Part 4

Coding in C

Contents

4.1 Background

Q: What is C ?
A: A concise programming language that is designed to run on any device with a central processing unit. C programs run on everything from embedded microprocessor controllers in machinery, network routers, and even is lurking in source code of many Math libraries and tools.

Q: Why learn C ?
A: Once you understand C then the derived languages are simple to pick up. Derived languages include Java, C++, C#, Objective C, and Python. For a more comprehensive list see C family wiki.

Q: Why use C ?
A: Pointers. These are a programming device that give the programmer direct access to data in memory and this will be critical in building high performance codes.

Q: No really, why use C?
A: Later in this course we will start parallel programming using the Message Passing Interface (MPI). This is primarily a C based library for sending messages between multiple instances of a running program. There are Java bindings for MPI here. However, you would end up programming Java in a C like fashion anyway and may miss some important details.

Q: Seriously, why C ?
A: Performance, performance, performance. As a lower level language that is not strongly sandboxed it is possible to squeeze maximum performance when using C.

Q: Why did we have to learn Python ?
A: Good question.

Q: All the cool programmers are using [INSERT FASHIONABLE PROGRAMMING LANGUAGE]

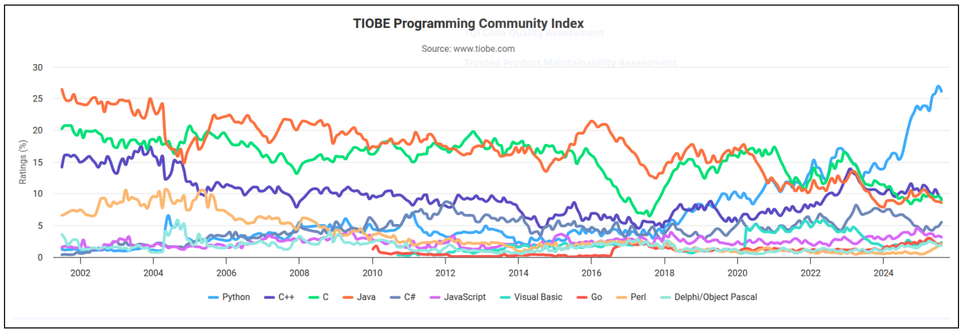


Figure 4.1: Tiobe programming index chart as of 09/19/2025. See: http://tiobe.com for the latest index charts

A: Actually, check out the Tiobe programming language index in Figure 4.1. The folks at Tiobe track the popularity of different programming languages over time. As of 09/19/25, C was number two in their chart and seems to be catching up with Java.

Q: What happened to C++ ?
A: C++ is an incredibly sophisticated and powerful programming language that has many useful applications. However, over time it has become progressively more complicated to understand the whole language and more difficult to develop and manage large C++ projects. This may have contributed to the decline in popularity.

4.2 Installing the GNU compiler collection

To install the GNU C compiler use the following package from the Terminal

```
sudo apt install gcc
```

4.3 Hello world, C edition

Writing a “hello world” program is a ritual that programmers perform when picking up a new programming language to help familiarize themselves with new syntax and general basics. It is so well established in programming practice that there are even web pages that catalog hello world programs in almost every known programming language. See http://helloworldcollection.de/ for a very comprehensive collection of hello world programs in a large number of different languages and their variations.

So without any more ado, the following program will print a Hello world message on the terminal.

```c
/* include standard input-output library header */
#include <stdio.h>

/* every C program has a main function */
int main(int argc, char **argv){

  /* the body of the main function is all commands between {} */

  /* call a function to print a message to standard out */
  printf("Hello world\n");

  /* return 0 from this program */
  return 0;
}
```

Listing 1: Hello world example in C

It encapsulates some elementary syntax that we highlight in this list:

- `/* ... */` is a comment that should be ignored when the source code is converted into a program. Anything between the `*` symbols is ignored, but can be used by the programmer to leave a message to future readers of the code to help understand what the code does.
- `#include <stdio.h>` is not actually C code! It is in fact an instruction to the C preprocessor that initially processes the source code. It instructs the preprocessor to include the `stdio.h` header file into the program it is building from the source code. The header file includes information

about the call signature for functions in the standard input-output library. We need this because of the `printf` statement later in the code.

- `int main(int argc, char **argv)` describes the input and output signature of the `main` function. This function has a special status since every C program must include one, and only one, `main` function. When the OS executes this program it will give you the programmer control of the execution of instructions by "entering" the `main` function. We will dwell on what the notation around `main` means in later lectures.
- `{ ... }` subtly surrounds everything that happens in the body of the `main` function. These brace characters are critical as they delimit all instructions associated with the `main` function. Fail to wrap the contents of a functions with { ... } and confusion will ensue.
- `printf("Hello World\n")` is an instruction to "call" the `printf` function and "pass" in a string of characters that should be output to the terminal. [To be precise, a piece of memory is reserved for a string literal and the location in memory of the first character of the string is passed to the `printf` function as a pointer which we will discuss in detail later]. A curious fact about the `printf` function is that it is not part of the core C language because not every device that can run C actually has the standard output for printing text to a terminal. That is why we had to include the standard input-output header that describes the `printf` function signature. When compile the program later, the compiler will automatically add in the definition of the `printf` function as it builds the executable.
- `;` lurks after the `printf` function call. I bet you almost didn't see it ! The semi-colon plays a critical role in C because in general C is rather indifferent to white space so a command could possibly be spread over multiple lines. The semi-colon is used to terminate a statement. If you neglect to terminate a statement with a semi-colon then all manner of confusion may ensue.
- `return 0;` at the end of the function returns the zero numerical value to the terminal. In general functions can return a value to their call site, and in this case the terminal is the call site.

In that one program we have revealed two key aspects of C syntax that bare repeating: statements are terminated with semi-colons and the contents of a function are wrapped with braces. Forgetting either of these things will yield weird compiler errors.

Which brings us to the issue of how to create the file and how to execute the hello world program. Fortunately we can use `emacs` to edit a text file.

Note #1: you should call the file `helloWorld.c` and that the `.c` file extension is important !

Note #2: avoid writing all your files to your Ubuntu home directory. I recommend creating a directory for each lecture

Note #3: remember that you can launch `emacs` to create the file.

Putting these together you can get started with

```
# make a directory for this lecture
mkdir ~/course/L04

# change to that directory
cd ~/course/L04
```

```
# launch emacs to edit the file, doing so in the background using ampersand
emacs helloWorld.c &
```

I snuck in an extra trick when you launch `emacs`. That ampersand `&` character tells the Terminal that you want to run the `emacs` command in the background. This is okay since `emacs` will run in a separate window. The `emacs` window should appear and you should get a new prompt in the Terminal window.

Go ahead and type in the hello world program. You can save by clicking on the icon of a disk+arrow+Save.

Go back to the terminal and use `ls` to verify that the source file was actually saved.

At this point you have only created the source file, you have not created an executable program. It is necessary to use a compiler to convert the ASCII source code (`helloWorld.c`) into a binary executable that the OS can understand. For this class we will use the C compiler from the GNU compiler collection. This may not be installed by default so you should go ahead and make sure that that it is installed using `apt`

```
sudo apt update
sudo apt install gcc
```

Now you should be all set to compile the hello world program. You can invoke the compiler with

```
gcc -o helloWorld helloWorld.c
```

The `-o helloWorld` option instructs the compiler to output the compiled program to a file named helloWorld. The last argument is just the name of the C source file you want to compile.

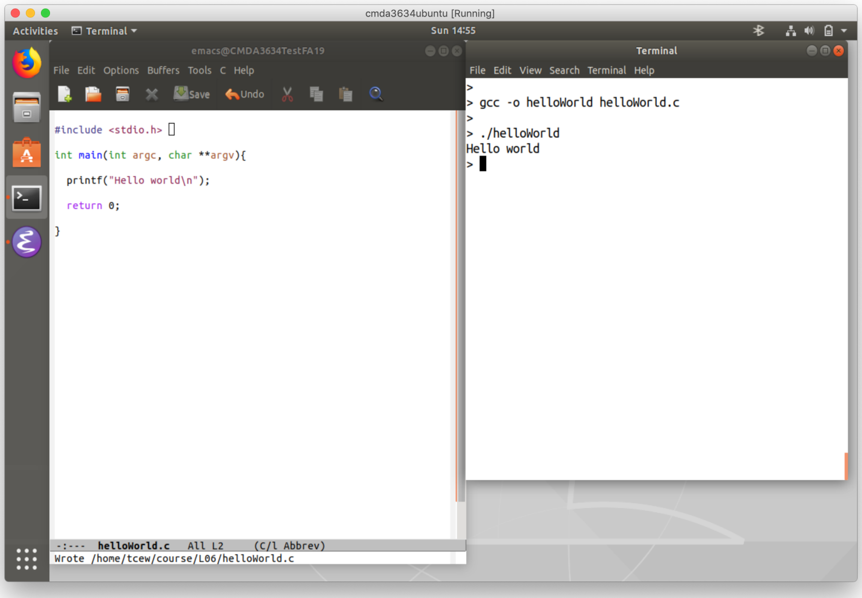
Assuming that you did not get any error messages from the `gcc` compiler you can run your program with

```
./helloWorld
```

Note #1: the Terminal is case sensitive. Note the W is capitalized (assuming you called your program helloWorld)

Note #2: we added a `./` to the executable name because the Terminal needs to know where the executable is located. In general `./` on its own is short hand for the present working directory.

If you got this far then congratulations you have literally added new functionality to your Ubuntu Terminal !



```c
#include <stdio.h>

int main(int argc, char **argv){

  printf("Hello world\n");

  return 0;

}
```

```
-:---  helloWorld.c   All L2    (C/l Abbrev)
Wrote /home/tcew/course/L06/helloWorld.c
```

```
>
> gcc -o helloWorld helloWorld.c
>
> ./helloWorld
Hello world
>
```

Figure 4.2: Output when you successfully run the hello world program.

4.4 Note on terminology

We will start throwing terms around like bit, byte, kilobyte, megabyte, . . . when referring to quantities of data. The smallest quantity of data is one bit. A bit of data can take the value zero or the value one. There are four bits in one nibble [1]. There are eight bits in a "byte". The term byte is very commonly used. For instance in C the smallest variable size that can be used is one byte used to represent a `char` character variable. The byte became commonly used as a base data size when the first 8-bit computers were released and adopted for home computing in the late 70s and early 80s.

The following table summarizes commonly used terms for data quantities. Notice the progression in powers of a thousand.

Number of bytes	Notation	Name	Example size
1	B	byte	text character
1000	kB	kilobyte	e-mail
1000^2	MB	megabyte	image
1000^3	GB	gigabyte	movie
1000^4	TB	terabyte	hard drive
1000^5	PB	petabyte	server storage
1000^6	EB	exabyte	human words spoken
1000^7	ZB	zettabyte	?
1000^8	YB	yottabyte	?

Table 4.1: Base 10 data quantity naming convention (see wiki for more info).

[1] You will hear the term "bit" a lot. The "nibble" (four bits) is used much less frequently.

There is a nice article discussing these quantities here.

4.5 Variables

`Variable: a name that we attach to a value stored in computer memory.`

4.5.1 Integer variable type: `int`

This C command instructs the compiler to set aside a certain number of bits of memory that we can access through a variable named `i`

```c
int i;
```

Note #1: because C is a strongly typed language we have to associate a type to the variable when it is created, in this case "`int`"

Note #2: `int` is shorthand for a finite integer represented by a number with (typically) 32 (binary) bits.

Note #3: the C language specification does not describe what the initial value of a variable will be when it is created. We cannot assume for instance that it will be set to zero when created.

For instance, if we decide to assign a starting value of 3 to the variable we must explicitly do so explicitly with

```c
int i = 3;
```

or

```c
int i; /* create variable named i */
i=3;   /* assign value 3 to i */
```

In brief the integer `int` type is encoded into (typically) 32 bits. Echoing decimal representation the 32 bits are presented with sign bit leftmost, and the actual binary digits running left to right with the most significant digit to the left. Some examples

```
0 = 00000000000000000000000000000000
```

```
         2 = 00000000000000000000000000000010
        10 = 00000000000000000000000000001010
        42 = 00000000000000000000000000101010
       170 = 00000000000000000000000010101010
       682 = 00000000000000000000001010101010
      2730 = 00000000000000000000101010101010
     10922 = 00000000000000000010101010101010
     43690 = 00000000000000001010101010101010
    174762 = 00000000000000101010101010101010
    699050 = 00000000000010101010101010101010
   2796202 = 00000000001010101010101010101010
  11184810 = 00000000101010101010101010101010
  44739242 = 00000010101010101010101010101010
 178956970 = 00001010101010101010101010101010
 715827882 = 00101010101010101010101010101010
```

where I have added 2^{2n} to a running accumulator for $n = 1 : 15$. Here is the sequence where I subtracted instead of adding

```
         0 = 00000000000000000000000000000000
        -2 = 11111111111111111111111111111110
       -10 = 11111111111111111111111111110110
       -42 = 11111111111111111111111111010110
      -170 = 11111111111111111111111101010110
      -682 = 11111111111111111111110101010110
     -2730 = 11111111111111111111010101010110
    -10922 = 11111111111111111101010101010110
    -43690 = 11111111111111110101010101010110
   -174762 = 11111111111111010101010101010110
   -699050 = 11111111111101010101010101010110
  -2796202 = 11111111110101010101010101010110
 -11184810 = 11111111010101010101010101010110
 -44739242 = 11111101010101010101010101010110
-178956970 = 11110101010101010101010101010110
-715827882 = 11010101010101010101010101010110
```

There is a fun number base converter here and discussion of integer representation with two's complement encoding here.

In brief: given an N digit two's complement representation $\{b_n\}_{n=0}^{n=N-1}$ then we can reconstruct the original integer a with

$$a = -b_{N-1}2^{N-1} + \sum_{n=0}^{n=N-2} b_n 2^n.$$

We give the formula here for completeness and it is important for fine grain understanding of integer representation. However, if it doesn't quite gel with your understanding of integers and how they might be stored you can come back to this later.

Note: keep in mind that the C standard does not actually specify how many bits should be used to represent an `int`. This harks back to the day when the natural word length for an integer was 16 bits, because that is what was supported in computers of the time equipped with 16 bit hardware registers [2] and 16 bit data pathways. These days it is highly likely but not absolutely certain that an `int` will be represented with 32 bits reflecting common register size and data pathways of the 1990s. You can actually find out for a given compiler on a given system how many bytes (groups of 8 bits) are used to represent an `int` using the `sizeof` macro demonstrated here

```
printf("An int takes %d bytes which is %d bits\n", sizeof(int), 8*sizeof(int));
```

Arithmetic integer operations

C can perform the usual arithmetic operations that you might expect including addition, subtraction, multiplication, and division.

Note: keep in mind that `int` objects are on a finite subset of the integers so the arithmetic operations can overflow and generate say negative numbers when you add two large numbers.

Here is a sequence of C programs that perform a range of arithmetic operations. What do you think the output of these programs should be ?

Q1 [yes it is a trap]

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

  int a, b, c; /* create some integers */

  c = a+b;      /* perform integer addition */

  printf("a = %d, b = %d, c = %d\n", a, b, c); /* print result */

  return 0;
}
```

Q2 [not a trap]

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

  int a = 1, b = 2, c; /* create some integers */

```

[2] a register is an on-chip physical data store

```c
  c = a+b;     /* perform integer addition */

  printf("a = %d, b = %d, c = %d\n", a, b, c); /* print result */

  return 0;
}
```

Q3 [small trap]

```c
int main(int argc, char **argv){

  int a = 1, b = 2, c; /* create some integers */

  c = a/b;     /* perform integer division */

  printf("a = %d, b = %d, c = %d\n", a, b, c); /* print result */

  return 0;
}
```

Q4 [nasty trap]

```c
int main(int argc, char **argv){

  int a = 1, b = 0, c; /* create some integers */

  c = a/b;     /* perform integer division */

  printf("a = %d, b = %d, c = %d\n", a, b, c); /* print result */

  return 0;
}
```

Answers on next page.

A1: this is sneaky, we did not initialize the `a` and `b` variables before performing the addition operation. Their values will be the value of whatever variables occupied their bits previously. So we cannot say with certainty what the **Q1** code will output.

A2: nothing to see here. The code will output `a=1,b=2,c=3`.

A3: when the arguments to the division operator are both integers as is the case here then integer division is performed (see this wiki). In brief when performing integer division the fractional part of the result is discarded. In this case the answer will be zero.

A4: this is an example of division by zero. Behavior is not strictly determined by the C standard. It is likely that the code will issue an exception when the division by zero happens. Careful code will try to catch such errant behavior. Bottom line: try to avoid division by zero.

For fun check out this video of a mechanical calculator trying to divide by zero: video.

4.5.2 Character variable type: `char`

In the early days of C the contemporary computers typically had at least 8 bit registers and 8 bit data pathways. Thus the smallest data size that C supports is 8 bits, also called a byte. There is an 8 bit variable type in C called a `char` which is short for character. Variables of type `char` are typically used to represent letters, numbers, and symbols in text.

In the previous section we saw that integers are represented by a two's complement bitwise encoding. The analogue for 8-bit character encoding is actually a look up table defined by the ASCII standard. An 8-bit `char` can represent numbers in binary in the range 0:255. Characters in the ASCII set are mapped to numbers in this range. For instance, the decimal numbers are mapped to the range 48:57. Capital letters are mapped to values in the range 65:90. Lower case letters map into the range 97:122. Space is 32. The whole ASCII character set is described here https://www.ascii-code.com/. Other descriptions are available via your favorite search engine.

Exercise: encode "The quick brown fox Jumped over the lazy dog 5 times!" into ASCII by hand. Reference answer on the next page.

Exercise solution: Being lazy I coded up a program to automatically print out the ASCII encoding of a string. The following C code outputs the ASCII encoding of a string literal

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

void asciiPrint(char *message){
  int n=0;
  while(1){ /* keep iterating the following code */

    printf("char %c => ascii %d\n", message[n], (int)message[n]);

    if(message[n]=='\0'){ break; } /* break if string terminator */

    n=n+1; /* increment counter */
  }
}

int main(int argc, char **argv){

  asciiPrint("The quick brown fox Jumped over the lazy dog 5 times!");
  return 0;
}
```

The string literal translation for character to ASCII code is shown here[3]

```
char T => ascii 84      char n => ascii 110     char v => ascii 118     char o => ascii 111
char h => ascii 104     char   => ascii 32      char e => ascii 101     char g => ascii 103
char e => ascii 101     char f => ascii 102     char r => ascii 114     char   => ascii 32
char   => ascii 32      char o => ascii 111     char   => ascii 32      char 5 => ascii 53
char q => ascii 113     char x => ascii 120     char t => ascii 116     char   => ascii 32
char u => ascii 117     char   => ascii 32      char h => ascii 104     char t => ascii 116
char i => ascii 105     char J => ascii 106     char e => ascii 101     char i => ascii 105
char c => ascii 99      char u => ascii 117     char   => ascii 32      char m => ascii 109
char k => ascii 107     char m => ascii 109     char l => ascii 108     char e => ascii 101
char   => ascii 32      char p => ascii 112     char a => ascii 97      char s => ascii 115
char b => ascii 98      char e => ascii 101     char z => ascii 122     char ! => ascii 33
char r => ascii 114     char d => ascii 100     char y => ascii 121     char  => ascii 0
char o => ascii 111     char   => ascii 32      char   => ascii 32
char w => ascii 119     char o => ascii 111     char d => ascii 100
```

Note #1: even the spaces get encoded and there is also a sneaky zero ASCII code at the end of string literal that terminates the string and is invisible in the original string. This extra character even trips up seasoned C programmers.

Note #2: there is a lot going on in the above code that we used to print out the ASCII encoding. We will see more of the constructs used in that code later in the course.

Anthony Austin pointed out that the Terminal has a built in man page for whole ASCII table which you can read with

[3] with one deliberate error, can you spot it ?

```
man ascii
```

and it outputs the table in several bases

```
ASCII(7)             BSD Miscellaneous Information Manual             ASCII(7)

NAME
     ascii -- octal, hexadecimal and decimal ASCII character sets
...
       0 nul    1 soh    2 stx    3 etx    4 eot    5 enq    6 ack    7 bel
       8 bs     9 ht    10 nl    11 vt    12 np    13 cr    14 so    15 si
      16 dle   17 dc1   18 dc2   19 dc3   20 dc4   21 nak   22 syn   23 etb
      24 can   25 em    26 sub   27 esc   28 fs    29 gs    30 rs    31 us
      32 sp    33  !    34  "    35  #    36  $    37  %    38  &    39  '
      40  (    41  )    42  *    43  +    44  ,    45  -    46  .    47  /
      48  0    49  1    50  2    51  3    52  4    53  5    54  6    55  7
      56  8    57  9    58  :    59  ;    60  <    61  =    62  >    63  ?
      64  @    65  A    66  B    67  C    68  D    69  E    70  F    71  G
      72  H    73  I    74  J    75  K    76  L    77  M    78  N    79  O
      80  P    81  Q    82  R    83  S    84  T    85  U    86  V    87  W
      88  X    89  Y    90  Z    91  [    92  \    93  ]    94  ^    95  _
      96  `    97  a    98  b    99  c   100  d   101  e   102  f   103  g
     104  h   105  i   106  j   107  k   108  l   109  m   110  n   111  o
     112  p   113  q   114  r   115  s   116  t   117  u   118  v   119  w
     120  x   121  y   122  z   123  {   124  |   125  .   126  ~   127 del
```

4.5.3 Floating point number: `float`

We have seen how integers and characters are encoded in C. However, there is still an important class of numbers missing: real numbers. The C language includes support for so-called floating point numbers. These are numbers that are represented in binary with 32 or 64 bits. The floating point representation consists of three parts: the sign bit, the exponent, and the mantissa (also called the fraction).

For a 32-bit floating point variable (of type `float` defined by the IEEE 754 standard: one bit is dedicated to the sign bit, 8 bits to the exponent, and the remaining 23 bits represent the fraction. For a full description of the IEEE 754 standard see wiki. There is a really useful diagram and description of how a real number is encoded into a `float`.

We can apply arithmetic operations to `float` variables. Here are some examples to ponder

Q5 [**small trap**]

```c
int main(int argc, char **argv){
  float a = 1, b = 2, c=3, d, e; /* create some float variables */

  d = a/b;      /* perform floating point division */
  e = a/c;      /* perform floating point division */
  printf("a=%17.15f, b=%17.15f, c=%17.15f\n, d=%17.15f, e=%17.15f \n",
         a, b, c, d, e); /* print result with 17 significant figures */
  return 0;
}
```

Q6 [big trap]

```c
#include <stdio.h>

int main(int argc, char **argv){
  float a = 1, b = 2.e-9, c; /* create some float variables */

  c = a+b;     /* perform floating point addition */

  /* print result with 17 significant figures */
  printf("a = %17.15f, b = %17.15f, c = %17.15f\n", a, b, c);
  return 0;
}
```

Note: we specify the precision and format for printing variable values by modifying the format string passed to `printf`. In this case `%17.15f` says print out the variable in scientific notation with 17 significant digits. Answers on the next page.

A5: `d=a/b` is easy since `d=a/b=1/2` is exactly representable by the floating point encoding. On the other hand `e=a/c=1/3` is not exactly representable by the IEEE 754 encoding. The output of the code is

```
a=1.000000000000000, b=2.000000000000000, c=3.000000000000000,
d=0.500000000000000, e=0.333333343267441
```

Notice how `d` is exactly represented but `e` is only represented to about 7 digits of accuracy. More on that later.

A6: Here we are playing a dangerous game by adding a small number to a one. In the floating point representation the 1 fixes the representation so that the exponent is 127, and then since 1e-9 is smaller than 2^{-23} we are effectively adding zero to 1. The output after `c=a+b` is

```
a = 1.000000000000000, b = 0.000000002000000, c = 1.000000000000000
```

So there you have it, according to C $1 + 2.10^{-9}$ is equal to 1. This single issue is responsible for enumerable problems in numerical computing. I will defer to other classes to cover this in more depth. You can find some cool examples of "Numerical Monsters" where the results of this type of finite precision rounding error manifest in strange ways here.

4.6 The stack

The stack, sometimes called the "Call stack", is a section of system memory that is used to store function variables. In this `main` function we create two integer variables `a` and `b`. Each integer occupies 4 bytes, so 8 bytes at the bottom of the stack will be set aside for these variables

```c
/* L04/stackVariables.c */
int main(int argc, char **argv){
  int a;
  int b;

  a = 1;
  b = 2;

  return 0;
}
```

Immediately after the stack space is reserved the actual value of the two `int` variables is not specified by the C standard. We then go ahead and assign the value 1 to `a` and 2 to `b`.

The stack grows as the number of function variables grows. Consider the following code where `main` calls another function as in this code

```c
/* L04/callStack.c */
int foo(int d, int e){
  int f = d + e;
  return f;
}
int main(int argc, char **argv){
  int a, b, c;

  a = 1;
  b = 2;
  c = foo(a,b);

  return 0;
}
```

In this case two things happen when the call too `foo` is made. First the memory address (sometimes called program counterr) of the call instruction in the `main` function is added to the stack, secondly two new variables are added to the stack. One for each argument passed into `foo` recalling that C uses a pass-by-copy mechanism to pass data into a function. In the context of the call stack this makes a lot of sense since each function gets to use part of the memory stack where its variables are stored.

When the function `foo` finishes executing its variables are removed from the stack and the program counter is set to the next instruction after the stored call site program counter which is also then removed from the stack. [The above is a rough description of how the stack works see wiki for more details.]

The bottom line is that the variables you create in a function live on the stack.

4.7 Stack arrays

Frequently we will need not just a handful of variables but instead a large number of variable values. For instance a data set with say hundreds of data points. We can reserve space on the stack for say a hundred integer variables with this notation

```c
int v[100];
```

This will do two things. First 400 bytes will be reserved on the stack, and secondly a variable `v` will be added to the stack. That variable is a so-called "pointer" variable that in this case will be a 32-bit or 64-bit variable (depending on if you installed the 32-bit or 64-bit Ubuntu OS). The pointer variable represents an unsigned integer offset in memory from the first byte of memory and thus "points" to the start of the array in memory.

We can read or write to individual entries of the stack array with the square bracket operator as follows

```c
/* see L04/stackArray.c */
int v[100];

v[0] = 11; /* set first entry in array to 11 */
v[66] = 5; /* set 67th entry in array to 5 */

int a = v[0]; /* copy first entry from array into a */
```

It is important to keep in mind that a stack array only lives as long as it takes the function it resides in to finish executing. As soon as the function returns the stack array is reclaimed.

4.8 Pointers

The `v` variable in the above example is an example of a pointer, i.e. it contains the memory address of the first entry of the stack array. A pointer is a curiously powerful programming device. In fact it is so powerful that entire languages have been created to avoid exposing pointers.

For instance Java gives access to memory through references, but hides pointers behind the scenes. This is part of the Java security model as it was originally envisioned as a programming language where you could write code once and run anywhere for instance as a web application. Thus to avoid the security issues of exposing pointers Java does not explicitly give programmers access directly to memory.

To illustrate the power of pointers consider this example where we

```c
/* see L04/pointers.c */
int a = 10;     /* create an integer variable on the stack */
int *pt_a = &a; /* use reference operator to get pointer to variable */
pt_a[0] = 6;      /* assign value to a via dereferenced pointer */
```

By using the reference operator (`int *pt_a = &a`) we are able to find the address of `a` in memory and store it in a pointer variable. We then use the `[]` operator to dereference the pointer and set the value of `a` to 6. Here we chose to treat the pointer as the pointer to the first entry in an array whiich contains the variable `a`.

We could have also achieved the same goal with `*pt_a = 6` as the `*` operator in this context is also a dereferencing operator.

Put another way, we can modify a variable as long as we have a pointer to its location in memory. Consider this example

```c
/* L04/pointerFunctionArg.c */
void changeViaPointer(int *pt_a){

  pt_a[0] = 4;

}

int main(int argc, char **argv){

   int a = 5;

   /* pass copy of pointer to a to changeViaPointer */
   changeViaPointer(&a);

   /* print a */
   printf("a=%d\n", a);

   return 0;
}
```

In this case, the `changeViaPointer` function will change the variable `a` via dereferencing the pointer to `a`. This is an example where a pointer is able to extend the scope of a variable from one function to another function. This allows us to circumvent the limitations of local scope imposed by the C argument pass-by-copy convention.

I hope this reveals some of the power of a pointer and why it is often maligned as a dangerous instrument. To clarify some possible misuses consider this code

```c
/* L04/snoop.c */

#include <stdio.h>
void snoop(char *message){
  /* read characters from the stack indefinitely */
  int n = 0;
  while(1){
   printf("%c", message[n]);
   n = n+1;
  }
}

int main(int argc, char **argv){

   char message1[] = "The message we are passing";
   char message2[] = "The message we are not passing";

   /* print out values from the stack starting at the beginning of message1 */
   snoop(message1);

   return 0;
}
```

When we build and run this we get a lot of extra output until the program tries to read from a segment of memory it is not allowed to access and ceases execution with the classic |Segmentation fault (core dumped)— message.

The output from running the `snoop` code includes some interesting information

```
The message we are passing^@9;V^@^@The message we are not passing^@^@ ...
./snoop^@CLUTTER_IM_MODULE=xim^@LS_COLO ...
GNOME_SHELL_SESSION_MODE=ubuntu ...
```

Notice how this simple snoop program has revealed information from the bash shell environment variables as well as other data from the stack.

In our `snoop` code we only used a pointer to read beyond our safe space on the stack. We could have also tried to write beyond the end of the `message1` character array.

Using a pointer to read or write outside the space allocated for an array is referred to as an "out of bounds" error. The `message1` array only has 27 characters including the null terminator. We only should access `message[n]` for `n in [0,27)` however as we have seen C will let us access way beyond the end of the array and the program only stops when the OS intervenes and forces the program to stop running with a seg fault when it detects a gross violation of memory access rights.

You might ask why discuss this ? Wouldn't a programmer lacking malevolent intent avoid intentionally committing an out of bounds array memory access ? It turns out that the key word here is "intentionally". Oftentimes beginning (and even seasoned) C programmers miscalculate arrays indices. A simple mistake is to forget that C is a zero-based language.

This is ok

```c
int v[10];
int i;
/* fill up entries of array with 3 */
for(i=0;i<10;++i){
  v[i] = 3;
}
```

This is NOT ok

```c
int v[10];
int i;
/* fill up entries of array with 3 */
```

```c
for(i=1;i<=10;++i){
  v[i] = 3; /* this causes an out of bounds error */
}
```

because the first entry of the arrays is at `v[0]` and the last entry of this array is at `v[9]`. In the bad code we are accessing entries `v[1]` to `v[10]` and that last entry is not part of the array of 10 `int` variables. Notice that the differences between these two codes are seemingly quite innocuous.

It is so easy to commit mistakes like this that we will introduce strategies to avoid doing so and also later on use the `valgrind` software to automatically detect when a code commits an out of bounds error.

Finally, keep in mind that there is a limit to how much data can be stored on the stack. The stack limit in Ubuntu tends to be measured in megabytes (millions of bytes) and this can be quickly consumed if you create large stack arrays. In the following we discuss a larger section of system memory called the heap.

4.9 Heap arrays

The heap is a separate section of memory that is distinct from the stack. It is common practice to allocate an array on the heap when the array is large and/or the array will be used by multiple functions and will have a relatively long lifetime during the execution of the program.

To allocate an array on the heap we use functions in the C standard library and thus will need to `#include <stdlib.h>`. The following code snippet goes through the process of allocating, populating, and freeing a heap array.

```c
/* L04/heapArray.c */

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){

  /* use N to specify number of entries in array */
  int N = 1000;

  /* allocate storage for N*sizeof(int) bytes on the heap */
  int *v = (int*) malloc(N*sizeof(int));

  /* use a  for loop to set the N entries in the array */
  int n;
  for(n=0;n<N;++n){
    v[n] = n;
  }

  /* free the heap array */
  free(v);
  v = NULL; /* NULL the pointer as a precaution */
}
```

Note at the end, after freeing the heap array, we zeroed out the value of the pointer to avoid having a stale pointer (to a location in heap memory) that could be misused later in the code since the array has been returned to the OS.

The key functions we used are: `malloc` to memory-allocate on the heap and `free` to release the heap memory back to the OS. These are extremely useful functions in the C standard library.

There are some caveats you should keep in mind when allocating heap arrays:

1. If you request an array that is larger than the available space on the heap `malloc` will return a NULL pointer (zero pointer).
2. The `malloc` function reserves space on the heap and provides a pointer to the first entry. However, it does not take any steps to initialize the values in the array. The values in the array will be whatever values are left over from previous data stored in that part of the heap.
3. If you want to guarantee that the all the bits of the array are zeroed you can alternatively use the `calloc` function (short for clear-alloc). It has a slightly different interface: `int *v = (int*) calloc(N, sizeof(int));`
4. The `free` command is a little fragile. Avoid freeing a heap array multiple times.
5. If you do not free a heap array before you lose track of its pointer then the heap memory "leaks". If you keep doing this then you will eventually fill up the heap and subsequent memory allocations will fail.

In the following section we discuss some strategies to avoid some of the commonest pitfalls of programming with heap pointers.

4.10 Defensive programming with pointers

By now I hope you are feeling mildly wary of using pointers given how their abuse can cause the obvious catastrophic failure of a segmentation fault or even worse a silent memory corruption during the execution of your program.

We can try to avoid some of the worst issues by adopting a more defensive approach to using the heap and pointers. One commonly used strategy is to create a new variable struct type for an array. The struct will include enough information that out of bounds array accesses can be detected at runtime by the use of `get` and `set` functions that are the only functions that are allowed to access entries in an array.

The following code implements a very barebones array struct

```c
typedef struct {

  int Ndata; /* number of data entries*/

  int *data; /* pointer to heap data array */
} array_t;
```

and this code implements a simple (but not fool proof) constructor to initialize an `array_t` instance

```c
array_t arrayConstructor(int Ndata){

  array_t a;
  a.Ndata = Ndata;
  a.data  = (int*) calloc(Ndata, sizeof(int));

  if(a.data == NULL) {
    fprintf(stderr, "WARNING: arrayConstructor calloc failed\n");
  }

  return a;
}
```

Note: that the `arrayConstructor` checks to see if `calloc` returned a NULL pointer, i.e. it checks to see if a heap array was actually allocated. If a heap array was not created then the code prints an error message to the standard error stream that is customarily used to print out error messages. In this code we chose not to cease execution of the program if the heap allocation failed, but since this is such a serious failure it may be prudent to do so.

It is now natural to provide `get` and `set` functions that read and write entries to the array respectively. In the next code we give simple implementations that check whether writing to entry `n` is permissible taking into account the number of entries in the array and that the `get` and `set` functions assumed zero indexing.

```c
int arrayGet(array_t a, int n){

  if(n<0 || n>=a.Ndata){
    fprintf(stderr, "WARNING: arrayGet size %d array bounds error with index %d\n",
            a.Ndata, n);
    return 0;
  }

  if(a.data == NULL){
    fprintf(stderr, "WARNING: arrayGet request to access NULL array\n");
    return 0;
  }
  return a.data[n];
}

void arraySet(array_t a, int n, int val){

  if(n<0 || n>=a.Ndata){
    fprintf(stderr, "WARNING: arrayPut size %d array bounds error with index %d\n",
           a.Ndata, n);
    return;
  }

  if(a.data == NULL){
    fprintf(stderr, "WARNING: arrayPut request to access NULL array\n");
    return;
  }

  a.data[n] = val;
}
```

Thus if all `array_t` instances are initialized using the `arrayConstructor` and entries are only accessed through `arrayGet` and `arraySet` then we can be relatively confident that the array will be a lessor source of errors than using a naked pointer.

However, keep in mind that other parts of your code that use naked pointers could still potentially be responsible for memory access errors that impact the member variables of your `array_t` struct and also the contents of the data array you have wrapped with the `array_t` struct.

The final minimal barebones `array_t` needs a destructor function that frees the heap array and sets the number of entries to zero. As usual we add some error checking to make sure that we don't try to free a NULL pointer.

```c
array_t arrayDestructor(array_t a){

  if(a.data == NULL){
    fprintf(stderr, "WARNING: arrayDestructor trying to destruct NULL array\n");
  }
  else{
    free(a.data);
  }

  // zero length of array
  a.Ndata = 0;

  return a;
}
```

We combine these function calls in the following example code to create an array, set a legitimate entry, try to set an illegitimate entry, get an entry, and finally destruct the array.

```c
int main(int argc, char **argv){

  int Na = 10;

  /* build array */
  array_t a = arrayConstructor(Na);

  arraySet(a, 3,  1); /* a[3] = 1 */
  arraySet(a, 11, 2); /* a[11] = 2 will get WARNING*/

  int a3 = arrayGet(a, 3); /* set a3 = a[3] */

  /* destroy array */
  a = arrayDestructor(a);

  return 0;
}
```

This more defensive style of programming hopefully has some apparently positive side benefits.

1. The code in `main` may be clearer in purpose.
2. The error checking is hidden from the top level of code and does not cloud purpose.
3. There is actual error handling !
4. We can potentially change the internal representation of the `array_t` without changing the high level code that uses it.

However also keep in mind some issues that may be caused by using this type of defensive programming style.

1. A code usually consists of nested structs (structs that contain struct member variables that contain struct members...).
2. Every level of indirection may take compute cycles and may reduce the performance of your code. For instance consider a task that requires a trillion (10^{12}) array get ops (not an outlandish number). Each array get op function call incurs the stack build up cost for a function call together with the conditional checks for valid array index and valid array pointer. For a CPU that executes instructions at a frequency of 1GHz, i.e. one billion operations per second then each operation takes 1ns. Thus you will have introduced an overhead a thousand seconds to process a trillion get ops.
3. A programmer who decides to uses your `array_t` based code may end up writing pretzel code that has to do a lot of extra operations like create new arrays and copying their data to get the benefit of your code base. This can deter programmers from using your code due to extra effort or possibly producing an inefficient code if they do it in a naïve way.

There are some steps you can take to improve the efficiency of the error handling code. One common approach is to use compiler directives to let you disable error checking at compile time. We can illustrate this via the `arrayGet` function as in the following

```c
int arrayGet(array_t a, int n){

#if ARRAY_DEBUG==1
  if(n<0 || n>=a.Ndata){
    fprintf(stderr, "WARNING: arrayGet size %d array bounds error with index %d\n",
            a.Ndata, n);
    return 0;
  }

  if(a.data == NULL){
    fprintf(stderr, "WARNING: arrayGet request to access NULL array\n");
    return 0;
  }
#endif

  return a.data[n];
}
```

It uses the following preprocessor syntax that is not part of the C language

```c
#if ARRAY_DEBUG==1
 ...
#endif
```

The code bracketed by the `#if/#endif` is only compiled if the `ARRAY_DEBUG` compiler variable is defined and has value 1 when the code is compiled.

To activate that code we would compile the code with the following additional compiler command line argument.

```
gcc -DARRAY_DEBUG=1 -o arrayStruct arrayStruct.c
```

where the `-DARRAY_DEBUG=1` defines a compiler variable called `ARRAY_DEBUG` and sets its value to 1. With this extra option the code will now do the error checking.

Without the option say compiling with

```
gcc -o arrayStruct arrayStruct.c
```

the compiler will omit that error-handling code.

It is quite common practice to use directives in this way to help debug a code and then disable them for production, with the understanding that once you take these figurative training wheels off your code might still fail with some as of yet undiagnosed issue. This approach can be refined further by using the value of the `ARRAY_DEBUG` compiler variable to decide the amount of error checking to do. For instance 0=none, 1=array bounds checking, 2=array bounds+array allocation...

4.11 Struct of arrays

In the previous section we saw how to build a relatively robust `array_t` struct. In practice and in your k-means code you will not have just one array but several arrays. To help manage multiple arrays you can design a `struct` to contain them.

Suppose you have two arrays `x` and `y` then you can design a container `struct` with

```c
typedef struct{
  array_t x;
  array_t y;
}container_t;
```

and follow the same pattern as in the definition of the `array_t` struct type and associated API functions by providing a constructor, destructor, get/set, print implementations.

4.12 Summary of C standard library heap management functions

Table 4.2 summarizes useful standard library function calls for interacting with the heap.

Action	Action implemented using standard library heap functions
Allocate heap array	`[TYPE]* ptr = ([TYPE]*) malloc([N]*sizeof(TYPE));`
Allocate heap array and zero entries	`[TYPE]* ptr = ([TYPE]*) calloc([N],sizeof(TYPE));`
Release heap array to system	`free(ptr);`
Change size of heap array	`ptr = [TYPE]* realloc(ptr, [NEW N]*sizeof(TYPE);`

Table 4.2: Summary of commonly used standard library heap functions. Here `[N]` is the number of entries of type `[TYPE]` in the arrays.

4.13 Reading data from a file

The C standard IO library internally represents a file using a struct of type `FILE`. We do not need to be concerned about the contents of a `FILE` struct, but conceptually we should be aware that once a file is open the library keeps track of our place in the file.

The following code includes the `stdio.h` header file, opens a file, and reads an integer value from the file.

```c
/* L04/readIntFromFile.c */

#include <stdio.h>

int main(int argc, char **argv){

   FILE *fp = fopen("foo.dat", "r");

   int n;
   fscanf(fp, "%d", &n);

   printf("Read %d from foo.dat\n", n);

   fclose(fp);
   return 0;
}
```

We first open the file using the file open function `fopen`.

> **Note #1**: in this case, we pass the name of the file as a constant string literal.
>
> **Note #2**: if the file is not in the present working directory we would have to include the path to the file in the file name string.
>
> **Note #3**: we also pass a second string that specifies that we wish to read from the file. If we had wanted to write to the file we would have replaced the r in the string with w.
>
> **Note #4**: if the file does not exist or cannot be opened for some reason `fopen` will return a NULL pointer. We should add code to check for this.

Once the file is opened we can proceed to scan through the file. We use the `fscanf` file-scan-formatted function in this case to read a single an integer. It will ignore white space (like tabs or blank spaces) and then try to read an integer.

> **Note #1**: we instruct `fscanf` what kind of text we want to read and what kind of number to output through the format string. In this case `%d` specifies that we want `fscanf` to treat the next group of non-space characters as a number.
>
> **Note #2**: once the number is read the standard IO library will advance its internal file pointer to the next character after the end of the number it read.
>
> **Note #3**: you can specify that more than one object should be read by adding extra format blocks to the format string. Some options are: `%c` for a `char`, `%s` for a string, `%f` for a float, etc. See this tutorial for more info.

> **Note #4**: notice how we passed a pointer to the `n` variable. Since `fscanf` can read more than one variable from the file it is necessary to provide pointers to the variables passed in to receive the data from the file.

Finally we use `fclose` to close the file. It is always important to close files after we are done with them, in particular when writing data to them as in the next section.

To demonstrate scanning for more than one thing from a file see the following code

```c
/* L04/readMixedFromFile.c */
#include <stdio.h>

int main(int argc, char **argv){

   FILE *fp = fopen("foo.dat", "r");

   int n,m;
   float f;
   fscanf(fp, "%d %d %f", &n, &m, &f);

   printf("Read %d,%d,%f from foo.dat\n", n,m,f);

   fclose(fp);
   return 0;
}
```

4.14 Writing data to a file

The process of writing data to a file is much as before. In the simplest approach we simply open a new file specifying that we will write to the file and then we use the file-print-formatted standard IO function `fprintf` to write data to the file.

```c
/* L04/writeMixedToFile.c */
#include <stdio.h>

int main(int argc, char **argv){

   /* open bah.dat in the pwd to write to */
   FILE *fp = fopen("bah.dat", "w");

   int n = 10,m = 20;
   float f = 1.3;

   fprintf(fp, "%d %d %f", n, m, f);

   fclose(fp);
```

```
  return 0;
}
```

Note the action string passed to `fopen` is “w” for write. Also note subtle difference between `fprintf` and `fscanf`. Because we are passing data into `fprintf` we pass by value and do not need to pass pointers into the data arguments.

We can further tweak the format string passed to `fprintf` to specify the number of decimal places or whether we should scientific notation for the `float`. You can find formatting string information with your favorite search engine (search: *C format string fprintf*) or you can use the builtin Terminal man pages with

```
man fprintf
```

There are `man` pages for most C standard library functions accessible through the `man` command.

4.15 Example: reading/writing arrays from/to text files

In this section, we will expand on the functionality of our `array` struct from the defensive programming Section 4.10.

We will specify that an `array` file has an expected format:

```
<NUMBER OF ENTRIES IN ARRAY>
number
<ENTRIES IN ARRAY>
first entry
second entry
...
last entry
```

This is a pretty barebones format. We could add a lot more meta-data to the file like data created, author program, format version, precision, variable format. For the moment we will just keep it simple.

To encapsulate interactions between the `array` and the file system we will create two functions: `arrayRead` and `arrayWrite` to read and write arrays respectively.

A barebones `arrayRead` function is as follows

```
array arrayRead(const char *fileName){

  array a;

  FILE *fp = fopen(fileName, "r");
  if(fp){
    char buffer[BUFSIZ];
    /* skip header line */
    fgets(buffer, BUFSIZ, fp);

    /* read next line */
    int Ndata;
    fgets(buffer, BUFSIZ, fp);
    sscanf(buffer, "%d", &Ndata);

    /* construct array */
    a = arrayConstructor(Ndata);

    /* skip header line */
    fgets(buffer, BUFSIZ, fp);

    /* read array entries */
    int n, m;
    for(n=0;n<a.Ndata;n=n+1){
      fscanf(fp, "%d", &m);
      arraySet(a, n, m); /* a[n] = m */
    }
    fclose(fp);
  }
  else{
    a = arrayConstructor(0);
  }
  return a;
}
```

The only new thing here is the use of the `fgets` to read a line of up to `BUFSIZ` characters into a stack string arrray called `buffer`. The `fgets` will read up `BUFSIZ` characters, finishing early if it encounters a line return in the file. If we were being super careful we would guard each call to `fgets` and verify that it does not return a NULL pointer. We would also check that the header line for each section contains the expected text.

A simplified implementation of `arrayWrite` follows

```
void arrayWrite(const char *fileName, array a){

  FILE *fp = fopen(fileName, "w");

  if(fp){

    fprintf(fp, "<NUMBER OF ENTRIES IN ARRAY>\n");
    fprintf(fp, "%d\n", a.Ndata);

    fprintf(fp, "<ENTRIES IN ARRAY>\n");
    /* read array entries */
    int n, m;
    for(n=0;n<a.Ndata;n=n+1){
      int m = arrayGet(a, n); /* m = a[n] */
      fprintf(fp, "%d\n", m);
    }
    fclose(fp);
  }
  else{
    fprintf(stderr, "WARNING: failed to open %s to write\n", fileName);
  }
}
```

This implementation applies concepts previously discussed.

In both `arrayRead` and `arrayWrite` we follow the spirit of the `array` implementation and use the constructor, get, and set functions. Reflecting on this code we should also add a function to the growing collection of `array` functions to query the `Ndata` i.e. the number of entries in the `array`.

The following illustrates how we can read and then write an `array` file.

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

  array a;

  /* read array */
  a = arrayRead("arrayA.dat");

  /* write array out to copy */
  arrayWrite("copyArrayA.dat", a);

  return 0;
}
```

We do not directly access member variables of the `array` struct at any point in this high-level code.

4.16 Summary of useful C standard input-output functions

Table 4.3 summarizes useful standard library function calls for interacting with the heap. Table 4.4 summarizes useful variables defined in the standard input-output library.

Action	Action implemented using standard input-output functions
Open file to read	`FILE* [FILE POINTER] = fopen("[FILE NAME]", "r"];`
Open file to write	`FILE* [FILE POINTER] = fopen("[FILE NAME]", "w"];`
Formatted print to file	`fprintf([FILE POINTER], "[FORMAT]", [VAL1], [VAL2], ...);`
Formatted read from file	`fscanf([FILE POINTER], "[FORMAT]", &[VAL1], &[VAL2], ...);`
Read string from file	`fgets([BUFFER POINTER], [NUMBER OF CHARS], [FILE POINTER]);`
Write a string to file	`fputs([BUFFER POINTER], [FILE POINTER]);`
Flush pending writes to file	`fflush([FILE POINTER]);`
Close file	`fclose([FILE POINTER]);`

Table 4.3: Summary of commonly used standard library heap functions. Here `[N]` is the number of entries of type `[TYPE]` in the arrays.

Name	Variable
Standard output terminal stream	`stdout`
Standard input terminal stream	`stdin`
Standard error terminal stream	`stderr`
Recommended buffer size	`BUFSIZ`

Table 4.4: Summary of standard input-output defines.

4.17 Debugging tips

In this section, we give common sense debugging strategies and a checklist of problems to look for when debugging a code. The list is by no means comprehensive because there are more ways to make a code buggy as there are ways to write a correct code.

1. **Typos**: You introduced a typo in code and the compiler will not compile your code. Scrutinize the compiler errors and warnings starting with the first one. A systematic approach is to review each compiler error one at a time starting with the first error in the compiler output. Fixing the first error may actually fix a slew of subsequent errors. The compiler is not judging your character when it emits an error, and it is a private process so do not be afraid to iteratively fix bugs with several compile-then-edit steps.

   Common syntactical typos include:

   (a) Missing a semi-colon at the end of a line of code or command.

   (b) Missing curly braces: { or }. Fortunately `emacs` and other so-called text editors can give you clues about what braces are needed to close a code section.

   (c) Not using proper code indentation (for instance not using the auto-indentation feature in `emacs`) will obscure the code block structure of your code.

   (d) Incorrect spelling of variable names, function names, or C language keywords.

   (e) Failing to terminate a C comment by not pairing the `/*` with `*/`.

   (f) Be careful not to nest `/* ... */` comments !

   (g) Using the wrong variable name.

   (h) Try to avoid the lower case `l` variable, it looks too much like the number one in many editors.

2. **Variable type errors**: C is a strongly typed language. We have seen repeatedly that integer arithmetic can cause problems, in particular when dividing integer by integer.

3. **Functions**: many bugs can creep in when writing or calling functions.

   (a) Calling the wrong function.

   (b) Providing the wrong list of arguments to a function.

   (c) Providing the right arguments but in the wrong order.

   (d) Providing arguments with the wrong type (e.g. `int` instead of `double`).

   (e) Misunderstanding exactly what a function does. It is important to document your own functions, for instance by adding comments about input, output, and purpose. If you use a function from a third party library you need to scrutinize the documentation.

   (f) Forgetting to `return` a value from a function that is supposed to return a value by design.

   (g) Not checking error codes returned from a function.

   (h) Stale comments that obscure the actual purpose of a function.

   (i) Providing the pointer to a pointer as an argument to a function when you only need to provide the pointer itself.

   (j) Forgetting that C passes variable values to functions by copy. If you pass a variable by copy, and the function changes the value of the copy, then the original paased variable will not change.

   (k) Neglecting to provide a function prototype to a function, by for instance forgetting to include a header file including the function prototype. If a function is not in scope of the calling site (i.e. defined or prototyped above the call site) then the C compiler assumes that the return argument and input arguments are of `int` type.

4. **Heap memory errors**: C exposes direct access to system memory. This is a strength in terms of code performance but also a weakness in terms of creating many ways to make mistakes.

   (a) Failing to allocate a heap array and leaving a pointer uninitialized.

   (b) Trying to allocate a heap array with `malloc` or `calloc` but receiving a `NULL` pointer because there was insufficient available heap space. Trying to dereference a `NULL` pointer will cause a segmentation fault.

   (c) Allocating a heap array with insufficient number of bytes because your array length calculation was wrong.

   (d) Trying to read a heap array entry outside the allocated array space. This so-called bounds error will yield garbage values.

   (e) Trying to write to a heap array entry outside of the allocated array space. This is a more serious array bounds error as it may change the value of some other heap array. This can cause silent errors that are only exposed when the other heap array is changed in a critical way.

   (f) The most common cause of array error bounds is incorrectly forming the array index. Check your index arithmetic !

   (g) Failing to `free` a heap array. Although this might not cause problems for your testing, leaking memory in this way may cause problems if the program runs long enough that eventually allocating new arrays is not possible because you have used all available heap space.

The above issues may be detected by using the `valgrind` memory checking tool described later in this lecture.

5. **The compiler did it**: It is only human to think that your code is perfect and that the compiler must have misunderstood your code and made a compilation error. This is almost never the case. However, just once in a while a compiler optimization subtly introduce unexpected behavior. If the code generates different results with optimization turned on and off then it is likely your issue.

6. **The CPU must be bad**: Very, very rarely there is a CPU hardware bug. An infamous example is the Intel Pentium CPU floating point division (FDIV) bug (see wiki). Hardcoded values used in specialized hardware floating point unit division were incorrect leading to inaccurate results from division operations for some denominators.

It is highly unlikely that a CPU hardware bug is responsible for your coding error.
This is almost certainly not the root cause of your problem.

The bottom line is that you have to be very detail oriented when writing code. The compiler expects precision and is not in the business of fixing your code for you ! In the following sections we describe how to use the GNU debugger `gdb` and the memory checking tool `valgrind` to try to find the errors that the compiler does not flag.

4.18 Debugging problems with the GNU debugger `gdb`

For as long as people have been programming they have been creating bugs. There is some entertaining background on software bugs here. Fortunately some useful tools can help find bugs in code. The GNU debugger, `gdb` for short, is a command line tool that allows you to troubleshoot issues with a code interactively. `gdb` is included in the `gcc` package.

Note #1: one prerequisite for using `gdb` is that your program compiles.

Note #2: the debugger is not magic, it is simply a tool that might help you to diagnose a coding error.

Running your program within `gdb` you can interactively step line by line to do some detective work about what might be wrong. When you are working with a non-trivial code with deep call stacks being able to insert break points in certain functions and stop execution there can be invaluable. Likewise it can be helpful to print out the values of variables when your code is in a function deep in the call stack.

The first step in preparing your code for debugging is to compile your code with the `-g` command line argument. This instructs the compiler to include the original source code into the executable. The following shows how to compile with debugging information and then launch a sample code with `gdb`

```
gcc -g -o debugExample debugExample.c
gdb ./debugExample
GNU gdb (Ubuntu 8.1-0ubuntu3) 8.1.0.20180409-git
...
Reading symbols from ./debugExample...done.
(gdb)
```

At this point the debugger has started. It is waiting for your input at the `(gdb)` prompt.

In this example I have loaded the `debugExample` executable compiled from this source code

```c
/* L04/debuggingExample.c */
#include <stdio.h>

float otherFunction(float a){
  float b = 3.2*a;
  return b;
}

float someFunction(float a){
  int b = 2*otherFunction(a);
  return b;
}

int main(int argc, char **argv){
  float a = 1.2;
  float b = someFunction(a);

  printf("a=%f, b=%f\n", a, b);
  return 0;
}
```

There is something wrong with this code. It should print out `a=1.2, b=7.68`. We can run it from inside `gdb` and it actually prints out

```
(gdb) run
Starting program: /home/tcew/course/L09/debugExample
a=1.200000, b=7.000000
```

You can probably figure out what is going on by carefully reading the code, but we can use `gdb` to see what might be going wrong.

To start I set a "breakpoint" in `main` and then start `gdb` running the code as follows.

```
(gdb) break main
Breakpoint 1 at 0x5555555546b2: file debugExample.c, line 14.
(gdb) run
Starting program: /home/tcew/course/L09/debugExample

Breakpoint 1, main (argc=1, argv=0x7fffffffe0a8) at debugExample.c:14
14 float a = 1.2;
(gdb)
```

The debugger stops execution at the first executable statement in the function where the breakpoint is set. Here the debugger has stopped at the line where the variable `a` is iniitialized. You can see the next few lines of code by typing `list`

```
(gdb) list
9   int b = 2*otherFunction(a);
10   return b;
11   }
12
13 int main(int argc, char **argv){
14   float a = 1.2;
15   float b = someFunction(a);
16
17   printf("a=18   return 0;
```

We can step through line by line as the code is executed using the `step` command. In the following code I type step and every time I press enter the debugger either processes the next line of code or steps into the function if the statement is a function call.

```
(gdb) step
15 float b = someFunction(a);
(gdb)
someFunction (a=1.20000005) at debugExample.c:9
9 int b = 2*otherFunction(a);
(gdb)
otherFunction (a=1.20000005) at debugExample.c:4
4 float b = 3.2*a;
(gdb)
5 return b;
(gdb)
6 }
(gdb)
someFunction (a=1.20000005) at debugExample.c:10
10 return b;
(gdb)
11 }
(gdb)
```

Thus we have seen the sequence of statements and functions calls that have been made prior to the `printf` statement and we didn't need to read the code to figure the sequence out.

We will suppose that something went wrong in the function named `someFunction` and stop the debugger executing use the `break` command as before. When we `run` the code will first stop in `main` at the first breakpoint. We type `cont` and the debugger will continue until the second breakpoint.

```
(gdb) break otherFunction
Breakpoint 2 at 0x653: file debugExample.c, line 4.
(gdb) run
Starting program: /home/tcew/course/L09/debugExample
Breakpoint 1, main (argc=1, argv=0x7fffffffe0a8) at debugExample.c:14
14 float a = 1.2;
(gdb) cont
Continuing.

Breakpoint 2, otherFunction (a=1.20000005) at debugExample.c:4
4 float b = 3.2*a;
(gdb)
```

We can then print the values of the variables using the `print` command and step to the next line using the `next` command.

```
Breakpoint 2, otherFunction (a=1.20000005) at debugExample.c:4
4 float b = 3.2*a;
(gdb) print a
$1 = 1.20000005
(gdb) next
5 return b;
(gdb) print b
$2 = 3.84000015
(gdb)
```

And so far this all seems good. We can see where we are in the call stack using the `where` command.

```
(gdb) where
#0 otherFunction (a=1.20000005) at debugExample.c:5
#1 0x0000555555554691 in someFunction (a=1.20000005) at debugExample.c:9
#2 0x00005555555546cf in main (argc=1, argv=0x7fffffffe0a8) at debugExample.c:15
(gdb)
```

This is a stack trace showing the arguments to each of the functions in the call stack. We see we are two functions down the stack from the `main` function. So the mystery continues. We can advance the code to return from the `otherFunction`.

```
(gdb) next
someFunction (a=1.20000005) at debugExample.c:10
10 return b;
(gdb) where
#0 someFunction (a=1.20000005) at debugExample.c:10
#1 0x00005555555546cf in main (argc=1, argv=0x7fffffffe0a8) at debugExample.c:15
(gdb) print b
```

```
$3 = 7
```

So something weird happened. A floating point number was returnedd from `otherFunction` but turned into an integer inside `someFunction`. Looking at the code for `someFunction` andd we see that the `b` variable is actually an `int`, i.e. an integer. That wasn't my original intent, bug found !

4.18.1 Summary of `gdb` commands

Summary of `gdb` commands discussed above plus a few extra commands for navigating the call stack.

Action	`gdb` command
Start running	`run [COMMAND LINE ARGUMENTS]`
Add a breakpoint in [FUNCTION]	`break [FUNCTION]`
Continue running after breakpoint	`cont`
Print value of [VARIABLE NAME]	`print [VARIABLE NAME]`
Process next statement	`next`
Step into next function or process next step	`step`
Print call stack trace	`where`
List next few lines of code	`list`
Move up the call stack	`up`
Mode down the call stack	`down`
Quite debugger	`quit`

Table 4.5: Summary of commonly used `gdb` commands discussed above.

Note: you can just use the first letter of each command instead of the whole command.

4.19 Using the `valgrind` memory checker

valgrind is an invaluable tool for detecting memory errors including heap bounds access errors, leaked heap arrays, and attempting to free previously freed arrays. It works by intercepting systems calls including memory accesses and doing the type of error handling that we added to the `array` functions. Unfortunately this might significantly prolong the time it takes to run a program.

We can run the `array` program by the following steps. First make sure that `valgrind` is installed the first time you need it with

```
sudo apt-get install valgrind
```

then we compile with the compiler debugging flag

```
# add debugging info to include line numbers
gcc -g -I. -o arrayMain arrayMain.c array.c

# run with valgrind
valgrind ./arrayMain
```

Doing this yielded

```
==5030== Memcheck, a memory error detector
==5030== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==5030== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==5030== Command: ./arrayMain
==5030==
==5030== Invalid write of size 4
==5030==    at 0x108A37: arraySet (arrayMain.c:62)
==5030==    by 0x108DBB: main (arrayMain.c:148)
==5030==  Address 0x522d06c is 4 bytes after a block of size 40 alloc'd
==5030==    at 0x4C31B25: calloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==5030==    by 0x10898D: arrayConstructor (arrayMain.c:18)
==5030==    by 0x108D79: main (arrayMain.c:145)
==5030==
==5030==
==5030== HEAP SUMMARY:
==5030==     in use at exit: 40 bytes in 1 blocks
==5030==   total heap usage: 6 allocs, 5 frees, 9,376 bytes allocated
==5030==
==5030== LEAK SUMMARY:
==5030==    definitely lost: 40 bytes in 1 blocks
==5030==    indirectly lost: 0 bytes in 0 blocks
==5030==      possibly lost: 0 bytes in 0 blocks
==5030==    still reachable: 0 bytes in 0 blocks
==5030==         suppressed: 0 bytes in 0 blocks
==5030== Rerun with --leak-check=full to see details of leaked memory
==5030==
==5030== For counts of detected and suppressed errors, rerun with: -v
==5030== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

This run revealed two interesting things. First the "Invalid write of size 4" shows that the code commits an out of bounds error and the line number is provided. A moments thought reveals that without the `ARRAY_DEBUG` compiler variable set to 1 the `arraySet` function actually does the out of bounds write that we requested to test our out of bounds checker.

Secondly, the leak summary shows that we neglected to call the `arrayDestructor` and that 40 bytes (ten 4-byte ints) were leaked.