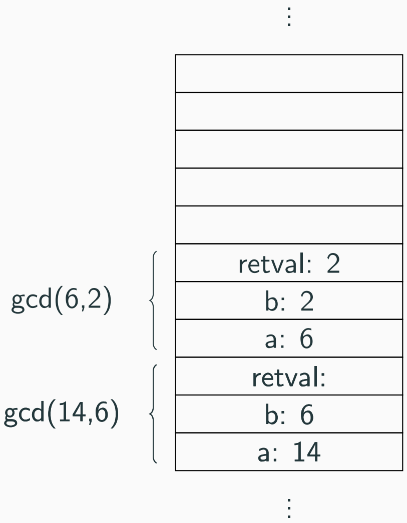
gcd - recursive

```c
int gcd(int a, int b) {
  if (b == 0) return a;
  retval = gcd(b, a%b);
  return retval;
}


int main() {
  gcd(14, 6);
}
```



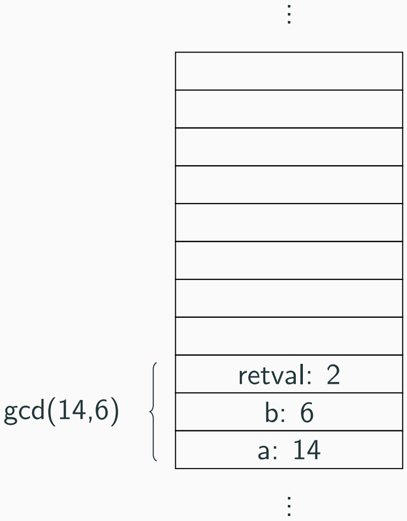
gcd - recursive

```
int gcd(int a, int b) {
  if (b == 0) return a;
  retval = gcd(b, a%b);
  return retval;
}


int main() {
  gcd(14, 6);
}
```



gcd - recursive

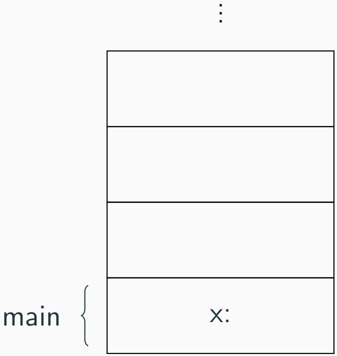
```
int gcd(int a, int b) {
  if (b == 0) return a;
  retval = gcd(b, a%b);
  return retval;
}


int main() {
  gcd(14, 6);
}
```



Mutation - revisited

Now let's ask ourselves again why this program prints 10.

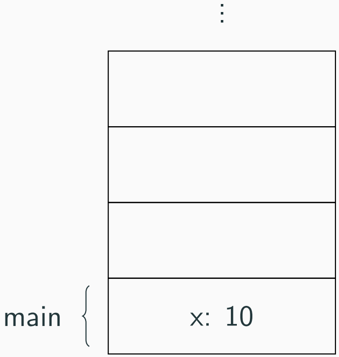
```c
void times2(int x) {
  x = x*2;
}

int main() { <----
  int x = 10;
  times2(x);
  printf("%d\n", x);
}
```



Mutation - revisited

Now let's ask ourselves again why this program prints 10.

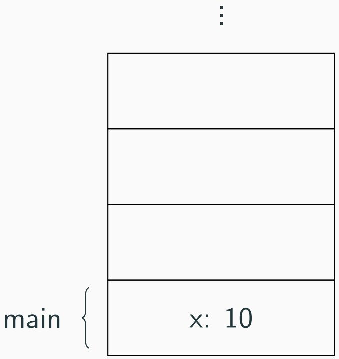
```c
void times2(int x) {
  x = x*2;
}

int main() {
  int x = 10; <----
  times2(x);
  printf("%d\n", x);
}
```



Mutation - revisited

Now let's ask ourselves again
why this program prints 10.

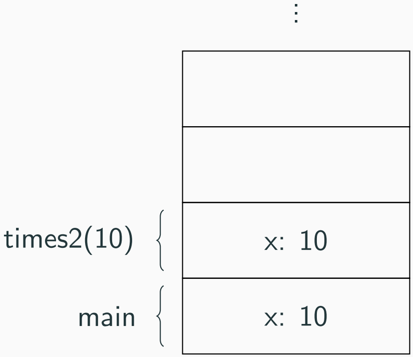
```c
void times2(int x) {
  x = x*2;
}

int main() {
  int x = 10;
  times2(x); <----
  printf("%d\n", x);
}
```



Mutation - revisited

Now let's ask ourselves again
why this program prints 10.

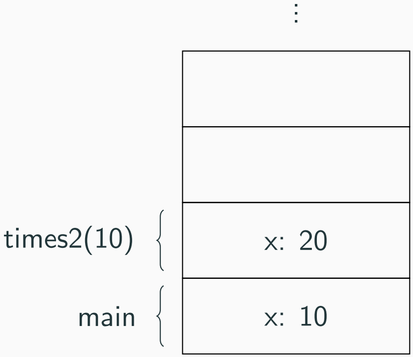
```c
void times2(int x) { <----
  x = x*2;
}

int main() {
  int x = 10;
  times2(x);
  printf("%d\n", x);
}
```



Mutation - revisited

Now let's ask ourselves again
why this program prints 10.

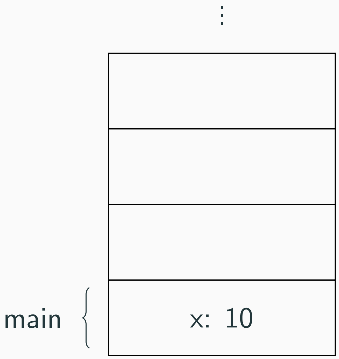
```c
void times2(int x) {
  x = x*2; <----
}

int main() {
  int x = 10;
  times2(x);
  printf("%d\n", x);
}
```



Mutation - revisited

Now let's ask ourselves again
why this program prints 10.

```c
void times2(int x) {
  x = x*2;
}

int main() {
  int x = 10;
  times2(x);
  printf("%d\n", x); <----
}
```



Functions - local variables

Now we should have a good understanding of why changes to our parameter cannot affect the argument that was supplied when the function was called.

The value of the argument is copied into the functions stackframe as the value of the parameter. It is a local copy of the value, stored at a completely different address than the argument.

But what if we wanted to be able to change the argument?