C for Python Programmers

BBM 101 - Introduction to Programming I

Hacettepe University
Fall 2015

Fuat Akal, Aykut Erdem, Erkut Erdem, Vahid Garousi

Slides based on the material prepared by Carl Burch (Hendrix College) with modifications by Elizabeth Patitsas (U Toronto)

Creating computer programs

- Each programming language provides a set of primitive operations
- Each programming language provides mechanisms for combining primitives to form more complex, but legal, expressions
- Each programming language provides mechanisms for deducing meanings or values associated with computations or expressions

Slide credit: E. Grimson, J. Guttag and C. Terman

Recall our goal

- Learn the syntax and semantics of a programming language
- Learn how to use those elements to translate "recipes" for solving a problem into a form that the computer can use to do the work for us
- Computational modes of thought enable us to use a suite of methods to solve problems

Slide credit: E. Grimson, J. Guttag and C. Terman

Today

- Overview of Programming languages (PLs)
  - Dimensions of a PL
  - Programming paradigms
- How Python & C are similar
- How Python & C are different
  - C fundamentals
  - C Examples

Today

- Overview of Programming languages (PLs)
  - Dimensions of a PL
  - Programming paradigms
- How Python & C are similar
- How Python & C are different
  - C fundamentals
  - C Examples

Dimensions of a Programming Language

- **Low-level vs. High-level**
  - Distinction according to the level of abstraction
  - In low-level programming languages (e.g. Assembly), the set of instructions used in computations are very simple (nearly at machine level)
  - A high-level programming language (e.g. C, Java) has a much richer and more complex set of primitives.

Dimensions of a Programming Language

- **General vs. Targeted**
  - Distinction according to the range of applications
  - In a general programming language, the set of primitives support a broad range of applications.
  - A targeted programming language aims at a very specific set of applications.
    - **e.g.**, MATLAB (matrix laboratory) is a programming language specifically designed for numerical computing (matrix and vector operations)

Dimensions of a Programming Language

- **Interpreted vs. Compiled**
  - Distinction according to how the source code is executed
  - In interpreted languages (e.g. Python), the source code is executed directly at runtime (by the interpreter).
    - Interpreter control the the flow of the program by going through each one of the instructions.
  - In compiled languages (e.g. C), the source code first needs to be translated to an object code (by the compiler) before the execution.
  - More later today!

Programming Language Paradigms

- **Functional**
  - Treats computation as the evaluation of mathematical functions (e.g. Lisp, Scheme, Haskell, etc.)
- **Imperative**
  - describes computation in terms of statements that change a program state (e.g. FORTRAN, BASIC, Pascal, C, etc.)
- **Logical (declarative)**
  - expresses the logic of a computation without describing its control flow (e.g. Prolog)
- **Object oriented**
  - uses "objects" – data structures consisting of data fields and methods together with their interactions – to design applications and computer programs (e.g. C++, Java, C#, Python, etc.)

C (1973)

- Developed by Ken Thompson and Dennis Ritchie at AT&T Bell Labs for use on the UNIX operating system.
  - now used on practically every operating system
  - popular language for writing system software
- Features:
  - An extremely simple core language, with non-essential functionality provided by a standardized set of library routines.
  - Low-level access to computer memory via the use of pointers.
- C ancestors: C++, C#, Java

Slide credit: Thomas J. Cortina

The Strange Birth and Long Life of Unix

Photo: Alcatel-Lucent

- http://spectrum.ieee.org/computing/software/the-strange-birth-and-long-life-of-unix

Python

- Created by Guido van Rossum in the late 1980s
- Allows programming in multiple paradigms: object-oriented, structured, functional
- Uses dynamic typing and garbage collection

Slide credit: Thomas J. Cortina

Today

- Overview of Programming languages (PLs)
  - Dimensions of a PL
  - Programming paradigms
- How Python & C are similar
- How Python & C are different
  - C fundamentals
  - C Examples

Building a simple program in C (as compared to Python)

- Compilers versus interpreters
- Variable declarations
- Whitespace
- The `printf()` function
- Functions

Compilers versus interpreters

- One major difference between C and Python is how the programs written in these two languages are executed.
- 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*.

Compilers versus interpreters

- An **interpreter** reads the user-written program and performs it directly.
- 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.

The Programming Process in C

- After creating a C program, executing it is a two step process:

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

The Programming Process in C

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

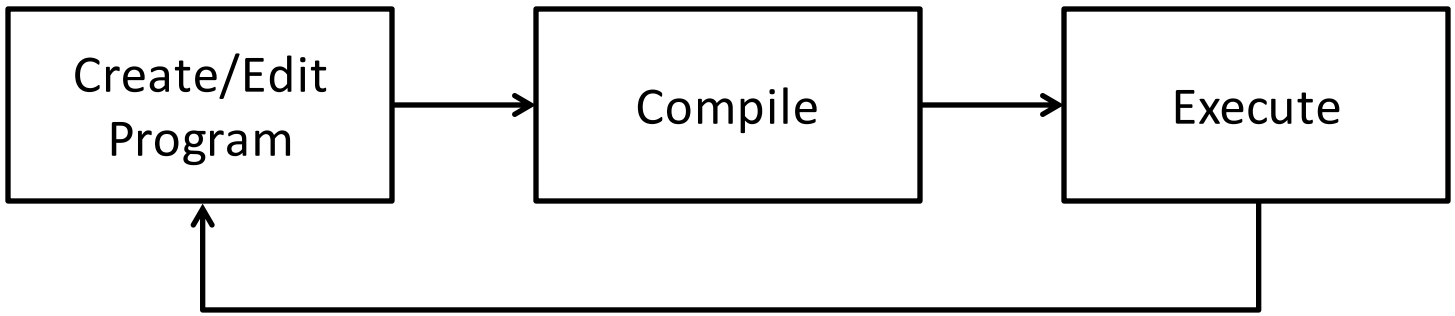
- invokes the compiler, named *gcc*.
- The compiler reads the source file `my_program.c` containing the C codes
- It generates a new file named `a.out` containing a translation of this code into the binary code used by the machine.

Compilers versus interpreters

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

- tells 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.

The Programming Process in C



"The cycle ends once the programmer is satisfied with the program, e.g., performance and correctness-wise."

Compilers versus interpreters

- An **interpreter** reads the user-written program and performs it directly.
- A **compiler** generates a file containing the translation of the program into the machine's native code.
- 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.

Variable declarations

- C requires **variable declarations**, informing the compiler about the variable before the variable is actually used.
- In C, the variable declaration defines the variable's **type.**
- **No such thing in Python!**

Declaring a Variable

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

```
double x;
```

- Value assignment is similar to Python:

```
x=3;
```

- x will actually hold the floating-point value 3.0 rather than the integer 3.
- However, once you declare a variable to be of a particular type, you cannot change its type!

Declaring a Variable

- 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 the program:
  - A variable is used but is not declared.
- 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.

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 using indentation.
  - like the body of a **while** or **if** statement
- 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 '}'.

Whitespace

C fragment

```c
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

```python
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
```

Whitespace

- As said, whitespace is insignificant in C.
- The computer would be just as happy if the previous code fragment is written as follows:

```
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;}}
```

- However, do not write your programs like this!

The `printf()` function

- In Python, displaying results for the user is accomplished by using **print**.
- In C, instead you use the `printf()` function which is provided by the C's standard library.
- The way the parameters to `printf()` work is a bit complicated but also quite convenient.

The `printf()` function

- The first parameter is a string specifying the format of what to print, and the following parameters indicate the values to print.
- Consider the following example:

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

- "`# solns: %d\n`" is the format string, `num_sol` is the value to be printed.
- The percent character is special to `printf()`.
  - It says to print a value specified in a subsequent parameter.
  - `%d` for integers/decimals
- If the value stored in `num_sol` is 2, the output is:

```
# solns: 2
```

The `printf()` function

- Like Python, C allows you to include escape characters in a string using a backslash:
  - The “\n” sequence represents the newline character,
  - The “\t” sequence represents the tab character,
  - “\"” sequence represents the double-quote character,
  - “\\” sequence represents the backslash character.
- These escape characters are part of C syntax, not part of the printf() function.

The `printf()` function

- Let's look at another example.

```
printf("# of solns: %d\n", 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 executes them sequentially.
- The output is:

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

The `printf()` function

- 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.
  - `%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.

Functions

- Unlike Python, all C code must be nested within functions, and functions cannot be nested within each other.
- 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.

Functions

- A C function is defined by naming the return type, followed by the function name, followed by a set of parentheses listing the parameters.
- Each parameter is described by including the type of the parameter and the parameter name.
- Here's a simple example of a function definition:

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

This is a function named `expon`, which takes two arguments, first a floating point number and next an integer, and returns a floating point number.

Functions

- 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.

Functions

C program

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

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

Python program

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

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

Statement-level constructs

- Operators
- Basic types
- Braces
- Statements
- Arrays
- Comments

Operators in C

Major operators in C and Python

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

- They look similar but there are some significant differences

Operators in C – 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.

Operators in C – Important Distinctions

- 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.
- Example:

```
while ((a = getchar()) != EOF)
```

  - The value returned by **`getchar()`** is assigned to the variable **`a`**,
  - The value assigned to **`a`** is tested whether it matches the **`EOF`** constant
  - It is used to decide whether to repeat the loop again.

Operators in C – Important Distinctions

- C defines assignment as an operator, whereas Python defines assignment as a statement.
- The value of the assignment oper… gned.
- A conseq… n le…
- Ex…
  - …, `getchar()` is assigned to the variable **a**,
  - The value assigned to **a** is tested whether it matches the **EOF** constant
  - It 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.

Operators in C – Important Distinctions

- 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.

Basic types in C

- C's list of basic 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

- Data Type Modifiers
  - **`signed`** / **`unsigned`**
  - **`short`** / **`long`**

`int`

- 4 bytes (on Unix)
- Base-2 representation.
- need one bit for + or -
- Range: -2^{31} to 2^{31}
- Variants: `short` (2 bytes), `long` (8 bytes), `unsigned` (only non-negative)

Slide credit: Bert Huang

`char`

- 1 byte
- ASCII representation in base-2
- Range: 0-255 (lots of unused)

Slide credit: Bert Huang

`float`

- Stands for “floating decimal point”
- 4 bytes
- Similar to scientific notation: $4.288 * 10^3$
- Very different interpretation of bits than `int` and `char`.
- Range: -10^{38} to 10^{38}

Slide credit: Bert Huang

No Boolean type for representing true/false

- 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*.
- Example

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

prints "**`in if`**" when executed since the value of **`(i)`** is 5 which is not 0

No Boolean type in C!

- 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);
```

No Boolean type in C!

- C's operators that look like they should compute B lean values (like `==`, `&&`, and ||) a lues instead
- I o re
- Th ng to

This quirk — that C regards all non-zero integers as true — is generally regarded as a mistake, and it 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. **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.)

Basic Data Types

Type	Size in Bytes	Range
`signed char`	1	-127 to +127
`unsigned char`	1	0 to 255
`short int`	2	-32,767 to +32,767
`unsigned short int`	2	0 to 65535
`int`	4	-32,767 to +32,767
`unsigned int`	4	0 to 65,535
`long int`	8	-2,147,483,647 to +2,147,483,647
`unsigned long int`	8	0 to 4,294,967,295
`float`	4	$\sim 10^{-37}$ to $\sim 10^{38}$
`double`	8	$\sim 10^{-307}$ to $\sim 10^{308}$
`long double`	16	$\sim 10^{-4931}$ to $\sim 10^{4932}$

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.
- Example:

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

Braces

- C programmers use this quite often when they want one of several **`if`** tests to be executed.
- Example:

```
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.

Braces

- C programmers use this quite often when they want one of several **`if`** tests to be executed.
- Example:

```
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.

Braces

- C programmers use this quite often when they want one of several **`if`** tests to be executed.
- Example:

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

Braces

- C programmers use this quite often when they want one of several `if` tests to be executed.
- Example:

```
disc = b*b
i
}
el
}
els
        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.

Statements

1. **Variable declarations**
   - No parallel in Python!
   - Example:

```
int x;
```

Statements

2. An expression as a statement

Two forms:

- An operator that changes a variable's value, like the assignment operator ("`x = 3;`"), the addition assignment operator **+=**, or the the increment operator **++**.
  - Example:

```
x = y + z;
```

- A function call, like a statement that simply calls the `printf()` function.
  - Example:

```
printf("%d", x);
```

Statements

3. An `if` statement

- 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`**".
- Example:

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

Statements

4. **A return statement**
   - 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**;".
   - Example:

```
return 0;
```

Statements

5. A `while` statement

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

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

Statements

6. A for statement

- While Python also has a **`for`** statement, its purpose and its syntax bear scant similarity to C's **`for`** statement
- Syntax:

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

  - The program will keep executing the *body* inside the **`for`** as long as the condition is true (non zero)
  - The *init* is tested before each iteration of the loop. The loop terminates when the condition is false.
  - The loop is controlled by a variable which is initialized and modified by the *init* and *update* (e.g. increment operation) expressions, respectively.

Statements

6. A for statement (cont'd.)

– Example 1:

```
for (i = 0; i < n; i++)
{
   body
}
```

for loops are mostly used for counting out *n* iterations

– Example 2:

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

Notice how the update portion of the **for** statement has changed to "**p *= 2**".

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**
  - Similar to Python's list except that an array in C cannot grow or shrink — its size is fixed at the time of creation.
- Example:

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

- Another way to make an array, if you know all the elements upfront, is:

```
char vowels[6] = {'a', 'e', 'i', 'o', 'u', 'y'};
```

Arrays

- 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)`**
- What happens if you access an array index outside the array, like accessing **`pops[50]`** or **`pops[-100]`**?
  - With Python, 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.

Arrays

- Example:

```
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 some Linux distributions) would place **`i`** in memory just after the **`vals`** array.
- When **`i`** reaches 5 and the computer executes "**`vals[i] = 0`**", it in fact resets the memory corresponding to **`i`** to 0.
  - 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.

Arrays

- Example:

```
int main() {
    int i
```

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. **Every once in a while, you'll see a C program crash, with a message like "segmentation fault".** 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.**

goes through the loop again, and

program never reaches the printf function call, and the program never terminates.

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.
- Example:

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

    ...
```

Comments

- 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.
- Example:

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

Libraries

- Separating a program into various files
  - Function prototypes
  - Header files
  - Constants

Function prototypes

- In C, a function must be declared above the location where you use it.
- The compiler would complain if a function is called before defining it.
- The reason is C assumes that a compiler reads a program from the top to bottom.
- One way to get around this, is to use **function prototyping**, writing the function header but omitting the body definition.

Function prototypes

- Consider the following example:

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

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

Line for the function prototype

- By using function prototypes, 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.

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**.

Header files

- A header file contains each prototype written just once (and possibly some additional shared information).
- The header files can then be referred to 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.

Header files

- Consider that the prototype `int gcd(int a,int b)` is put into a header file called `mathfun.h`.
- We can incorporate this header file at the top of main.c.

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

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

- The `#include` directive tells the preprocessor to replace this line with the contents of the file specified.
  - The angle brackets are for standard header files such as `stdio.h`.
  - The quotation marks are for custom-written header files that can be found in the same directory as the source files.

Constants

- `#define` directive tells the preprocessor to substitute all future occurrences of some word with something else.

- Example:

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

  - The preprocessors automatically translate the above expression into:

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