C for Python programmers

by Carl Burch, Hendrix College, August 2011

Contents

1. Building a simple program
   1.1. Compilers versus interpreters
   1.2. Variable declarations
   1.3. Whitespace
   1.4. The `printf()` function
   1.5. Functions
2. Statement-level constructs
   2.1. Operators
   2.2. Basic types
   2.3. Braces
   2.4. Statements
   2.5. Arrays
   2.6. Comments
3. Libraries
   3.1. Function prototypes
   3.2. Header files
   3.3. Constants

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. Knowing C is in itself a good thing — it is an excellent starting point for relating more directly with what a computer does. But learning C is also a good starting point for becoming familiar with all these other languages.

This document is directed at people who have learned programming in Python and who wish to learn about C. 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). So learning Python is a good first step toward learning C.

1. Building a simple program

We'll start with several general principles, working toward a complete — but limited — C program by the end of Section 1.

1.1. Compilers versus interpreters

One major difference between C and Python is simply how you go about executing programs written in the two languages. With C programs, you usually use a *compiler* when you are ready to see a C program execute; by contrast, with Python, you typically use an *interpreter*. A **compiler** generates a file containing the translation of the program into the machine's native code. The compiler does not actually execute the program; instead, you first execute the compiler to create a native executable, and then you execute the generated executable. Thus, after creating a C program, executing it is a two step process.

```
me@computer:~$ gcc my_program.c
me@computer:~$ ./a.out
GCD: 8
```

In the first command ("`gcc my_program.c`"), we invoke the compiler, named *gcc*. The compiler reads the file `my_program.c`, into which we've saved our C code, and it generates a new file named `a.out` containing a translation of this code into the binary code used by the machine. In the second command ("`./a.out`"), we tell the computer to execute this binary code. As it is executing the program, the computer has no idea that `a.out` was just created from some C program: It is simply blindly executing the code found within the `a.out` file, just as it blindly executes the code found within the `gcc` file in response to the first command.

In contrast, an **interpreter** reads the user-written program and performs it directly. This removes a step from the execution process, but a compiler has the advantage that it generates an executable that runs just like most other applications on the machine, and it can conceivably lead to faster runtimes.

Being compiled has some radical implications to language design. C is designed so the compiler can tell everything it needs to know to translate the C program without actually executing the program.

1.2. Variable declarations

Among the ways that C requires programmers give information to the compiler, one of the most notable examples is that C requires **variable declarations**, informing the compiler about the variable before the variable is actually used. This is typical of many prominent programming languages, particularly among those intended to be compiled before executed.

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 we'll study later. Once you declare a variable to be of a particular type, you cannot change its type: If the variable `x` is declared of type `double`, and you assign "`x = 3;`", then `x` will actually hold the floating-point value 3.0 rather than the integer 3. You can, if you wish, imagine that `x` is a box that is capable only of holding `double` values; if you attempt to place something else into it (like the `int` value 3), the compiler converts it into a `double` so that it will fit.

Declaring a variable is simple enough: You enter the variable's type, some whitespace, the variable's name, and a semicolon:

```
double x;
```

In C, variable declarations belong at the top of the function in which they are used.

If you 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 such as "symbol undeclared."

To a Python programmer, it seems a pain to have to include these variable declarations in a program, though this gets easier with more practice. C programmers tend to feel variable declarations are worth the minor pain. The biggest 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.

1.3. Whitespace

In Python, whitespace characters like tabs and newlines are important: You separate your statements by placing them on separate lines, and you 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. (Admittedly, FORTRAN and BASIC also use 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. Most statements are terminated with a semicolon '`;`', and blocks of statements are indicated using a set of braces, '`{`' and '`}`'. Here's an example fragment from a C program, with its Python equivalent.

Figure 1: C fragment and Python equivalent

C fragment

```
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;
    }
}
```

Python equivalent

```
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 = disc ** 0.5 / a
        sol0 = (t0 + t1) / 2
        sol1 = (t0 - t1) / 2
```

The C program at left is how *I* would write it. However, whitespace is insignificant, so the computer would be just as happy if I had instead written the following.

```
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 sane human would prefer it. So any competent programmer tends to be very careful with 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`".)"

1.4. The `printf()` function

As we work toward writing useful C programs, one important ingredient is displaying results for the user to see, which you would accomplish using `print` in Python. In C, you use `printf()`

instead. This is actually a function, one of the most useful among C's library of language-defined functions.

The way the parameters to `printf()` work is a bit complicated but also quite convenient: The first parameter is a string specifying the format of what to print, and the following parameters indicate the values to print. The easiest way to understand this is to look at an example.

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

This line says 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()`: It says to print a value specified in a subsequent parameter. In this case, a lower-case *d* follows the percent character, indicating to display the parameter 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 (let's imagine `num_sol` is 2 in this example) and displays that value instead. It then continues through the format string, in this case displaying a newline character. Overall, then, the user sees the following line of output:

```
# solns: 2
```

Like Python, C allows you to include escape characters in a string using a backslash. The "`\n`" sequence represents the newline character — that is, the character that represents a line break. Similarly, "`\t`" represents the tab character, "`\"`" represents the double-quote character, and "`\\`" 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 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 illustrated below.

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

Note that `printf()` displays only what it is told, 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);
```

By the way, the `printf()` function displays "4.0" rather than simply "4" because the format string uses "`%f`", which says to interpret the parameters as floating-point numbers. If you want it to display just "4", you 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 my computer, replacing each "`%f`" with "`%d`" leads to the following output:

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

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

- `%d`, as we've already seen, says to print an `int` value in decimal form.
- `%x` says to print an `int` value in hexadecimal form.
- `%f` says to print a `double` value in decimal-point form.
- `%e` says to print a `double` value in scientific notation (for example, `3.000000e8`).
- `%c` says to print a `char` value.
- `%s` says to print a string. There's no variable type for representing a string, but C does support some string facilities using arrays of characters. We'll defer discussion of these facilities to later, after we discuss pointers, as strings involve some more complex concepts that we haven't seen yet.

You can also include a number between the percent character and the format descriptor as in "`%10d`", which tells `printf()` to right-justify a decimal integer over ten columns.

1.5. Functions

Unlike Python, all C code must be nested within functions, and functions cannot be nested within each other. Thus, a C program's overall structure is typically very straightforward: It 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 a simple example of a function definition:

```
double expon(double b, int e) {
    if (e == 0) {
        return 1.0;
    } else {
        return b * expon(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 (`expon`), 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 you nest the body of the function.

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

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 wants to execute the program. The integer return value is largely meaningless; we'll always return 0 rather than worrying about how the return value might be used.

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

Figure 2: A complete C program and a Python equivalent

C program

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

Python program

```
def gcd(a, b):
  if b == 0:
    return a
  else:
    return gcd(b, a % b)
```

Figure 2: A complete C program and a Python equivalent

C program

```
  }
}

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

Python program

```
print("GCD: " + str(gcd(24, 40)))
```

As you can see, 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 the program's `main` function, since this is where we put all top-level code that is to complete when the program is executed.

2. Statement-level constructs

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

2.1. Operators

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

Figure 3. Major operators in C and Python

C operator precedence

```
++ -- (postfix)
+ - ! (unary)
* / %
+ - (binary)
< > <= >=
== !=
&&
||
= += -= *= /= %=
```

Python operator precedence

```
**
+ - (unary)
* / % //
+ - (binary)
< > <= >= == !=
not
and
or
```

Some important distinctions:

- C does not have an exponentiation operator like Python's `**` operator. For exponentiation in C, you'd want to 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 you end up needing parentheses most times you want to use the `!` operator.
- C defines assignment as an operator, whereas 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.

```
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 extraordinarily bad style; others find it too convenient to avoid. Python, of course, 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`")."
- C's division operator `/` does integer division if both sides of the operator have an `int` type; that is, 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."

  By contrast, with 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.

2.2. Basic types

C's list of types is quite constrained.

`int`	for an integer
`char`	for a single character
`float`	for a single-precision floating-point number
`double`	for a double-precision floating-point number

The `int` type is straightforward. You can also 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 2.15×10^9. 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 is also straightforward: It represents a single character, like a letter or punctuation symbol. You can represent an individual character in a program by enclosing the character in single quotation marks: "`digit0 = '0';`" would place the zero digit character into the `char` variable `digit0`.

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 6.02×10^{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 (-3.4×10^{38} to 3.4×10^{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 (-1.7×10^{308} to 1.7×10^{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/false values. This has major implications for a statement like `if`, where you 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.

```
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`) turns out to be 5, which isn't 0 and thus the `if` condition succeeds.

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

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

This quirk — that C regards all non-zero integers as true — is generally regarded as a mistake. C introduced it machine languages rarely have direct support for Boolean values, but typically machine languages expect you 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. Most newer languages choose to have a special type associated with Boolean values. (Python has its own Boolean type, but it also treats 0 as false for `if` statements.)

2.3. 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 with no braces, since the body of both the `if` and the `else` contain only a single statement.

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

(We could also include braces on just one of the two bodies, as long as that body contains just one statement.)

C programmers use this quite often when they want one of several `if` tests to be executed. An example of this is with the quadratic formula code above. We could compute the number of solutions as follows:

```
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 `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;
    }
```

But 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.

```
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 you find in Python. You can just string together as many "`else if`" combinations as you want.

Other than this particular situation, I recommend that you include the braces anyway. As you continue working on a program, you often find that you want to add additional statements into the body of an `if`, and having the braces there already saves you 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.

2.4. Statements

We've seen four types of statements, three of which correlate closely with Python.

1. `int x;`

   We already discussed variable declarations in Section 1.2. They have no parallel in Python.

2. `x = y + z;` or `printf("%d", x);`

   You can have an expression as a statement. Technically, the expression could be "`x + 3;`", but such a statement has no point: We ask the computer to add `x` and 3, but we don't ask anything to happen to it. Almost always, the expressions have one of two forms: One form is an operator that changes a variable's value, like the assignment operator ("`x = 3;`"), the addition assignment operator `+=`, or the the increment operator `++`. The other form of expression that you see as a statement is a function call, like a statement that simply calls the `printf()` function.

3. `if (x < 0) { printf("negative"); }`

You can have an `if` statement, which works very similarly to Python's `if` statement. The only major difference is the syntax: In C, an `if` statement's condition must be enclosed in parentheses, there is no colon following the condition, and the body has a set of braces enclosing it.

As we've already seen, C does not have an `elif` clause as in Python; instead, C programmers use the optional-brace rule and write "`else if`".

4. `return 0;`

You can have a `return` statement to exit a function with a given return value. Or for a function with no return value (and a `void` return type), you would write simply "`return;`".

There are three more statement types that correlate closely to equivalents from Python.

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

The `while` statement works identically to Python's, although the syntax is different in the same way that the `if` syntax is different.

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

Again, the test expression requires a set of parentheses around it, there is no colon, and we use braces to surround the loop's body.

6. `break;`

As in Python, the `break` statement immediately exits the innermost loop in which it is found. Of course, the statement has a semicolon following it.

7. `continue;`

Also as in Python, the `continue` statement skips to the bottom of the innermost loop in which it is found and tests whether to repeat the loop again. It has a semicolon following it, too.

And there are two types of statements that have no good parallel in Python.

8. `for (i = 0; i < 10; i++) {...`

While Python also has a `for` statement, its purpose and its syntax bear scant similarity to C's `for` statement. In C, the `for` keyword is followed by a set of parentheses containing three parts separated by semicolons.

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

The intent of C's `for` loop is to enable 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 reached; it is for initializing the variable that will count. 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++) {
  body
}
```

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.

But you 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);
}
```

9. `switch (grade) { case 'A':...`

The `switch` statement has no equivalent in Python, but it is essentially equivalent to a particular form of an `if`...`elif`...`elif`...`else` statement where each of the tests are for different values of the same variable.

A `switch` statement is useful when you have several possible blocks of code, one of which should be executed based on the value of a particular expression. Here is a classic 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 its value. 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 (like 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 you actually want the computer to continue to the next case (called "fall-through"), and so you omit a `break` statement; but in practice you almost always want `break` statement at the end of each case.

There is one important exception where fall-through is quite common: Sometimes you 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.

2.5. Arrays

Python supports many types that combine the basic atomic types into a group: tuples, lists, strings, dictionaries, sets.

C's support is much more rudimentary: The *only* composite type is the **array**, which is similar to Python's list except that an array in C cannot grow or shrink — its size is fixed at the time of creation. You can declare and access an array as follows.

```
double pops[50];
pops[0] = 897934;
pops[1] = pops[0] + 11804445;
```

In this example, we create an array containing 50 slots for `double` values. The slots are indexed 0 through 49.

C does not have an support for accessing the length of an array once it is created; that is, there is nothing analogous to Python's `len(pops)` or Java's `pops.length`.

An important point with respect to arrays: What happens if you access an array index outside the array, like accessing `pops[50]` or `pops[-100]`? With Python or Java, this will terminate the program with a friendly message pointing to the line at fault and saying that the program went beyond the array bounds. C is not nearly so friendly. When you access beyond an array bounds, it blindly does it.

This can lead to peculiar behavior. For example, consider the following program.

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

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

Some systems (including a Linux installation I've encountered) would place `i` in memory just after the `vals` array; thus, when `i` reaches 5 and the computer executes "`vals[i] = 0`", it in fact resets the memory corresponding to `i` to 0. As a result, the `for` loop has reset, 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 you

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 you should consider the error messages provided by Python (or Java) as extraordinarily friendly: Not only does it tell you the cause of a problem, it even tells you exactly which line of the program was at fault. This saves a lot of debugging time.

Every once in a while, you'll see a C program crash, with a message like “segmentation fault” or “bus error.” It won't helpfully include any indication of what part of the program is at fault: all you 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. (It often instead indicates an attempt to reference an uninitialized pointer or a NULL pointer, which we'll discuss later.)”

2.6. 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 second 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);
  }
}
```

3. Libraries

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

3.1. Function prototypes

In C, a function must be declared above the location where you use it. In the C program of Figure 2, we defined the `gcd()` function first, then the `main()` function. This is significant: If we swapped the `gcd()` and `main()` functions, 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 raises 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, `math.c`, will contain the `gcd()` function, and the second file, `main.c`, will contain the `main()` function. The problem with this is that, in compiling `main.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 `main.c`.

```
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. You can see that it begins the same as a function definition begins, but we simply 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.

3.2. 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 `math.h`.

```
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`.

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

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

This particular example isn't very convincing, 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.

You'll notice that we've placed `stdio.h` in angle brackets, while `math.h` is in double quotation marks. The angle brackets are for standard header files — files more or less built into the C system. The quotation marks are for custom-written header files that can be found in the same directory as the source files.

3.3. 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've 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 that the programmer won'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, you could use `forever` as if it were a loop construct, and the preprocessor would replace it with "`while(1)`."

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

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

These are the basics of writing C programs, giving you enough to be able to write reasonably useful programs. But to be a proficient programmer in C, you'd need to know about pointers — a topic that we'll defer to another time.