An introduction to C

by Mike Jackson, EPCC, The University of Edinburgh, August 2019.

Acknowledgements

Sections 1-4 are derived, and expanded, from C for Python programmers by Carl Burch, Hendrix College, August 2011, licensed under a Creative Commons Attribution-Share Alike 3.0 United States License.

The current author converted that document into Markdown, reworded parts of sections 1-4, added notes on other languages, added notes on exit codes, and added sections 5 onwards.

Licensing

This document is licensed under a Creative Commons Attribution-Share Alike 3.0 United States License.

Contents

- 1. Introduction
- 2. Building a program
  - 2.1. Compilers versus interpreters
  - 2.2. Variable declarations
  - 2.3. Whitespace
  - 2.4. The `printf()` function
  - 2.5. Functions
- 3. Statement-level constructs
  - 3.1. Operators
  - 3.2. Basic types
  - 3.3. Braces
  - 3.4. Declarations
  - 3.5. Expressions
  - 3.6. `if…else if…else`
  - 3.7. `return`
  - 3.8. `while`
  - 3.9. `for`
  - 3.10. `break;`
  - 3.11. `continue;`
  - 3.12. `switch`
  - 3.13. Arrays
  - 3.14. Arrays and strings
  - 3.15. Comments
- 4. Libraries
  - 4.1. Function prototypes
  - 4.2. Header files
  - 4.3. Standard header files

    - 4.4. Constants
- 5. Building C programs revisited
- 6. Pointers
    - 6.1. Dereferencing pointers
    - 6.2. `NULL` pointers
    - 6.3. Pointers are variables
    - 6.4. Pointers to pointers
    - 6.5. Pointers and strings
    - 6.6. Pointers and arrays
    - 6.7. Passing pointers to functions
- 7. Dynamic memory allocation
    - 7.1. Freeing memory
    - 7.2. Allocating memory for arrays and EPCC's `arralloc` function
- 8. User-defined data types
    - 8.1. Passing `structs` to functions
    - 8.2. Dynamically allocating memory for arrays of `struct`
- 9. File output
- 10. Miscellaneous
    - 10.1. Command-line arguments
    - 10.2. Timing
- 11. Further information

---

1. Introduction

In the 1970's at Bell Laboratories, Ken Thompson designed the C programming language to help with developing the UNIX operating system. Through a variety of historical events, few intentional, UNIX grew from a minor research diversion into a popular industrial-strength operating system. And along with UNIX's success came C, since the operating system was designed so that C programs could access all of its features. As more programmers gained experience with C, they began to use it on other platforms, too, so that it became one of the primary languages for developing software by the end of the 1980's.

While C does not enjoy the broad dominance it once did, its influence was so great that many other languages were designed to look like it, including C++, C#, Objective-C, Java, JavaScript, PHP, and Perl. In particular, C's "influence on Python is considerable," in the words of Python's inventor, Guido van Rossum ("An Introduction to Python for UNIX/C Programmers," 1993).

Knowing C is in itself a good starting point for relating more directly with what a computer does.

---

2. Building a program

We'll start with several general principles, working toward a complete - but limited - C program by the end of this section.

2.1. Compilers versus interpreters

In common with C++, FORTRAN and Java, we use a *compiler* when we are ready to execute a C program. This is in contrast to Python or R, where we typically use an *interpreter*.

A **compiler** generates a file containing the translation of the program into the machine's native code (also known as **machine code**). The compiler does not actually execute the program. It only creates the native machine code, an **executable**. We then run this executable. So, after writing a C program, executing it is a two step process:

```
$ gcc example.c
$ ./a.out
GCD: 8
```

In the first command (`gcc example.c`), we invoke the C compiler, `gcc`. The compiler reads our C source code, in `example.c`, and it generates a new file, `a.out`, containing a translation of the source code into the machine code that can be run by the computer.

In the second command (`./a.out`), we tell the computer to execute this machine code. As it is executing the machine code, the computer has no idea that `a.out` was created from a C program. The computer executes the machine code found within the `a.out` file, just as it executes the code found within the `gcc` file, the machine code which implements the compiler itself (which is also a program).

In contrast, an **interpreter** reads our program and runs it directly. This removes the compilation step from our process. If our program was in Python we would run:

```
$ python my_program.py
GCD: 8
```

The Python interpreter, `python`, would read each Python statement in our program, convert it to machine code and execute it.

While the use of a compiler introduces an additional compilation step when building and running C programs it does offer advantages too. Compiled programs can conceivably lead to faster runtimes. Compilers can also detect errors in our code when the code is compiled, errors that may go undetected until runtime if our code is in a language that is interpreted.

Being compiled has implications on programming language design. C is designed so that the compiler can deduce everything it needs to know to translate the C program into machine code without actually executing the program.

2.2. Variable declarations

Among the ways that C requires programmers to give information to the compiler, one of the most notable examples is that C requires **variable declarations**. These inform the compiler about each variable before the variable is actually used. This is typical of many programming languages, particularly among those that are compiled before being executed, such as C++, FORTRAN and Java.

In C, the variable declaration defines the variable's **type**, specifying whether it is an integer (`int`), floating-point number (`double`), character (`char`), or some other type. Once we declare a variable to be of a particular type, we cannot change its type: if the variable `x` is declared of type `double`, and we assign `x = 3;`, then `x` will

actually hold the floating-point value 3.0 rather than the integer 3. We can imagine that x is a box that is capable only of holding `double` values; if we attempt to place something else into it (like the `int` value 3), the compiler converts it into a `double` so that it will fit.

To declare a variable, we enter the variable's type, some whitespace, the variable's name, and a semicolon. For example:

```
double x;
```

If we forget to declare a variable, the compiler will refuse to compile a program in which a variable is used but is not declared. The error message will indicate the line within the program, the name of the variable, and a message. For example:

```
$ gcc example.c
example.c: In function "main":
example.c:12:3: error: "a" undeclared (first use in this function)
   a = 1;
   ^
```

To a Python or R programmer, it may seem a pain to have to include these variable declarations in a program, though this gets easier with more practice. One advantage is that the compiler will automatically identify any time a variable name is misspelled, and point directly to the line where it is misspelled. This is a lot more convenient than executing a program and finding that it has gone wrong somewhere because of the misspelled variable name. These errors are detected at compile time rather than at run time.

2.3. Whitespace

In Python, whitespace characters like tabs and newlines are important: we separate our statements by placing them on separate lines, and we indicate the extent of a block (like the body of a `while` or `if` statement) using indentation. These uses of whitespace are idiosyncrasies of Python. FORTRAN also uses line breaks to separate statements, but no other major language relies on whitespace for indicating blocks.

Like the majority of programming languages, C does not use whitespace except for separating words. In common with C++ and Java, most statements are terminated with a semicolon `;`, and blocks of statements are indicated using a set of braces, `{` and `}`. Here is an example fragment from a C program, with its Python equivalent:

```
C fragment                | Python equivalent
                          |
disc = b * b - 4 * a * c; | disc = b * b - 4 * a * c
if (disc < 0) {           | if disc < 0:
  num_sol = 0;            |     num_sol = 0
} else {                  | else:
  t0 = -b / a;            |     t0 = -b / a
  if (disc == 0) {        |     if disc == 0:
    num_sol = 1;          |         num_sol = 1
```

```
    sol0 = t0 / 2;          |         sol0 = t0 / 2
  } else {                  |     else:
    num_sol = 2;            |         num_sol = 2
    t1 = sqrt(disc) / a;    |         t1 = disc ** 0.5 / a
    sol0 = (t0 + t1) / 2;   |         sol0 = (t0 + t1) / 2
    sol1 = (t0 - t1) / 2;   |         sol1 = (t0 - t1) / 2
  }                         |
}                           |
```

In C, whitespace is insignificant, so the computer would be just as happy if it had instead been written as:

```
disc=b*b-4*a*c;if(disc<0){
num_sol=0;}else{t0=-b/a;if(
disc==0){num_sol=1;sol0=t0/2
;}else{num_sol=2;t1=sqrt(disc/a;
sol0=(t0+t1)/2;sol1=(t0-t1)/2;}}
```

While the computer might be just as happy with this, no human would prefer it. So programmers tend to be very careful in how they use whitespace to indicate program structure.

There are some exceptions to the rule of ignoring whitespace: it is occasionally significant for separating words and symbols. The fragment `intmain` is different from the fragment `int main`; likewise, the fragment `a++ + 1` is different from the fragment `a+ + +1` (the former does an in-place increment of `a`, denoted by `++`, the latter does not).

2.4. The `printf()` function

As we work toward writing useful C programs, one important ingredient is displaying results for the user to see. Many languages provide commands for printing results, for example `print` in FORTRAN, `System.out.println` in Java or `print` in Python.

In C, we can use the `printf()` function, one of the most useful among C's library of language-defined functions.

`printf()` takes two parameters. The first parameter is a string specifying the format of what to print, and the following parameters are the values to print. For example:

```
printf("# solns: %d\n", num_sol);
```

This line tells C to print using `# solns: %d\n` as the format string. The `printf()` function goes through this format string, printing the characters `# solns:` before getting to the percent character `%`. The percent character is special to `printf()` instructing it to print a value specified in a subsequent parameter. In this case, a lower-case `d` follows the percent character, stating that the parameter is to be printed as an `int` in decimal form (the `d` stands for decimal). So when `printf()` reaches `%d`, it looks at the value of the following parameter, `num_sol`, and prints the value of that parameter. It then continues through the format string, in

this case displaying a newline character, `\n`. If `num_sol` has the value 2, then the user sees the following line printed:

```
# solns: 2
```

C allows us to include escape characters in a string using a backslash:

- `\n` represents the newline character, that is, the character that represents a line break.
- `\t` represents the tab character.
- `\"` represents the double-quote character.
- `\\` represents the backslash character.

These escape characters are part of C syntax, not part of the `printf()` function. That is, the string the `printf()` function receives actually contains a newline, not a backslash followed by an `n`. Thus, the nature of the backslash is fundamentally different from the percent character, which `printf()` would see and interpret at run-time.

Let's look at another example:

```
printf("# solns: %d", num_sol);
printf("solns: %f, %f", sol0, sol1);
```

Let's assume `num_sol` holds the value 2, `sol0` holds 4, and `sol1` holds 1. When the computer reaches these two `printf()` function calls, it first executes the first line, which displays `# solns: 2`, and then the second, which displays `solns: 4.0, 1.0`, as shown below:

```
# solns: 2solns: 4.0, 1.0
```

Note that `printf()` displays only what it is told to display, without adding any extra spaces or newlines. If we want a newline to be inserted between the two pieces of output, we would need to include `\n` at the end of the first format string.

The second call to `printf()` in this example illustrates how the function can print multiple parameter values. In fact, there's really no reason we couldn't have combined the two calls to `printf()` into one in this case:

```
printf("# solns: %dsolns: %f, %f", num_sol, sol0, sol1);
```

In this example `printf()` displays `4.0` rather than `4` because the format string uses `%f`, which says to interpret the parameters as floating-point numbers. If we want it to display just `4`, we might be tempted to use `%d` instead. But that wouldn't work, because `printf()` would interpret the binary representation used for a floating-point number as a binary representation for an integer, and these types are stored in completely different ways. On the author's computer, replacing each `%f` with `%d` in the above example leads to the following output:

```
# solns: 2solns: 0, 1074790400
```

There is a variety of characters that can follow the percent character in the formatting string.

- %d: print an `int` value in decimal form.
- %x: print an `int` value in hexadecimal form.
- %f: print a `double` value in decimal-point form.
- %e: print a `double` value in scientific notation (for example, `3.000000e8`).
- %c: print a `char` value.
- %s: print a string. There is no variable type for representing a string, but C does support some string facilities using arrays of characters which will be discussed later. In the meantime here is an example of printing a string and the output when the program is run:

```
char name[2] = "jo";

printf("Hello %s\n", name);
```

```
Hello jo
```

We can also include a number between the percent character and the format descriptor. For example %4d tells `printf()` to right-justify a decimal integer over 4 columns. Compare and contrast:

```
printf("%d %d %d\n", 1, 10, 100);
printf("%4d %4d %4d\n", 2, 20, 200);
```

```
1 10 100
   2   20  200
```

2.5. Functions

All C code must be nested within functions, and functions cannot be nested within each other. This is similar to C++ and Java (which expects all code to be nested within classes or other object-oriented constructs) and, in contrast to Python or R, where programs can be written with no functions definitions at all.

A C program's overall structure is a list of function definitions, one after another, each containing a list of statements to be executed when the function is called.

Here's an example of a function definition:

```
double exponent(double b, int e) {
  if (e == 0) {
    return 1.0;
  } else {
    return b * exponent(b, e - 1);
  }
}
```

A C function is defined by naming the return type (`double` here, since the function produces a floating-point result), followed by the function name (`exponent`), followed by a set of parentheses listing the parameters. Each parameter is described by including the type of the parameter and the parameter name. Following the parameter list in parentheses is a set of braces, in which we nest the body of the function.

If we have a function that does not have any useful return value, then we must use `void` as the return type.

C differs from languages such as Python or R in that both the types of parameters, and the return type (or `void` if no value is returned) must be defined.

C programs have one special function named `main`, whose return type is an integer. This function is the "starting point" for the program: the computer essentially calls the program's `main` function when it executes the program. The integer return value can be used as an **exit code** to indicate whether execution completed successfully or failed somehow. Convention is that an exit code of 0 means execution completed successfully. A non-zero exit code means that execution failed in some way. How errors are mapped to non-zero exit codes, and what exit codes are used for what errors, is a choice for us as developers.

We are now in a position to present a complete C program, along with its Python equivalent:

```
C program                  | Python program
                           |
#include <stdio.h>         |
                           |
int gcd(int a, int b) {    | def gcd(a, b):
  if (b == 0) {            |     if b == 0:
    return a;              |         return a
  } else {                 |     else:
    return gcd(b, a % b);  |         return gcd(b, a % b)
  }                        |
}                          |
                           |
int main() {               |
  printf("GCD: %d\n",      | print("GCD: " + str(gcd(24, 40)))
    gcd(24, 40));          |
  return 0;                |
}                          |
```

The C program consists of two function definitions. In contrast to the Python program, where the `print` line exists outside any function definitions, the C program requires `printf()` to be in a function too, in this case we include it in the program's `main` function.

The `#include <stdio.h>` allows us to use the `printf` function, one of C's standard library functions. We discuss these further in section 4.

---

3. Statement-level constructs

Now that we've seen how to build a complete C program, let's learn the universe of what we can do inside a C function by running through C's statement-level constructs.

3.1. Operators

An **operator** is something that we can use in arithmetic expressions, like a plus sign + or an equality test ==. C's operators are similar to those in C++, Fortran and Java. They may also look familiar to Python developers since Python's designer Guido van Rossum chose to start from C's list of operators; but there are some significant differences.

```
C operator precedence | Python operator precedence
                      |
++ -- (postfix)       | **
+ - ! (unary)         | + - (unary)
* / %                 | * / % //
+ - (binary)          | + - (binary)
< > <= >=             | < > <= >= == !=
== !=                 | not
&&                    | and
|                     | or
||                    |
= += -= *= /= %=      |
```

C does not have an exponentiation operator. For exponentiation in C, we can use the library function `pow()`. For example, `pow(1.1, 2.0)` computes 1.1^2.

C uses symbols rather than words for the Boolean operations AND (`&&`), OR (`||`), and NOT (`!`).

The precedence level of NOT (the `!` operator) is very high in C. This is almost never desired, so we end up needing parentheses most times we want to use the `!` operator.

C defines assignment as an operator, whereas some languages, for example, Python, defines assignment as a statement. The value of the assignment operator is the value assigned. A consequence of C's design is that an assignment can legally be part of another statement. For example:

```
while ((a = getchar()) != EOF)
{
    ...
}
```

Here, we assign the value returned by `getchar()` to the variable `a`, and then we test whether the value assigned to `a` matches the `EOF` constant, which is used to decide whether to repeat the loop again. Many

people contend that this style of programming is bad style; others find it too convenient to avoid. Python, in contrast, was designed so that an assignment must occur as its own separate statement, so nesting assignments within a `while` statement's condition is illegal in Python.

C's operators `++` and `--` are for incrementing and decrementing a variable. Thus, the statement `i++` is a shorter form of the statement `i = i + 1` (or `i += 1`). Similarly, `j--` is a shorter form of the statement `j = j - 1`.

C's division operator, `/`, does integer division if both sides of the operator have an `int` type. Any remainder is ignored with such a division. Thus, in C the expression `13 / 5` evaluates to 2, while `13 / 5.0` is 2.6: the first has integer values on each side, while the second has a floating-point number on the right.

In contrast, in newer versions of Python (3.0 and later), the single-slash operator, `/`, always does floating-point division. With older Python versions, the single-slash operator worked as with C, but this would often lead to bugs - in part because the type associated with a variable is not fixed as it is in a C program.

3.2. Basic types

C's list of types is quite constrained compared to other languages:

- `int`: integer
- `char`: single character
- `float`: single-precision floating-point number
- `double`: double-precision floating-point number

We can create other types of integers, using the type names `long` and `short`. A `long` reserves at least as many bits as an `int`, while a `short` reserves fewer bits than an `int` (or the same number). The language does not guarantee the number of bits for each, but most current compilers use 32 bits for an `int`, which allows numbers up to 2.15x10^9. This is sufficient for most purposes, and many compilers also use 32 bits for a `long` anyway, so people typically use `int` in their programs.

The `char` type represents a single character, like a letter or punctuation symbol. We can represent an individual character in a program by enclosing the character in single quotation marks. For example, `digit0 = '0';" would place the zero digit character into the charvariabledigit0`.

Of the two floating-point types, `float` and `double`, most programmers today stick almost exclusively to `double`. These types are for numbers that could have a decimal point in them, like 2.5, or for numbers larger than an `int` can hold, like 6.02x10^23. The two types differ in that a `float` typically takes only 32 bits of storage while a `double` typically takes 64 bits. The 32-bit storage technique allows a narrower range of numbers (-3.4x10^38 to 3.4x10^38) and - more problematic - about 7 significant digits. A `float` could not store a number like 281,421,906 (the U.S.'s population in 2000, according to the census), because it requires nine significant digits; it would have to store an approximation instead, like 281,421,920. By contrast, the 64-bit storage technique allows a wider range of numbers (-1.7x10^308 to 1.7x10^308) and roughly 15 significant digits. This is more adequate for general purposes, and the extra 32 bits of storage is rarely worth saving, so `double` is almost always preferred.

C does *not* have a Boolean type for representing *true* or *false* values. This has major implications for a statement like `if`, where we need a test to determine whether to execute the body. C's approach is to treat the integer 0 as *false* and all other integer values as *true*. The following would be a legal C program:

```c
#include <stdio.h>

int main() {
  int i = 5;
  if (i) {
      printf("in if\n");
  } else {
      printf("in else\n");
  }
  return 0;
}
```

This program would compile, and it would print `in if` when executed, since the value of the `if` expression, `i`, evaluates to 5, which is not 0, and thus the `if` condition succeeds.

C's operators that look like they should compute Boolean values (==, &&, and ||) actually compute `int` values instead. In particular, they compute 1 to represent *true* and 0 to represent *false*. This means that we could use the following to count how many of values `a`, `b`, and `c` are positive:

```c
pos = (a > 0) + (b > 0) + (c > 0);
```

This aspect of C - that C regards all non-zero integers as true - is generally regarded as a mistake. C introduced it as machine languages rarely have direct support for Boolean values, but typically machine languages expect us to accomplish such tests by comparing to zero. But compilers have improved beyond the point they were when C was invented, and they can now easily translate Boolean comparisons to efficient machine code. What's more, this design of C leads to confusing programs, so most expert C programmers eschew using the shortcut, preferring instead to explicitly compare to zero as a matter of good programming style. But such avoidance doesn't fix the fact that this language quirk often leads to program errors. Many newer languages have a special type associated with Boolean values, for example the type `boolean` with values `true` and `false` in Java or the type `bool` with values `True` and `False` in Python.

3.3. Braces

Several statements, like the `if` statement, include a body that can hold multiple statements. Typically the body is surrounded by braces, { and }, to indicate its extent. But when the body holds only a single statement, the braces are optional. Thus, we could find the maximum of two numbers using an `if` statement with no braces, since the body of both the `if` and the `else` contain only a single statement:

```c
if (first > second)
  max = first;
else
  max = second;
```

C programmers use this quite often when they want one of several if tests to be executed. For example, consider this code:

```
disc = b * b - 4 * a * c;
if (disc < 0) {
  num_sol = 0;
} else {
  if (disc == 0) {
    num_sol = 1;
  } else {
    num_sol = 2;
  }
}
```

Notice that the first `else` clause here holds just one statement (an `if`...`else` statement), so we can omit the braces around it. We might then write it thus:

```
disc = b * b - 4 * a * c;
if (disc < 0) {
  num_sol = 0;
} else
  if (disc == 0) {
    num_sol = 1;
  } else {
    num_sol = 2;
  }
```

Or even:

```
disc = b * b - 4 * a * c;
if (disc < 0)
  num_sol = 0;
else
  if (disc == 0)
    num_sol = 1;
  else
    num_sol = 2;
```

This situation arises often enough that C programmers follow a special rule for indenting in this case - a rule that allows all cases to be written at the same level of indentation. For example:

```
disc = b * b - 4 * a * c;
if (disc < 0) {
  num_sol = 0;
} else if (disc == 0) {
  num_sol = 1;
} else {
  num_sol = 2;
}
```

Because this is feasible using C's bracing rules, C does not include the concept of an `elif` clause that we find in Python. We can string together as many `else if` combinations as we want.

It is recommended always including the braces. As we continue working on a program, we often find that we want to add additional statements into the body of an `if`, and having the braces there already saves us the bother of adding them later on. And it makes it easier to keep track of braces, since each indentation level requires a closing right brace.

3.4. Declarations

```
int count;
double temperature;
char code;
```

We already discussed variable declarations in Section 2.2. These are analogous to declarations in C++, FORTRAN or Java but have no equivalent in Python.

3.5. Expressions

Expressions can be statements. The following is a valid statement:

```
y + z;
```

However, such a statement has no point: we ask the computer to add `y` and `z`, but we don't ask it to do anything with the result.

Almost always, expressions have one of two forms: one form is an operator that changes a variable's value, like the assignment operator (`x = y + z;`), the addition assignment operator `+=`, or the the increment operator `++`. The other form of expression that we see as a statement is a function call, like a statement that simply calls the `printf()` function.

3.6. `if...else if...else`

```
if (x < 0) {
  printf("negative");
} else if (x > 0) {
  printf("positive");
} else {
  printf("zero");
}
```

The `if` statement works similarly to the `if` statement in other languages.

In C, the `if` statement's condition must be enclosed in parentheses and the body must have a set of braces enclosing it (unless the body consists of one statement only).

As we've already seen, C does not have an `elif` clause as in Python; instead `else if` can be used.

3.7. `return`

```
return 0;
```

Yhe `return` statement exits a function with a given return value. The value returned must be consistent with the return type of the function.

For a function with no return value, and a `void` return type, we can write:

```
return;
```

3.8. `while`

```
while (i >= 0) {
  printf("%d\n", i);
  i--;
}
```

C's `while` statement is comparable to that provided in C++, Java, Fortran and Python. While the condition is true, the loop executes.

In C, the `while` statement's condition must be enclosed in parentheses and the body must have a set of braces enclosing it (unless the body consists of one statement only).

3.9. `for`

C's `for` loop is comparable to that provided in C++, Java and FORTRAN. However, it differs in both purpose and syntax from the `for` statement provided by Python.

In C, the `for` keyword is followed by a set of parentheses containing three parts separated by semicolons:

```
for (init; test; update)
```

C's `for` loop enables stepping a variable through a series of numbers, like counting from 0 to 9. The part before the first semicolon (*init*) is performed as soon as the `for` statement is executed; it is for initializing the counting variable. The part between the two semicolons (*test*) is evaluated before each iteration to determine whether the iteration should be repeated. And the part following the final semicolon (*update*) is evaluated at the end of each iteration to update the counting variable for the following iteration.

In practice, `for` loops are used most often for counting out *n* iterations. The standard idiom for this is the following:

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

Here we have a counter variable `i` whose value starts at 0. With each iteration, we test whether `i` has reached *n* or not; and if it hasn't, then we execute the `for` statement's body and then perform the `i++` update so that `i` goes to the following integer. The result is that the body is executed for each value of `i` from 0 up to *n - 1*.

We can use a `for` loop for other purposes, too. In the following example, we display the powers of 2 up to 512. Notice how the update portion of the `for` statement has changed to `p *= 2`:

```
for (p = 1; p <= 512; p *= 2) {
    printf("%d\n", p);
}
```

The values of p on successive iterations is 1, 2, 4, 8, ..., 512.

`for` loops can also be used for counting down too. For example:

```
for (i = 10; i >= 0; i--) {
  printf("%d\n", i);
}
```

3.10. `break;`

C's `break` statement is comparable to that provided in C++, Java and Python, and FORTRAN's `exit` statement. The `break` statement immediately exits the innermost `while` or `for` loop in which it is found.

3.11. `continue;`

C's `continue` statement is comparable to that provided in C++, Java and Python, and FORTRAN. The `continue` statement skips to the bottom of the innermost `while` or `for` in which it is found and tests whether to repeat the loop again.

3.12. `switch`

C's `switch` statement is comparable to that provided in C++, Java and FORTRAN's `select case` statement. There is no equivalent statement in Python.

C's `switch` statement is equivalent to a particular form of an `if...else if...else if...else` statement where each of the tests are for different values of the same variable.

A `switch` statement is useful when we have several possible blocks of code, one of which should be executed based on the value of a particular expression. Here is an example instance of the `switch` statement:

```
switch (letter_grade) {
case 'A':
  gpa += 4;
  credits += 1;
  break;
case 'B':
  gpa += 3;
  credits += 1;
  break;
case 'C':
  gpa += 2;
  credits += 1;
  break;
case 'D':
  gpa += 1;
  credits += 1;
  break;
case 'W':
  break;
default:
  credits += 1;
}
```

Inside the parentheses following the `switch` keyword, we have an expression, whose value must be a character or integer. The computer evaluates this expression and goes down to one of the `case` keywords based on the value of the expression. If the value is the character `A`, then the first block is executed (`gpa += 4; credits += 1;`); if it is `B`, then the second block is executed; if it is none of the characters (such as an `F`), the block following the `default` keyword is executed.

The `break` statement at the end of each block is a crucial detail: if the `break` statement is omitted, then the computer continues into the following block. In our above example, if we omitted all `break` statements, then a grade of `A` would lead the computer to execute not only the `A` case but also the `B`, `C`, `D`, `W`, and `default` cases. The result would be that `gpa` would increase by 4 + 3 + 2 + 1 = 10, while `credits` would increase by 5. Occasionally we actually want the computer to continue to the next case (called "fall-through"), and so we omit a `break` statement; but in practice we almost always want a `break` statement at the end of each case.

There is one important exception where fall-through is quite common: sometimes we want the same code to apply to two different values. For instance, if we wanted the nothing to happen whether the grade is `P` or `W`, then we could include `case 'P':` just before `case 'W':`, with no intervening code:

```
case 'P':
case 'W':
  break;
```

3.13. Arrays

C provides an **array** type, comparable to arrays in C++, Fortran and Java. C arrays are also similar to Python lists except that, unlike lists, arrays cannot grow or shrink, their size is fixed at the time of creation.

We can declare and access an array by specifying the type of the array, the name of the array and then, in square brackets, `[` and `]`, the size of the array. For example:

```
double population[50];

population[0] = 897934;
population[1] = population[0] + 11804445;
```

In this example, we create an array for 50 `double` values. Each element of the array is indexed 0 through 49. This is termed **zero-based numbering** or **zero-based indexing**. The first element is accessed using index 0, the second using index 1, to the *N*th using index *N - 1* (for the above array, the 50th element is accessed via index 49). This is in contrast to FORTRAN which uses one-based numbering, the first element has index 1.

C does not have support for accessing the length of an array once it is created; that is, there is nothing analogous to Python's `len(population)` or Java's `population.length`, for example. It is up to us, as developers, to ensure our C programs know the length of our arrays.

An important point with respect to arrays is what happens if we access an array index outside the array, for example, `population[50]` or `population[-100]`? In Python or Java, this would terminate the program with a message pointing to the line at fault and saying that the program went beyond the array bounds. C is not nearly so friendly. When we request access beyond an array's bounds, C does it.

For example, consider the following program:

```
#include <stdio.h>

int main() {
  int i;
  int vals[5];

  for (i = 0; i <= 5; i++) {
    vals[i] = 0;
  }
  printf("%d\n", i);
  return 0;
}
```

Some systems can place `i` in memory just after the `vals` array; thus, when `i` reaches 5 and the computer executes `vals[i] = 0`. This, then, resets the memory corresponding to `i` to 0. As a result, this resets the `for` loop, and the program goes through the loop again, and again, repeatedly. The program never reaches the `printf` function call, and the program never terminates.

In more complicated programs, the lack of array-bounds checking can lead to very difficult bugs, where a variable's value changes mysteriously somewhere within hundreds of functions, and we as the programmer must determine where an array index was accessed out of bounds. This is the type of bug that takes a lot of time to uncover and repair.

That's why we should consider the error messages provided by Python or Java as extraordinarily friendly: not only does they tell us the cause of a problem, they even tell us exactly which line of the program was at fault. This can save a lot of debugging time.

Every once in a while, we'll see a C program crash, with a message like "segmentation fault" or "bus error". The message will not helpfully include any indication of what part of the program is at fault: all we get is those those two words. Such errors usually mean that the program attempted to access an invalid memory location. This may indicate an attempt to access an invalid array index, but typically the index needs to be pretty far out of bounds for this to occur.

As in C++, Java and FORTRAN, we can also define 2D arrays, 3D arrays etc. For example:

```
int landscape[5][4];
double temperature[5][4][2];
int i;
int j;
int k;

for (i = 0; i < 5; i++) {
  for (j = 0; j < 4; j++) {
    landscape[i][j] = 0;
    for (k = 0; k < 2; k++)
    {
      temperature[i][j][k]= 0.0;
    }
  }
}
```

3.14. Arrays and strings

We can define a string in C as follows:

```
char greeting1[12] = "Hello World!";
```

This creates an array of 12 characters.

We can then use this, for example:

```
printf("%s\n", greeting);
```

Running this gives:

```
Hello World!
```

We can also change the value of each element in the array. For example, to change `!` to `?`:

```
greeting[11] = '?';
printf("%s\n", greeting);
```

Running this gives:

```
Hello World?
```

However, we cannot reassign the string value in one go. So, for example, the following is not permitted in C:

```
greeting = "Hi";
```

We return to this in section 6.5.

3.15. Comments

In C's original design, all comments begin with a slash followed by an asterisk, `/*`, and end with an asterisk followed by a slash, `*/`. The comment can span multiple lines.

```
/*
 * gcd - returns the greatest common
 * divisor of its two parameters
 */
int gcd(int a, int b) {
```

The asterisk on the last comment line is ignored by the compiler. Most programmers would include it, though, both because it looks prettier and also because it indicates to a human reader that the comment is being continued from the previous line.

Though this multi-line comment was the only comment originally included with C, C++ introduced a single-line comment that has proven so handy that most of today's C compilers also support it. It starts with two slash characters, `//`, and goes to the end of the line.

```
int gcd(int a, int b) {
  if (b == 0) {
    return a;
  } else {
    // recurse if b != 0
```

```
    return gcd(b, a % b);
  }
}
```

---

4. Libraries

Having discussed the internals to functions, we now turn to discussing issues surrounding functions and separating a program into various files.

4.1. Function prototypes

In C, a function must be declared above the location where we use it. In the C program earlier, we defined the `gcd()` function first, then the `main()` function. This is significant: if we swapped the `gcd()` and `main()` functions around, the compiler would complain in `main()` that the `gcd()` function is undeclared. The is because C assumes that a compiler reads a program from the top down: by the time it gets to `main()`, it hasn't been told about a `gcd()` function, and so it believes that no such function exists.

This creates a problem, especially in larger programs that span several files, where functions in one file will need to call functions in another. To get around this, C provides the notion of a **function prototype**, where we write the function header but omit the body definition.

As an example, say we want to break our C program into two files: the first file, `gcd.c`, will contain the `gcd()` function, and the second file, `example.c`, will contain the `main()` function.

gcd.c:

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

example.c:

```
#include <stdio.h>

int main() {
  printf("GCD: %d\n", gcd(24, 40));
  return 0;
}
```

The problem with this is that, in compiling `example.c`, the compiler won't know about the `gcd()` function that it is attempting to call. A solution is to include a function prototype in `example.c`:

```c
#include <stdio.h>

int gcd(int a, int b);

int main() {
  printf("GCD: %d\n", gcd(24, 40));
  return 0;
}
```

The `int gcd...` line is the function prototype. It begins the same as a function definition begins, but we put a semicolon where the body of the function would normally be. By doing this, we are declaring that the function will eventually be defined, but we are not defining it yet. The compiler accepts this and obediently compiles the program with no complaints.

In this example, our source code is now in two files, `gcd.c` and `example.c`. When compiling the code we need to tell the compiler to compile both these files. For example:

```
$ gcc gcd.c example.c
$ ./a.out
GCD: 8
```

4.2. Header files

Larger programs spanning several files frequently contain many functions that are used many times in many different files. It would be painful to repeat every function prototype in every file that happens to use the function. So we instead create a file, called a **header file**, that contains each prototype written just once (and possibly some additional shared information), and then we can refer to this header file in each source file that wants the prototypes. The file of prototypes is called a header file, since it contains the "heads" of several functions. Conventionally, header files use the `.h` prefix, rather than the `.c` prefix used for C source files.

For example, we might put the prototype for our `gcd()` function into a header file called `gcd.h`:

```c
int gcd(int a, int b);
```

We can use a special type of line starting with `#include` to incorporate this header file at the top of `main.c`:

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

int main() {
  printf("GCD: %d\n", gcd(24, 40));
  return 0;
}
```

By convention we would also add this to `gcd.c`, as this can help to make it clear that this file holds the implementation of the function prototype:

```
#include "gcd.h"

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

To compile our program we would run:

```
$ gcc gcd.c example.c
$ ./a.out
GCD: 8
```

Note that we have not told C where the header file `gcd.h` is. By default it will look in the same directory as the file that runs the `#include` statement, that is, the directory that also holds `main.c`.

This is a small example, but imagine a library consisting of dozens of functions, which is used in dozens of files: suddenly the time savings of having just a single prototype for each function in a header file begins making sense.

The `#include` line is an example of a directive for C's **preprocessor**, through which the C compiler sends each program before actually compiling it. A program can contain commands (**directives**) telling the preprocessor to manipulate the program text that the compiler actually processes. The `#include` directive tells the preprocessor to replace the `#include` line with the contents of the file specified.

Header files can also include other header files.

In the same way that related functions can be grouped into classes and packages in Java, and related functions grouped into classes or modules in Python, we can we can group functions into related groups defined in header files.

4.3. Standard header files

We have already been using `#include` in the form of the `#include <stdio.h>` line. This allows us to use the `printf` function, whose prototype is defined in `stdio.h`, one of C's standard libraries.

Notice that `stdio.h` is in angle brackets, while oour `gcd.h` was in double quotation marks. The angle brackets are for the header files of these standard libraries. The quotation marks are for custom-written header files that can be found in the same directory as the source files.

4.4. Constants

Another particularly useful preprocessor directive is the `#define` directive. It tells the preprocessor to substitute all future occurrences of some word with something else.

```
#define PI 3.14159
```

In this fragment, we have told the preprocessor that, for the rest of the program, it should replace every occurrence of `PI` with `3.14159` instead. Suppose that later in the program is the following line:

```
printf("area: %f\n", PI * r * r);
```

Seeing this, the preprocessor would translate it into the following text for the C compiler to process:

```
printf("area: %f\n", 3.14159 * r * r);
```

This replacement happens behind the scenes, so we don't see the replacement.

The `#define` directive is not restricted to defining constants like this, though. Because it uses textual replacement only, the directive can be used (and abused) in other ways. For example, one might include the following.

```
#define forever while(1)
```

Subsequently, we could use `forever` as if it were a loop construct, and the preprocessor would replace it with `while(1)`.

```
forever {
    printf("hello world\n");
}
```

However, expert C programmers would consider this very poor style, since it quickly leads to unreadable programs.

---

5. Building C programs revisited

There are many C compilers available. Here, we focus on the GNU C Compiler (`gcc`) which is a popular C compiler.

C source code files (with `.c` extension) can be **compiled** as follows, where the `-c` flag means "compile only":

```
$ gcc -c <FILE>.c <FILE>.c <FILE>.c ...
```

For example:

```
$ gcc -c gcd.c example.c
```

The compiler converts each source code file into a collection of machine language statements, or object code. Each source code file has a corresponding object file (with `.o` extension) each of which has machine language statements for that source code. For example:

```
$ ls
example.o gcd.o
```

These object files can then be **linked** together into an executable program. This can be done as follows:

```
$ gcc <FILE>.o <FILE>.o <FILE>.o ...
```

For example:

```
$ gcc gcd.o example.o
```

This produces an executable program called `a.out`. This can then be run as follows:

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

As `a.out` is not a particularly meaningful name, the name of the executable can be specified via a `-o` flag when linking. For example:

```
$ gcc -o gcd gcd.o example.o
```

This produces an executable called `gcd`, which can then be run:

```
$ ./gcd
GCD: 8
```

The commands to compile and link can be combined into a single command as follows:

```
$ gcc -o <EXECUTABLE> <FILE>.c <FILE>.c <FILE>.c ...
```

For example:

```
$ gcc -o gcd gcd.c example.c
```

If the program needs other libraries to be linked also, then these libraries can be specified using `-l<LIBRARY_NAME>` flags. For example to link together files and add the C math library:

```
$ gcc <FILE>.o <FILE>.o <FILE>.o ... -lm
```

where `m` is the name of the C math library.

For example:

```
$ gcc -o analyse arralloc.c utils.c analyser.c -lm
```

For more information on the `gcc` compiler and its command-line options, check out its manual (`man`) page:

```
man gcc
```

---

6. Pointers

Variables can be viewed as a human-readable names for areas, or addresses, in memory which holds the values of the variables. Here `sample` is our human-readable name for an area, an address, in memory which holds the `int` value 123:

```
int sample;

sample = 123;
printf("sample's value: %d\n", sample);
```

We can print the address of our variable in memory as follows:

```
printf("sample's address: %x\n", &sample);
```

`&` tells the computer to get the address in memory of the variable `sample`.

Here is an example of running this example twice:

```
$ ./a.out
sample's value: 123
sample's address: ed98b040
$ ./a.out
sample's value: 123
sample's address: 9d356cc0
```

Note how the address is different. In the second run of the program, the computer has provided a different location in memory for the variable.

In C we can define variables that are **pointers**. A pointer is a variable whose value is an address in memory where a value can be found. We can declare a variable as a pointer, and assign it, as follows:

```
int* sample_ptr;

sample_ptr = &sample;
```

Here, `int*` states that we are declaring a pointer to a variable of type `int`, and our pointer variable is called `sample_ptr`.

`&` tells the computer to get the address in memory of the variable `sample`. This address is assigned to `sample_ptr`. `sample_ptr` now holds the address in memory which corresponds to `sample`. For example, if we add the statement:

```
printf("sample's address in sample_ptr: %x\n", sample_ptr);
```

This statement prints the value of `sample_ptr`. This is the address in memory of the variable `sample`. If we run our code, we would see something like:

```
sample's address in sample_ptr: ddbf9be4
```

6.1. Dereferencing pointers

How do we get the value itself, via the pointer? To do that we **dereference** the pointer. For example:

```
int sample_via_ptr;

sample_via_ptr = *sample_ptr;
printf("sample's value via dereferencing sample_ptr: %d\n", sample_via_ptr);
```

`*sample_ptr` is how we dereference the pointer `sample_ptr`. This is an instruction to the computer to go to the address pointed at by `sample_ptr` and get the value that is there. As the address held by `sample_ptr` is that of the variable `sample` and the value at that address is `sample`'s value, the value 123 would be printed. For example:

```
sample's value via dereferencing sample_ptr: 123
```

We can use dereferencing not just to get the value at the address pointed at by the pointer but to update that value. For example, here we dereference `sample_ptr` and update the value at the address it points to, i.e. the value of `sample`. We print both the value accessed via dereferencing `sample_ptr` and the value of `sample` itself to show that the value of `sample` was indeed updated:

```
*sample_ptr = 456;
printf("sample's value via dereferencing sample_ptr: %d\n", *sample_ptr);
printf("sample's value: %d\n", sample);
```

If we run this code we would see:

```
sample's value via dereferencing sample_ptr: 123
sample's value via dereferencing sample_ptr: 456
sample's value: 456
```

Note that the symbol * has two roles, depending upon its context. It is used to declare a pointer in a variable declaration. It is also used to dereference a pointer. In fact, * has another role, as it can also denote multiplication.

6.2. `NULL` pointers

Until a pointer variable is assigned it may have no address or point to a random part of memory. It can be useful to explicitly set points as **null pointers** until they are assigned a value. We can use the value 0 to denote a null pointer. Alternatively, C provides a C constant `NULL` to represent a null pointer. To use this constant we need to include a standard C library header (for example, `stddef.h`, `stdlib.h`, or, as we've used it already, `stdio.h`). We can then use this constant to both set a new pointer to be a null pointer and to check whether a pointer is null, for example:

```
int* some_ptr;

some_ptr = NULL;
printf("some_ptr: %x\n", some_ptr);
if (some_ptr == NULL) {
  printf("some_ptr is NULL\n");
} else {
  printf("some_ptr is not NULL\n");
}
```

```
some_ptr = &sample;
printf("some_ptr: %x\n", some_ptr);
if (some_ptr == NULL) {
  printf("some_ptr is NULL\n");
} else {
  printf("some_ptr is not NULL\n");
}
```

If we run this code we would see:

```
some_ptr: 0
some_ptr is NULL
some_ptr: 5b265124
some_ptr is not NULL
```

Note that `NULL` has value 0.

Every once in a while, we'll see a C program crash, with a message like "segmentation fault" or "bus error". This can indicate that we have attempted to dereference a null pointer.

6.3. Pointers are variables

Pointers are variables so we can use them in assignments. Here we create another pointer, `sample_ptr_copy` and assign it the value of `sample_ptr`. As the value of `sample_ptr` is an address, the address of `sample`, so `sample_ptr_copy` will be given this same value. We can then dereference `sample_ptr_copy` to print the value of `sample`:

```
int* sample_ptr_copy;

sample_ptr_copy = sample_ptr;
printf("sample's value via dereferencing sample_ptr_copy: %d\n",
       *sample_ptr_copy);
```

6.4. Pointers to pointers

Pointers are variables which store addresses. We can create a pointer to a pointer. For example:

```
int** sample_ptr_ptr;

sample_ptr_ptr = &sample_ptr;
printf("sample_ptr's address in sample_ptr_ptr: %x\n", sample_ptr_ptr);
printf("Dereference sample_ptr_ptr: %x\n", *sample_ptr_ptr);
printf("Dereference sample_ptr_ptr's value: %d\n", **sample_ptr_ptr);
```

Here `sample_ptr_ptr` is a pointer to a pointer of type `int`. That is, it contains an address in memory at which can be found another address in memory at which a value of type `int` can be found.

If we run this code we would see something like (assuming `sample` has value 456):

```
sample_ptr's address in sample_ptr_ptr: 45f16338
Dereference sample_ptr_ptr: 45f16344
Dereference sample_ptr_ptr's value: 456
```

6.5. Pointers and strings

We saw in section 3.14 that we can define strings using arrays. For example:

```
char greeting[12] = "Hello World!";
```

One limitation of this was that we could not assign a new string to the variable, `greeting`.

An alternative way of defining strings is to use pointers. For example:

```
char* greeting = "Hello World!";

printf("%s\n", greeting);
```

This creates a variable, `greeting`, which is a pointer to an area in memory with a value of type `char` (which holds the letter H), followed by consecutive blocks of memory for each of the remaining letters in the string.

One advantage of using `char*` for strings is that the variable can be reassigned. For example:

```
greeting = "Hi there!";
printf("%s\n", greeting);
```

6.6. Pointers and arrays

Following on from our previous example, consider what happens when we run this code:

```
int i;
for (i = 0; i < 9; i++) {
  printf("%c\n", greeting[i]);
}
```

The following is printed:

```
H
i

t
h
e
r
e
!
```

Because our pointer points to a block of memory with the characters of the string, we can access the successive values using the array syntax.

6.7. Passing pointers to functions

Pointers can be passed to functions in the same way as other parameters.

For example, if we had a function that takes a two integers, a character and a 2D integer array, we could define:

```
void some_function(int a, int b, char* file, int** map){
    ...
}
```

---

7. Dynamic memory allocation

In section 3.13. on arrays, we had:

```
double population[50];

population[0] = 897934;
population[1] = population[0] + 11804445;
```

This assumes we know how big our array is when we write out code. Suppose the size of the array was dependant upon a variable that is not known until runtime? We could take a guess and define an array as large as we might need, but this is wasteful in memory. We want to dynamically create our array.

We can ask C to perform runtime **memory allocation** for us, that is, dynamically allocate memory of the required size at runtime. For example, assume that `op_size` holds the size of the population, we can do:

```
double* population;

population = (double*)malloc(pop_size * sizeof(double));
if (population == NULL) {
    printf("Could not allocate memory for population.\n");
```

```
    exit(0);
}
```

We define our array type `double*`, a pointer to a value of type `double`. This variable will hold a pointer to the first element of our array.

`malloc` is a C function that will dynamically allocate a block of memory of a specific size. It takes one argument, the size of the space it needs to allocate. This is a factor both of the desired size of our array (`pop_size`) and the size, in memory, needed by each element of our array. As our array is an array of `double` we use the C function `sizeof`, with argument `double` to calculate the size in memory for each element.

`malloc` returns a pointer of type `void*` which we cast to the pointer of the type of our array i.e. `double*` (the operator `(double*)` does the cast). We assign this to `population` so that `population` holds the address, in memory, of the first element of our array.

If there is no memory available for the array then `malloc` fails and will return a `NULL` pointer, When using `malloc` it is important to check for the `NULL` pointer.

Once allocated, we can then use our array, for example:

```
for (i = 0; i < pop_size; i++) {
  population[i] = 0.0;
}
```

7.1. Freeing memory

When we dynamically allocate memory, the computer uses space on its **heap**, an area of memory available for programs to use. It is important to free up dynamically allocated memory when it is no longer needed otherwise we can run out of heap, both for our program, when running, or other programs running at the same time as ours.

If we were to run a program with the above code, using the `valgrind` tool, which monitors memory use, we'd see something like:

```
$ valgrind --leak-check=summary ./a.out
...
==11526== HEAP SUMMARY:
==11526==     in use at exit: 80 bytes in 1 blocks
==11526==   total heap usage: 1 allocs, 0 frees, 80 bytes allocated
==11526==
==11526== LEAK SUMMARY:
==11526==    definitely lost: 80 bytes in 1 blocks
==11526==    indirectly lost: 0 bytes in 0 blocks
==11526==      possibly lost: 0 bytes in 0 blocks
==11526==    still reachable: 0 bytes in 0 blocks
==11526==         suppressed: 0 bytes in 0 blocks
==11526== Rerun with --leak-check=full to see details of leaked memory
==11526==
```

```
==11526== For counts of detected and suppressed errors, rerun with: -v
==11526== Use --track-origins=yes to see where uninitialised values come from
==11526== ERROR SUMMARY: 490 errors from 19 contexts (suppressed: 0 from 0)
```

It warns us that we have a memory leak of `80 bytes` - we requested C dynamically allocate 80 bytes of memory but we did not free them up when we were done with them. This exactly corresponds to our array of `double` values: each `double` value is 64 bits, or 8 bytes, and our array has size 10, so our total array mneeded 80 bytes.

We can free up dynamically allocated memory using the `free` command, for example:

```
free(population);
```

If we were to add this line and then rerun `valgrind`, we'd see something like:

```
$ valgrind --leak-check=summary ./a.out
...
==11549== HEAP SUMMARY:
==11549==     in use at exit: 0 bytes in 0 blocks
==11549==   total heap usage: 1 allocs, 1 frees, 80 bytes allocated
==11549==
==11549== All heap blocks were freed -- no leaks are possible
==11549==
==11549== For counts of detected and suppressed errors, rerun with: -v
==11549== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Note that we now have no memory leaks, all the memory we requested for our program was then freed by our program/

The need to explicity dynamically allocate memory and then free this dynamically allocated memory is is in contrast to languages such as Java or Python. Java and Python take care of a lot of memory management for us. In particular, they regularly traverse the heap and identify memory on the heap that is no longer being used and free it up automatically, a process called **garbage collection**.

7.2. Allocating memory for arrays and EPCC's `arralloc` function

EPCC has written an `arralloc` function to make assigning memory for dynamic arrays of any dimension easier. Its function prototype (defined in an `arralloc.h` file) is:

```
void *arralloc(size_t size, int ndim, ...);
```

This can allocate memory for an `ndmin` array of any type. The `...` is C **ellipsis** which represents any number of variables.

To allocate a 1D `int` array of dimension 10, call:

```
array = (int*)arralloc(sizeof(int), 1, 10);
```

To allocate a 2D `int` array of dimensions 10x12, call:

```
array = (int**)arralloc(sizeof(int), 2, 10, 12);
```

To allocate a 3D `double` array of dimensions 10x12x5, call:

```
array = (double***)arralloc(sizeof(double), 3, 10, 12, 5);
```

To free up memory once finished with our array, we can use `free`. For example for the above:

```
free(array)
```

---

8. User-defined data types

C allows user-defined data types to be declared using the `struct` keyword. For example:

```
#include <stdio.h>

struct student
{
   char* name;
   int marks;
};
```

Each `struct` has a name, plus one or more fields, each of which has a name and a type.

We can declare variables of our `struct` types as follows:

```
struct student fred;
struct student jo;
```

We can access and update the fields of a `struct` using the syntax `<STRUCT_VARIABLE>.<FIELD>`. For example:

```
fred.name = "fred";
fred.marks = 1;
```

```
jo.name = "jo";
jo.marks = 2;

printf("Student: %s: %d\n", jo.name, jo.marks);
```

`struct` data types are often defined in header files, for the same reason that we define function prototypes in header files, so that they can be defined in one place and used to compile whatever code requires them.

8.1. Passing `structs` to functions

To use a `struct` as a parameter within a function we need to remember the `struct` keyword and the `struct` name, for example:

```
void print_student(struct student s) {
  printf("Student: %s: %d\n", s.name, s.marks);
  return;
}
```

A `struct` parameter is **passed by value** i.e. a copy of the `struct` is passed to the function. Any changes to the `struct` within the function will have no effect on the `struct` in the caller. For example, if we had the function:

```
void set_marks(struct student s, int marks) {
  s.marks = marks;
  return;
}
```

and we called:

```
print_student(jo);
set_marks(jo, 10);
print_student(jo);
```

we would see:

```
Student: Jo: 2
Student: Jo: 2
```

If we want to change the value of the fields of the `struct` that we pass to the function we need to ensure that the `struct` parameter is **passed by reference**. We do this by declaring that the function expects a pointer to a `struct`:

```
void set_marks(struct student* s, int marks) {
  (*s).marks = marks;
  return;
}
```

As we're using a pointer, we have to, in the function, defererence the pointer to the `struct`, then update the value of the field.

When calling the function, we need to pass the address of the `struct`:

```
set_marks(&jo, 10);
```

Running our code would now give:

```
Student: Jo: 2
Student: Jo: 10
```

The notation to dereference the pointer to the `struct` then access the field is a bit cumbersome:

```
(*s).marks = marks;
```

C allows a shorthand to be used, `->`, so the above can be replaced by:

```
s->marks = marks;
```

8.2. Dynamically allocating memory for arrays of `struct`

We can dynamically allocate memory for arrays of `struct` using `malloc` or `arralloc` as already shown. For example:

```
students = (struct student*)malloc(10 * sizeof(struct student));

students_classes = (struct student*)arralloc(sizeof(struct student),
                                             2, 10, 20);
```

---

9. File output

As for C++, FORTRAN, Java and Python, there are many ways in which files can be created and written to. A simple example is as follows based on C's stdio library. To use this library, the following needs to be defined in

a C source file or header file included by that source file:

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

C's `fopen` function can take a file name and whether the file is to be opened for reading (`"r"`) or writing (`"w"`) and returns a pointer to the file. For example, the following opens a file, whose name is in the variable `datafile`, for writing:

```
FILE *fp;
fp = fopen(datafile, "w");
```

`fp` holds a file pointer of type `FILE`, which is a type defined in `stdio.h`. This is a pointer to the open file, which we can use when performing operations on the file.

Data can be printed into the file using C's `fprintf` function. This is similar to `printf` in that it takes a format string and parameters. It also, however, takes as its first argument, a pointer to an open file. For example:

```
fprintf(fp, "# Sample data file\n");
fprintf(fp, "%4d %4d\n", 1, 2);
fprintf(fp, "%4f %4d\n", 3.4, 5.6);
```

would write the content:

```
# Sample data file
   1    2
 3.4  5.6
```

When the data has been written, the file can be closed using the `fclose` function, with the file pointer as an argument:

```
fclose(fp);
```

Despite `fp` being a function pointer we don't need to call `free` as `fclose` cleans up for us.

---

10. Miscellaneous

10.1. Command-line arguments

It is useful to be able to pass arguments into our programs from the command-line. A simple way to access arguments on the command-line is to change the signature of our `main` method to:

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

If we define a `main` method this way then, when our program is run, the number of command-line arguments will be passed in as the first parameter, and the command-line arguments themselves as the second parameter, an array of `char*` strings. For example, suppose our program is:

```
#include <stdio.h>

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

  int i;

  printf("Number of arguments: %d\n", argc);
  for (i = 0; i < argc; i++) {
    printf("Argument %d: %s\n", i, argv[i]);
  }
  return 0;
}
```

We can run this on various examples to see what happens:

```
$ ./a.out
Number of arguments: 1
Argument 0: ./a.out

$ ./a.out test sample
Number of arguments: 2
Argument 0: ./a.out
Argument 1: test

$ ./a.out test sample
Number of arguments: 3
Argument 0: ./a.out
Argument 1: test
Argument 2: sample

$ ./a.out test sample 123 1.2
Number of arguments: 5
Argument 0: ./a.out
Argument 1: test
Argument 2: sample
Argument 3: 123
Argument 4: 1.2
```

In each case the first argument, `argv[0]`, is always the name of our executable (in this case `a.out`).

C provides functions `atoi` and `atof` to parse `char*` values into `int` and `float` respectively. The C standard library `stdlib.h` includes a version of `atof` which can also handle `char*` to `double`. These then give us the building blocks to get values from the command-line into our code. For example, if we have:

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

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

  char* file_name;
  file_name = argv[1];
  char* tag;
  tag = argv[2];
  int count;
  count = atoi(argv[3]);
  double ratio;
  ratio = atof(argv[4]);
  printf("File name: %s\n", file_name);
  printf("Tag: %s\n", tag);
  printf("Count: %d\n", count);
  printf("Ratio: %f\n", ratio);

  return 0;
}
```

We can run this program to see what happens:

```
$ ./a.out sample.dat test 123 4.5678
File name: sample.dat
Tag: test
Count: 123
Ratio: 4.567800
```

There are more advanced command-line processing libraries you may also want to investigate. Here is an example program that uses `getopt`:

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

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

  char* file_name;
  char* tag;
  int count;
  double ratio;
  int opt;
  while((opt = getopt(argc, argv, ":vpf:t:c:r:")) != -1)
```

```
    {
      switch(opt)
        {
    case 'v':
    case 'p':
      printf("Option: %c\n", opt);
      break;
    case 'f':
      printf("Option and argument: %c: %s\n", opt, optarg);
      file_name = optarg;
      break;
    case 't':
      printf("Option and argument: %c: %s\n", opt, optarg);
      tag = optarg;
      break;
    case 'c':
      printf("Option and argument: %c: %s\n", opt, optarg);
      count = atoi(optarg);
      break;
    case 'r':
      printf("Option and argument: %c: %s\n", opt, optarg);
      ratio = atof(optarg);
      break;
    case ':':
      printf("Missing value\n");
      break;
    case '?':
      printf("Unknown option: %c\n", optopt);
    break;
        }
    }
    for(; optind < argc; optind++){
        printf("Extra argument: %s\n", argv[optind]);
    }

    printf("File name: %s\n", file_name);
    printf("Tag: %s\n", tag);
    printf("Count: %d\n", count);
    printf("Ratio: %f\n", ratio);
    return 0;
}
```

The string `:vpf:t:c:r:` determines the command-line options that are expected, each of which is a single letter. The `:` after an option letter states that there should be an associated argument with the option. So, our options string states that our program expects any of the following single-letter options:

- `-v`
- `-p`
- `-f <ARGUMENT>`
- `-t <ARGUMENT>`
- `-c <ARGUMENT>`
- `-r <ARGUMENT>`

We can run this program to see what happens:

```
$ ./a.out -v -f sample.dat -t test -c 123 -r 4.5678 other stuff
Option: v
Option and argument: f: sample.dat
Option and argument: t: test
Option and argument: c: 123
Option and argument: r: 4.5678
Extra argument: other
Extra argument: stuff
File name: sample.dat
Tag: test
Count: 123
Ratio: 4.567800
```

For more information on `getopt`, see GNU's Parsing program options using getopt.

For another command-line processing library, see Parsing Program Options with Argp

10.2. Timing

As for C++, FORTRAN, Java and Python, C has functions to get dates and times, in its standard library `sys/time.h`. To use this library, the following needs to be defined in a C source file or header file included by that source file (we also include `stddef.h` as we'll need the `NULL` pointer constant):

```
#include <stddef.h>
#include <sys/time.h>
```

The library provides a `struct` called `timeval` that represents a time value in seconds and microseconds.

The library also provides a function, `gettimeofday` which sets the current time in seconds and microseconds. The number of seconds is calculated from the date and time 1970-01-01 00:00:00 +0000 (UTC).

The following code creates a variable of type `struct timeval`, then uses `gettimeofday` to set the current time in seconds and micrseconds on this variable:

```
#include <stdio.h>
#include <sys/time.h>

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

  struct timeval current_time;

  gettimeofday(&current_time, NULL);
  printf("%d s %d us\n", current_time.tv_sec, current_time.tv_usec);

  return 0;
}
```

Running this program gives an output like:

```
1567000611 s 898745 us
```

If we convert the number of seconds into hours we get (((1567000611/60.0)/60)/24/365 = 49.68926341324201 i.e. just over 49 1/2 years since 1970-01-01 00:00:00.

---

11. Further information

Tutorials:

- C Programming's C Tutorial
- Tutorialspoint's C Programming Tutorial
- Learn-C.org

References:

- gcc(1) - Linux man page: online version of the `gcc` manual page.
- cppreference.com's C reference
- Tutorialspoint's C Standard Library

Books:

- Brian W. Kernighan and Dennis Ritchie (1988), "The C Programming Language (2nd Edition)", Prentice Hall, March 1988. ISBN-10: 0131103628, ISBN-13: 978-0131103627