The C Programming Language
Chapter 1

(material from Dr. Michael Lewis, William & Mary Computer Science)

Overview

- Motivation
- Hello, world!
- Basic Data Types
- Variables
- Arithmetic Operators
- Relational Operators
- Assignments
- Boolean Operators

About C

- **texts**
  - The C Programming Language, 2nd ed. (1988) by Brian Kernighan and Dennis Ritchie (known as K&R; classic text)
  - C in a Nutshell, 2nd ed. (2016) by Peter Prinz and Tony Crawford (more current text; describes every function in the C standard library)

- **origins**
  - developed by Dennis Ritchie in early 1970s
  - BCPL > B > C
  - standardized in 1989
  - revisions: C11, C17, C2x
  - influential language
  - the language used to create the internet
  - heart of Unix, Linux, Windows operating systems (and Python interpreter!)

About C

- **language**
  - typed – all variables must be declared
  - compiled – must be transformed to machine language
  - fast
  - near proper subset of C++

- **low-level**
  - closer to the hardware – must know about bits and bytes
  - must know about memory
  - bugs can be difficult to fix
  - much more capability than other languages
  - C trusts that you know what you're doing – no guardrails or safety
    - C was not designed to stop you from doing stupid things, because that would also stop you from doing clever things.

Python vs. C vs. C++ vs. Java

	Python	C	C++	Java
printing	`print()`	`printf()`	`std::cout <<`	`System.out.println()`, `System.out.printf()`
string literals	`'boo!'` or `"boo!"`	`"boo!"`	same as C	same as C
line comments	`#`	`/* */`	`//`	`//`
block comments		`/* */`	same as C	same as C
addition, subtraction, multiplication	`+`, `-`, `*`	same	same	same
regular division	`/`	same[1]	same as in C	same as in C
integer division	`//`	`/`[2]	same as in C	same as in C
remainder	`%`	same	same	same
software integers:	`int`			
hardware integers:		`char`	same as C	
		`unsigned char`	same as C	
		`signed char`	same as C	`byte`
		`short` or `short int`	same as C	`short`
		`unsigned short` or `unsigned short int`	same as C	
		`int`	same as C	`int`
		`unsigned int`	same as C	
		`long` or `long int`	same as C	`long`
		`unsigned long` or `unsigned long int`	same as C	
binary32 floating point:		`float`	same as C	
binary64 floating point:	`float`	`double`	same as C	
booleans:	`True`, `False`	`true`, `false` or `1`, `0` or non-zero, zero	same as C	
single characters:		`char`, `unsigned char`	same as C	

[1] When one or both operands is non-integer.

[2] When both operands are integers.

Transitioning to C from Python

- lower level – more for you to program
- sometimes unsafe
- standard library is smaller
- different syntax
- structured vs. script
- paradigm shift: not object-oriented

- like going from automatic transmission to stick shift

Programming in C

- C is procedural, not object-oriented
- C is fully compiled (to machine code)
- C allows direct manipulation of memory via pointers
- C does not have garbage collection
- C has many important, yet subtle, details

Source: xkcd.com/571

Hello, world!

- C code in file named hello.c

```c
#include <stdio.h>

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

- to list in linux, we can use

```
cat -n ch01/hello.c  # cat is a command to concatenate and print files;
                     # the -n option gives us line numbers.
```

- Python equivalent

```
print('Hello, world!')
```

Hello world!

- **`#include <stdio.h>`**
  - tells the compiler to include this header file for compilation
  - **`stdio.h`** for I/O functions (e.g., **`printf`**)
- **`main()`**
  - main function, where execution begins
  - every C, C++, and Java program must have a **`main`** function
  - returns **`int`**
  - takes **`argc`** and **`argv`** command line parameters (Python: **`sys.argv`**)
- **`{}`**
  - curly braces are equivalent to stating "block begin" and "block end"
  - the code in between is called a "block“
  - Python uses indentation

Hello world!

- `printf()`
  - the actual print statement
  - Python: `print`
  - no newline default – use `"\n"`
  - strings in C/C++/Java delimited by double quotes: `"Hello, world!"`
- `return 0`
  - similar to Python
  - here, returns a value to execution environment

Header Files

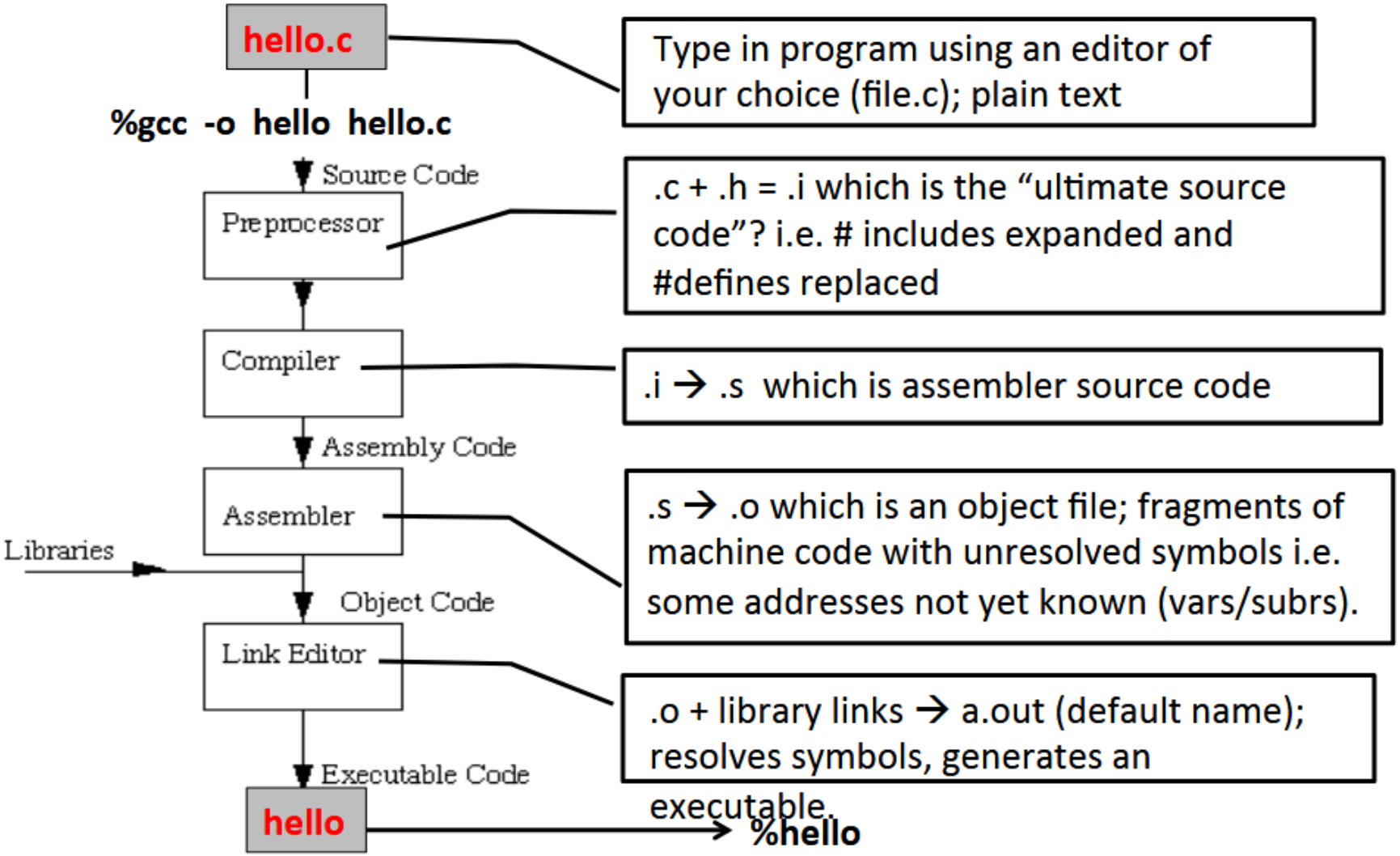
- **functions, types and macros of the standard library are declared in standard headers**
- **header file accessed by**
  - `#include <header>`
  - note: no semicolon
- **headers can be included in any order and any number of times**
  - must be included outside of any function
  - before any use of anything it declares
- **NEVER include C source files**
- **should use header guards**

```
#ifndef FILENAME_H
#define FILENAME_H
…
#endif
```

Coding Style

- **always explicitly declare the return type of the function**
  - defaults to a type integer
- **replace return 0 with return EXIT_SUCCESS (in <stdlib.h>)**
- **comments**
  - /* comment */
  - comments cannot be nested
  - // is a single line comment from // to the end of the line
- **blanks, tabs, and newlines (or "white space"), as well as comments, are ignored except to separate tokens**
  - free-form spacing

Compilation and Linking



Compilation and Linking

- use gcc (GNU C compiler) to compile and link
- flags begin with a – (dash or minus) or --
  - here we use options **`–Wall`** (all warnings) and **`–pedantic`** (cautions)
  - **`-o hello`** to create the executable (if not included, creates **`a.out`**)

```
gcc -Wall -pedantic -o hello ch01/hello.c
```

- confirm file creation

```
date          # Print date and time.
ls -ls hello  # List information about the file hello, including last modified date.
file hello    # Check what type of file hello is.
```

```
Thu Feb 15 13:06:52 EST 2024
72 -rwxr-xr-x  1 rml  staff  33432 Feb 15 13:06 hello
hello: Mach-O 64-bit executable arm64
```

Statements and Comments

- statements terminated by a semicolon ;
- statements can be split across multiple lines
- here's what happens when you omit a semicolon

```
cat -n ch01/no_semicolon.c
```

```
     1  #include <stdio.h>
     2
     3  int main(void)
     4  {
     5    printf("Hello, world!\n")
     6    return 0
     7  }
```

```
gcc -Wall -pedantic -c ch01/no_semicolon.c
```

```
ch01/no_semicolon.c:5:28: error: expected ';' after expression
  printf("Hello, world!\n")
                           ^
                           ;
ch01/no_semicolon.c:6:11: error: expected ';' after return statement
  return 0
          ^
          ;
2 errors generated.
```

Statements and Comments

- **single line comments: `//`**
  - similar to Python **#**
- **multiline comments: `/*  */`**
  - popular style

```
/*
 * I prefer formatting multiline comments this
 * way so that the body of the comment is clear.
 */
```

Statements and Comments

- comments cannot be nested

```
cat -n ch01/nested_comments.c
```

```
     1  #include <stdio.h>
     2
     3  int main(int argc, char **argv)
     4  {
     5    /* You cannot /* nest block comments */. */
     6    // Some compilers accept // to indicate the start of a comment line.
     7    return 0;
     8  }
```

```
gcc ch01/nested_comments.c
```

```
ch01/nested_comments.c:5:17: warning: '/*' within block comment [-Wcomment]
  /* You cannot /* nest block comments */. */
                ^
ch01/nested_comments.c:5:42: error: expected expression
  /* You cannot /* nest block comments */. */
                                           ^
1 warning and 1 error generated.
```

Variable Names

- **rules for naming variables in C/C++/Java same as Python**
  - letters, numbers, and underscores (case sensitive)
  - start with letter or underscore only
  - spaces not allowed – use underscores instead

```
float your_boat;
long john_silver;
short bread_cookie;
```

- **variable names should be mnemonic, but not ridiculously long**
  - first 31 characters significant for function names and global variables
  - first 63 characters for other variables
- **avoid _ as beginning character, as could collide with standard C library names**
- **avoid CamelCase, as it is an abomination**

C Keywords

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

- **keywords are reserved words, and may not be used as identifiers**
- **reserves a word or identifier to have a particular meaning**
  - meaning of keywords — and, indeed, the meaning of the notion of keyword differs widely from language to language.
  - do not use them for any other purpose in a C program
  - allowed, of course, within double quotation marks

Type Declarations

- **in C/C++/Java, you must declare the type of a variable before using it**
  - unlike Python, which infers the type
- **C is not happy if we try to change the type of a variable, or declare its type twice (even if the type is the same)**

Type Declarations

```
cat -n ch01/buggy_types.c
```

```
     1  #include <stdio.h>
     2
     3  int main(int argc, char **argv)
     4  {
     5    n = 42;  /* n is used without declaration. */
     6    int m;
     7    int m;   /* Redefinition of m. */
     8    printf("Th-th-th-that's all, folks!\n");
     9    return 0;
    10  }
```

```
gcc ch01/buggy_types.c
```

```
ch01/buggy_types.c:5:3: error: use of undeclared identifier 'n'
  n = 42;  /* n is used without declaration. */
  ^
ch01/buggy_types.c:7:7: error: redefinition of 'm'
  int m;   /* Redefinition of m. */
      ^
ch01/buggy_types.c:6:7: note: previous definition is here
  int m;
      ^
2 errors generated.
```

Type Declarations

- corrected version

```
cat -n ch01/types.c
```

```
     1  #include <stdio.h>
     2
     3  int main(int argc, char **argv)
     4  {
     5    int n = 42;
     6    int m;
     7    printf("Th-th-th-that's all, folks!\n");
     8    return 0;
     9  }
```

```
gcc -Wall ch01/types.c  # The -Wall option tells gcc to show us all warnings.
```

```
ch01/types.c:5:7: warning: unused variable 'n' [-Wunused-variable]
  int n = 42;
      ^
ch01/types.c:6:7: warning: unused variable 'm' [-Wunused-variable]
  int m;
      ^
2 warnings generated.
```

Type Declarations

- warnings are not errors, but may indicate a bug
- executable will be generated and program will run

```
./a.out
```

```
Th-th-th-that's all, folks!
```

Type Declarations

- printf is similar to Python print, but insists on formal strings
- e.g., %f interprets the bits of the variable as a float

```
cat -n ch01/printf.c
```

```
     1  #include <stdio.h>
     2
     3  int main(int argc, char **argv)
     4  {
     5    float x = 54.0;
     6    printf("%f\n", x);
     7    return 0;
     8  }
```

```
gcc -Wall ch01/printf.c
```

```
./a.out
```

```
54.000000
```

Type Declarations

- C has many format codes
- Python originally used printf() style strings and continues to support them

```
cat -n ch01/printf1.c
```

```
     1  #include <stdio.h>
     2
     3  int main(int argc, char **argv)
     4  {
     5    float x = 54.0;
     6    printf("%d\n", x);
     7    return 0;
     8  }
```

```
gcc -Wall ch01/printf1.c
```

```
ch01/printf1.c:6:18: warning: format specifies type 'int' but the argument has type 'float' [-Wformat]
  printf("%d\n", x);
          ~~     ^
          %f
1 warning generated.
```

Type Declarations

- note in previous example, only a warning was generated
- C trusts you know what you're doing and that you have a good reason to interpret the bits of a float as an int

```
./a.out
```

```
0
```

Primitive Data Types

- **integer**
  - char – smallest addressable unit; each byte has its own address
  - short – short int; not used as much
  - int – default type for an integer constant value
  - long – do you really need it?
- **floating point**
  - inexact
  - float – single precision (about 6 decimal digits of precision)
  - double – double precision (about 15 decimal digits of precision)
    - default for literal unless suffixed with ‘f’
- **no Boolean or string types**
- **no high-level types, such as lists, dictionaries, etc., but can be built**

Type Declarations

- sizeof() determines byte size of variable or type

```
cat -n ch01/int_widths.c
```

```
     1  #include <stdio.h>
     2
     3  int main(int argc, char **argv)
     4  {
     5      int n;
     6
     7      /* sizeof() applied to types. */
     8      printf("Sizes (in bytes) of different flavors of integers:\n");
     9      printf("A char is           %lu byte long.\n", sizeof(char));
    10      printf("A signed char is    %lu byte long.\n", sizeof(signed char));
    11      printf("An unsigned char is %lu byte long.\n\n", sizeof(unsigned char));
    12
    13      printf("A short int is           %lu bytes long.\n", sizeof(short int));
    14      printf("An unsigned short int is %lu bytes long.\n\n", sizeof(unsigned short int));
    15
    16      printf("An int is          %lu bytes long.\n", sizeof(int));
    17      printf("An unsigned int is %lu bytes long.\n\n", sizeof(unsigned int));
    18
    19      printf("A long int is           %lu bytes long.\n", sizeof(long int));
    20      printf("An unsigned long int is %lu bytes long.\n\n", sizeof(unsigned long int));
    21
    22      printf("A long long int is           %lu bytes long.\n", sizeof(long long int));
    23      printf("An unsigned long long int is %lu bytes long.\n\n", sizeof(unsigned long long int));
    24
    25      printf("Sizes (in bytes) of different flavors of floating-point numbers:\n");
    26      printf("A float is       %lu bytes long.\n", sizeof(float));
    27      printf("A double is      %lu bytes long.\n", sizeof(double));
    28      printf("A long double is %lu bytes long.\n\n", sizeof(long double));
    29
    30      printf("A pointer is %lu bytes long.\n", sizeof(int*));
    31      return 0;
    32  }
```

Type Declarations

- `sizeof()` is not like `len()`, but `sys.getsizeof()`

```
gcc ch01/int_widths.c
```

```
./a.out
```

```
Sizes (in bytes) of different flavors of integers:
A char is           1 byte long.
A signed char is    1 byte long.
An unsigned char is 1 byte long.

A short int is           2 bytes long.
An unsigned short int is 2 bytes long.

An int is          4 bytes long.
An unsigned int is 4 bytes long.

A long int is           8 bytes long.
An unsigned long int is 8 bytes long.

A long long int is           8 bytes long.
An unsigned long long int is 8 bytes long.

Sizes (in bytes) of different flavors of floating-point numbers:
A float is       4 bytes long.
A double is      8 bytes long.
A long double is 8 bytes long.

A pointer is 8 bytes long.
```

Basic Data Types

```
char           1 bytes -128 to 127
unsigned char  1 bytes 0 to 255
short          2 bytes -32768 to 32767
unsigned short 2 bytes 0 to 65535
int            4 bytes -2147483648 to 2147483647
unsigned int   4 bytes 0 to 4294967295
long           4 bytes -2147483648 to 2147483647
unsigned long  4 bytes 0 to 4294967295
float          4 bytes 1.175494e-38 to 3.402823e+38
double         8 bytes 2.225074e-308 to 1.797693e+308
```

Type	bytes	bits
char	1	8
short	2	16
int	4	32
long	8	64
long long	8	64

- **char guaranteed to be one byte**
- **no maximum size for a type, but the following relationships must hold:**
  - sizeof (short) <= sizeof (int) <= sizeof (long)
  - sizeof (float) <= sizeof (double) <= sizeof (long double)

Type Declarations

- **C/C++ have a variety of integers which differ in number of bits**
- **integers in Python implemented in software and are unbounded**
- **C/C++ integers depend on hardware and may behave differently across machines, but hardware becoming more standardized**
- **integers can be signed or unsigned**
  - signed: negative or non-negative; 1 bit is sign bit
  - unsigned: always non-negative (>=0); no sign bit, extra bit doubles range
- **integers used in different ways**
  - color channels in pixels: 8-bits to fill one unsigned char; three channels, so total number of colors 2^24

Type Declarations - Integers

- **signed integer: int**
- **unsigned integer: unsigned int**
- **literal for unsigned int: u or U**
- **signed and unsigned int same size, but bit pattern is determined differently**
  - signed int: %d
  - unsigned int: %u
- **short int (2 bytes)**
- **long and long long types (l/L and ll/LL) (8 bytes and 8 bytes)**

Type Declarations - Integers

- example

```
cat -n ch01/int_format.c
```

```
     1  #include <stdio.h>
     2
     3  int main(int argc, char **argv)
     4  {
     5    int n = -1;
     6
     7    /* %d is the format code for signed int */
     8    printf("The bits in n interpreted as a signed int: %d\n", n);
     9
    10    /* %u is the format code for unsigned int */
    11    printf("The bits in n interpreted as an unsigned int: %u\n", n);
    12
    13    return 0;
    14  }
```

```
gcc -Wall ch01/int_format.c
```

```
./a.out
```

```
The bits in n interpreted as a signed int: -1
The bits in n interpreted as an unsigned int: 4294967295
```

Integer Ranges

C Language Variable Types

Whether you're working with regular or unsigned variables in your C program, you need to know a bit about those various variables. The following table show C variable types, their value ranges, and a few helpful comments:

Type	*Value Range*	*Comments*
char	–128 to 127	
unsigned char	0 to 255	
int	–32,768 to 32,767	16-bit
	–2,147,483,648 to 2,147,483,647	32-bit
unsigned int	0 to 65,535	16-bit
	0 to 4,294,967,295	32-bit
short int	–32,768 to 32,767	
unsigned short int	0 to 65,535	
long int	–2,147,483,648 to 2,147,483,647	
unsigned long int	0 to 4,294,967,295	
float	1.17×10^{-38} to 3.40×10^{38}	6-digit precision
double	2.22×10^{-308} to 1.79×10^{308}	15-digit precision

Type Declarations - Integers

- integers can overflow

```
cat -n ch01/int_overflow.c
```

```
     1  #include <stdio.h>
     2
     3  int main(int argc, char **argv)
     4  {
     5    int n;
     6
     7    /* A quiet integer overflow. */
     8    n = +2147483647;  /* 2**31 - 1 is the maximum for 32 bit signed integers **/
     9    printf("n     = %+d\n", n);
    10
    11    n = n + 1;
    12    printf("n + 1 = %+d\n\n", n);
    13
    14    /* A noisier integer overflow. */
    15    n = 4294967295;  /* 2**32 - 1 */
    16    printf("As a signed int, 4294967295 = %d !\n", n);
    17
    18    return 0;
    19  }
```

```
gcc ch01/int_overflow.c
```

```
ch01/int_overflow.c:15:7: warning: implicit conversion from 'long' to 'int' changes value from 4294967295 to -1 [-Wconstant-con
version]
  n = 4294967295;  /* 2**32 - 1 */
    ~ ^~~~~~~~~~
1 warning generated.
```

Type Declarations - Integers

- integers can overflow

```
./a.out
```

```
n     = +2147483647
n + 1 = -2147483648

As a signed int, 4294967295 = -1 !
```

- overflow errors can be
  - amusing: https://www.cbc.ca/news/entertainment/psy-s-gangnam-style-breaks-the-limit-of-youtube-s-video-counter-1.2860186
  - disruptive:https://www.bleepingcomputer.com/news/microsoft/microsoft-exchange-year-2022-bug-in-fip-fs-breaks-email-delivery/
  - dangerous: https://www.cs.wm.edu/~rml/teaching/c/docs/787_overflow.pdf

Type Declarations - Floats

- **no industry standard for floating point numbers before 1985**
  - each hardware vendor had its own system
- **1985 – IEEE 754 floating point standard**
- **C/C++ have three types of floats**
  - `float` : 32 bits
  - `double` : 64 bits
  - `long` : 80 bits
- **two special float values**
  - `inf`
    - infinity, larger than all other numbers (except itself) (division by 0)
  - `nan`
    - not a number (divide 0.0 by itself)
    - not equal to any number, even itself!

Type Declarations - Floats

```
cat -n ch01/inf_nan.c
```

```
     1  #include <stdio.h>
     2
     3  int main(void)
     4  {
     5    double inf = 1.0/0.0;
     6    double nan = 0.0/0.0;
     7
     8    printf("Here is an Inf: %lf\n", inf);
     9    printf("Here is a  NaN: %lf\n\n", nan);
    10
    11    printf("1/inf = %lf\n", 1.0/inf);
    12    printf("inf - inf = %lf\n", inf - inf);
    13
    14    if (nan != nan) {
    15      printf("Yow!  nan != nan!\n");
    16    }
    17
    18    return 0;
    19  }
```

```
gcc ch01/inf_nan.c
```

```
./a.out
```

```
Here is an Inf: inf
Here is a  NaN: nan

1/inf = 0.000000
inf - inf = nan
Yow!  nan != nan!
```

Type Declarations - Characters

- **unlike Python, C distinguishes between characters and strings**
- **delimited by single quotes**
- **`char`'s are really `int`'s**
  - index in the ASCII table
  - similar to **`ord`** and **`chr`** in Python
  - we can use **`char`**'s in mathematical expressions:

```
c - '0' = 50 - 48 = 2    if c == '2'
```

Type Declarations - Characters

```
cat -n ch01/char_int.c
```

```
     1  #include <stdio.h>
     2
     3  int main(void)
     4  {
     5      printf("'0' as an int: %d\n", '0');  /* print '0' as an int */
     6      printf("'2' as an int: %d\n", '2');  /* print '2' as an int */
     7      printf("'2' - '0': %d\n", '2' - '0');
     8      printf("48 as a char: %c\n", 48);  /* print 48 as a character */
     9      printf("54 as a char: %c\n", 54);  /* print 54 as a character */
    10
    11      return 0;
    12  }
```

```
gcc ch01/char_int.c
```

```
./a.out
```

```
'0' as an int: 48
'2' as an int: 50
'2' - '0': 2
48 as a char: 0
54 as a char: 6
```

Type Declarations - Arrays

- an array is like Python's `array` or `list`
- contiguous chunk of memory to hold a number of variables of the same type
- use an index to specify a single element in an array
- sample declarations

```
int n[42];    /* An array that can hold 42 ints. */
float x[54];  /* An array that can hold 54 floats. */
```

- sample code with arrays

```
cat -n ch01/variable_size.c
```

```
     1  int main(void)
     2  {
     3    int n = 42;
     4    int a[n];  /* The same as int a[42], an array of 42 int. */
     5
     6    a[0] = 54;
     7
     8    return 0;
     9  }
```

Type Declarations - Arrays

- **often need to keep length of array in a variable**
  - no `len()` function, as in Python
- **differences between C arrays and Python lists**
  - arrays span contiguous regions of memory, while lists can be scattered across memory
  - contiguity allows C to work creatively with data
  - all items in arrays are the same type, unlike Python lists
    - must know width of each element to find it in the array
  - no checking for out of bounds indices for speed

Type Declarations - Strings

- C has no string type
- string: array of characters terminated by `'\0'`

```
cat -n ch01/string.c
```

```
     1  #include <stdio.h>
     2
     3  int main(void)
     4  {
     5    char c = 'a';    // Character literal.
     6    char s[] = "a";  // String as an array of characters.
     7
     8    printf("sizeof(c): %lu bytes\n", sizeof(c));
     9    printf("sizeof(s): %lu bytes\n", sizeof(s));
    10
    11    return 0;
    12  }
```

```
gcc ch01/string.c
```

```
./a.out
```

```
sizeof(c): 1 bytes
sizeof(s): 2 bytes
```

Type Declarations - Strings

- use `strlen()` to find the length of a string
- if string has no NULL character at end, C will keep reading past the end of the string until it finds a one

```
cat -n ch01/strlen.c
```

```
     1  #include <stdio.h>
     2  #include <strings.h>
     3
     4  int main(void)
     5  {
     6    char s[] = "a";  // String as an array of characters.
     7
     8    printf("sizeof(s): %lu bytes\n", sizeof(s));
     9    printf("strlen(s): %lu bytes\n", strlen(s));
    10
    11    return 0;
    12  }
```

```
gcc ch01/strlen.c
```

```
./a.out
```

```
sizeof(s): 2 bytes
strlen(s): 1 bytes
```

Type Conversions

- types can converted explicitly or implicitly
- cast to convert the type
- implicit conversion when using variables of different types in expressions

```
int n;
double x;
x = (double) n;
```

More on Assignment

- **in C, assignments are expressions, not statements**
  - allows multiple assignment a = b = c = 1;
- **assignment operators**
  - same precedence: right to left
  - = assignment
  - == comparison
  - perform the indicated operation between the left and right operands, then assign the result to the left operand
    - += add to
    - -= subtract from
    - *= multiply by
    - /= divide by
    - %= modulo by

C Operators

C Language Operators

In programming with C, you occasionally want to use common mathematical operators for common mathematical functions and not-so-common operators for logic and sequence functions. Here's a look at C language operators to use:

Operator, Category, Duty	Operator, Category, Duty	Operator, Category, Duty
=, Assignment, Equals	!=, Comparison, Is not equal to	>, Bitwise, Shift bits right
+, Mathematical, Addition	&&, Logical, AND	~, Bitwise, One's complement
–, Mathematical, Subtraction	||, Logical, OR	+, Unary, Positive
*, Mathematical, Multiplication	!, Logical, NOT	–, Unary, Negative
/, Mathematical, Division	++, Mathematical, Increment by 1	*, Unary, Pointer
%, Mathematical, Modulo	--, Mathematical, Decrement by 1	&, Unary, Address
>, Comparison, Greater than	&, Bitwise, AND	sizeof, Unary, Returns the size of an object
>=, Comparison, Greater than or equal to	|, Bitwise, Inclusive OR	., Structure, Element access
<, Comparison, Less than	^, Bitwise, Exclusive OR (XOR or EOR)	->, Structure, Pointer element access
<=, Comparison, Less than or equal to	<<, Bitwise, Shift bits left	?:, Conditional , Funky if operator expression
==, Comparison, Is equal to		

Boolean Operators

- **C does not have a distinct boolean type**
  - int is used instead
- **treats integer 0 as FALSE and all non-zero values as TRUE**

```
i = 0;
while (i - 10) { ... }
```

  - will execute until the variable i takes on the value 10 at which time the expression (i - 10) will become false (i.e., 0)
- **a sampling of Logical/Boolean Operators:**
  - &&, ||, and ! → AND, OR, and NOT in Python
- **&& is used as logical and**
  - x != 0 && y != 0
- **short-circuit evaluation: above example, if x != 0 evaluates to false, the whole statement is false regardless of the outcome of y != 0 (same for or if first condition is true)**

Increment Operators

- **prefix/postfix**

Example 1	Example 2

`x = 3;`
`y = ++x;`
`// x contains 4, y contains 4`
	`x = 3;`
	`y = x++;`
	`// x contains 4, y contains 3`

Printing Decimal and Floating Point

- **integers: %nd**
  - n = width of the whole number portion for decimal integers
- **float: %m.nf**
  - m = total character width, including decimal point
  - n = precision width after decimal

%d	print as decimal integer
%6d	print as decimal integer, at least 6 characters wide
%f	print as floating point
%6f	print as floating point, at least 6 characters wide
%.2f	print as floating point, 2 characters after decimal point
%6.2f	print as floating point, at least 6 wide and 2 after decimal point

Escape Sequences

Escape code	Description
\n	newline
\r	carriage return
\t	tab
\v	vertical tab
\b	backspace
\f	form feed (page feed)
\a	alert (beep)
\'	single quote (')
\"	double quote (")
\?	question mark (?)
\\	backslash (\)

Formatted I/O

- **printf and scanf**
  - both formatted I/O
  - both use standard I/O location

- **printf**
  - converts values to character form according to format string
  - outputs to stdout

- **scanf**
  - converts characters according to the format string, followed by pointer arguments indicating where the resulting values are stored
  - inputs from stdin

scanf

- **requires two parameters**
  - format string argument with specifiers
  - set of variable pointers to store corresponding values
- **format string**
  - skips over all leading white space (spaces, tabs, newlines)
  - % and type indicator
    - in between: maximum field-width, type indicator modifier, or * (input suppression)
  - input stops at end of format string, type mismatch in reading
    - next call to scanf resume searching for input of correct type where previous scanf left off
- **return value**
  - # of values converted

Basic I/O

```
#include <stdio.h>

int main ()
{
    int x;
    scanf ("%d", &x);    /* why need & ? */
    printf ("%d\n", x);

    float var;
    scanf ("%f", &var);    printf ("%f\n", var);
    scanf ("%d", &var);    printf ("%d\n", var);
    scanf ("%lf", &var);   printf ("%lf\n", var);

    int first, second;
    scanf ("%d %d", &first, &second);

    int i, j;
    scanf (" %d %*d %*d %*d %d ", &i, &j);
}
```