C Programming, Chapter 1: C vs. Java

T. Karvi

March 2011

C and Java I

Although the syntax of Java and C are very similar, they are very different languages. The following table shows some of the major differences:

C	**Java**
Procedural	Object-oriented
Compiled	Interpreted
No Memory Management	Memory Management
Pointers	References
Error Codes	Exceptions

Object-Oriented vs. Procedural:

- One of the largest differences between Java and C is the use of a different programming paradigm. Java is an Object- Oriented language. A Java program consists of a collection of objects. These objects contain the data used in the program, and have methods to perform operations on this data.

C and Java II

- The C language is procedural. A C program consists of a collection of procedures (or functions). The data used by the program can be put into local variables (inside of a function) or global variables (outside of functions) There is no notion of objects in C. Just like in Java, there is a special main function,Interpreted vs. Compiled Java is an interpreted language. Java source code is transformed to bytecode, which is then loaded by a program called an interpreter. This program then 'executes' each of the bytecode instructions one by one, translating them into something the machine understands. C programs are compiled. Instead being translated to some intermediate format (like bytecode) it is translated directly into machinecode. This machinecode is directly executed by the processor. which is used to start the program.

Interpreted vs. Compiled:

C and Java III

- Java is an interpreted language. Java source code is transformed to bytecode, which is then loaded by a program called an interpreter. This program then 'executes' each of the bytecode instructions one by one, translating them into something the machine understands.
- C programs are compiled. Instead being translated to some intermediate format (like bytecode) it is translated directly into machinecode. This machinecode is directly executed by the processor.

Memory Management vs. NoMemoryManagement:

- In Java, the memory management is done automatically by the system. New objects can be created using the new keyword. When objects are no longer used (i.e., no longer have any references pointing to them) they are removed by the garbage collector.
- In C, the programmer has to do his own memory management. Using the keyword sizeof and the library calls malloc and free, blocks of memory can be allocated and freed.

C and Java IV

References vs. Pointers:

- A reference in Java is a special variable which references (points-to) an object. Only objects can be referenced. For example, it is not possible to have a reference to an int.
- Pointers in C are in some ways similar to references in Java (they point to things), but in many ways they are very different.

Exceptions vs. Error Codes:

- Whenever an error occurs in Java, an exception is thrown.
- C has no exceptions. A function either returns some error code (when an error is expected), or your program crashes (usually long after the error has occurred).

Example I

```
#include <stdio.h>
double value;
/* This is a comment */
int main(void)
{
int local = 0;
value = 0.42;
printf("local = %d value = %f\n", local, value);
return 0;
}
```

- The program starts with the line `#include <stdio.h>`, which is actually not C-code, but a preprocessor directive. The preprocessor is a special program which pre-processes the C program before it is compiled. All statements beginning with # are preprocessor directives.

Example II

- The purpose of the `#include <stdio.h>` statement is similar to an 'import' in Java. It imports a header file called stdio.h into this program. Header files contain descriptions (or prototypes) of functions and types which are implemented and defined somewhere else. They usually have the extension '.h'. By importing the header file stdio.h, the functions and type described in that file can be used in this file (in this case we are interested in using the 'printf' function).
- The next line 'double value;' defines a global variable. Global variables are variables which are defined outside the scope of a function. They exist throughout the lifetime of the program (they are created when the program starts and destroyed when the program exits). Because they are global, they can be 'seen' and used in every function in this file.

Example III

- After the comment '/* This is a comment */' (which is similar to a Java comment) the special startup function 'int main(void)' is declared, which returns an 'int' result, and takes no parameters. In C programs, it is customary to return an 'int' result from the 'main' function. A result of 0 indicates that no error has occurred. Any other value indicates an error. When a function in C takes no parameters, this is indicated by the (void) parameter list.
- The 'int local = 0;' line declares a local variable in the main function The rules in C for declaring local variables are a little different from Java. All local variables must be declared at the beginning of the function. Declaring them at a later point will result in a compile time error.
- After declaring the local variable, a value is assigned to the global variable in the statement 'value = 0.42;'. The values of the variables are then printed to the screen using the 'printf("local = statement.

Example IV

- Finally, a value of 0 is returned (return 0;) to indicate that the program has finished without any errors.
- We can now compile the C program using the following command:
  gcc myprogram.c
- In this example, the C compiler is called 'gcc' (this stands for GNU C Compiler. The result is an executable file, called 'a.out' which we can run.
  ./a.out
- Our program then produces the following output:
  local = 0 value = 0.420000

Integer Types I

- There are three basic integer types, all with two version: *short int, unsigned short int, int, unsigned int, long int, unsigned long int*.
- One way to determine the ranges of the integer types for a particular implementation is to check the *limits.h* header, which is part of the standard library. For example, the usual range for an int variable x is $-2147483648 \leq x \leq 2147483647$ on 32-bit and 64-bit machines.
- You can get the size of a type in bytes using the function *sizeof*.
- C99 provides two additional standard integer types, *long long int* and *unsigned long long int*.
- C allows integer constants to be written in decimal, octal, or hexadecimal:

  Decimal 15 255 32767
  A decimal constant cannot start with zero 0.

Integer Types II

Octal An octal constant must start with zero and a constant contains only digits 0,...,7.
017 0377 077777

Hexadecimal starts with 0X or 0x and a constant contains digits 0,...,9 and letters a,...,f (upper or lower case):
0xff 0xfF 0xFf OXFF

- To force the compiler to treat a constant as a long integer, just follow it with the letter L or l: 15L, 0377L, 0x7fffL.
- To indicate that a constant is unsigned, put the letter U or u after it: 15U, 0377U, 0x7fffU.
- L and U may be used in combination to show that a constant is both long and unsigned: 0xffffffffUL.
- In C99, integer constants that end with either LL or ll have type long long int.

Integer Types III

- When integer overflow occurs during an operation on signed integers, the program's behaviour is undefined. When overflow occurs during an operation on unsigned integers, the result is defined: we get the correct answer modulo 2^n, where n is the number of bits used to store the result.
- Reading and writing integers:

```
int i;

scanf("%d", &i);
printf("%d",i);
```

- Reading and writing unsigned integers:

Integer Types IV

```
unsigned int u;

scanf("%u", &u);
printf("%u",u);

scanf("%o", &u); /* reads u in base 8 */
printf(%o",u);   /* writes u in base 8 */

scanf("%x", &u); /* reads u in base 16 */
printf(%x",u);   /* writes u in base 16 */
```

- Reading and writing short integers, put the letter h in front of d, o, u, or x:

Integer Types V

```
short s;

scanf("%hd", &s);
printf("%hd",s);
```

- Reading and writing long integers, put the letter l in front of d, o, u, or x:

```
long l;

scanf("%ld", &l);
printf("%ld",l);
```

- Reading and writing long long integers, put the letters ll in front of d, o, u, or x.

Floating Types I

- C provides three floating types:
  - *float*,
  - *double*,
  - *long double*.
- Use float when the precision is not very important, double in normal numerical calculations. Long double is rarely used.
- Floating constants can be written in various ways. For example the number 57.0 can be written

  `57.0    57.    57.0e0    57E0    5.7e1    5.7e+1    .57e2`

  By default, floating constants are stored as double-precision numbers.
- Reading a double constant:

Floating Types II

```
double d;

scanf("%lf", &d);
```

- Use l only in a scanf format string, not a printf string. In print, e, f, and g can be used to write float and double values. (C99 legalises the use of le, lf, lg, although l has no effect.)
- Reading and writing long double:

```
long double ld;

scanf("%Lf", &ld);
printf("%Lf, ld);
```

Character Types I

- The character type is *char*.
- C treats characters as small integers. Thus characters can be added and increased. They can be compared just as numbers can:

  ```
  if ('a' <= ch && ch <= 'z')
      ch = ch - 'a' +'A';
  ```

  We can easily write a for statement whose control variable steps through all the upper-case letters:

  ```
  for (ch = 'A'; ch <= 'Z'; ch++) ...
  ```

- Characters may be either signed or unsigned! The C standard does not specify whether char is signed or unsigned, but it allows the definitions:

  ```
  signed char sch;
  unsigned char uch;
  ```

Character Types II

Signed character typically have values between -128 and 127, while unsigned characters have values between 0 and 255.

- To convert a lower-case letter to upper-case: `ch = toupper(ch);`
  To convert a upper-case letter to lower-case: `ch = tolower(ch);`
- Reading and writing characters using *scanf* and *printf*:

  ```
  char ch;
  scanf("%ch", &ch);
  printf("%c", ch);
  ```

  Scanf does not skip white characters before reading a character. To force *scanf* to do so, put a space in its format string just before %c:

  ```
  scanf(" %c", &ch);
  ```

- Since scanf does not skip white space, it is easy to to detect the end of an input line:

```
do {
   scanf("%c", &ch);
} while (ch != '\n');
```

When scanf is called the next time, it will read the first character on the next input line.

- Writing a single character using *putchar*:

```
putchar(ch);
```

- Reading a single character using *getchar*:

```
ch = getchar();
```

- *getchar* and *putchar* are faster than *scanf* and *printf*, because the latter are much more complicated.
- *getchar* and *putchar* make it popssible to write short code. Compare:

```
do {
   scanf("%c", &ch);
} while (ch != '\n');
```

```
while ((ch = getchar();) != '\n');
```

```
while ((getchar();) != '\n');
```

All these skip the rest of the input line.

- To skip an indefinte number of blank characters:

```
while ((ch = getchar()) == ' ');
```

When the loop terminates, ch will contain the first nonblank character that getchar encountered.

- Be careful if you ix scanf and getchar in the same program. For example

Character Types V

```
printf("Enter an integer: ");
scanf("%d", &i);
printf("Enter a command: ");
command = getchar();
```

Scanf will leave behind any character that were not consumed during the reading of i, including the new-line character. getchar will fetch the first leftover character, which wasn't what we expected.

Example Program: The length of a message I

```
#include <stdio.h>
int main(void)
{
   char ch;
   int len = 0;

   printf("Enter a message: ");
   ch = getchar();
   while (ch != '\n') {
      len++;
      ch = getchar();
   }
   printf("Your message was %d charcaters long. \n", len);
   return 0;
}
```

Example Program: The length of a message II

Or shorter:

```
#include <stdio.h>
int main(void)
{
   char ch;
   int len = 0;

   printf("Enter a message: ");
   while (getchar != '\n') {
      len++;
   }
   printf("Your message was %d charcaters long. \n", len);
   return 0;
}
```

Arithmetical Type Conversion I

- C allows the basic types to be mixed in expressions. The compiler handles type conversions automatically, without the programmer's involvement. We speak of *implicit conversions*.
- The rules for performing implicit conversions are somewhat complex. Conversions are performed in the following situations:
  - When the operands in an arithmetic or logical expression don't have the same type.
  - When the type of the expression on the right side of an assignment doesn't match the type of the variable on the left side.
  - When the type of an argument in a function call doesn't match the type of the corresponding parameter.
  - When the type of the expression in a return statement doesn't match the function's return type.

Arithmetical Type Conversion II

- The strategy behind the usual arithmetic conversions is to convert operands to the "narrowest" type that will safely accommodate both values. Thus the rules for floating types are:

  If large operand has type *long double*, the convert the other operand to the same type. If large is double, convert the other to the same type. If larger has the type float, convert the narrower to float.

  You can represent these rule with a diagram

  ```
  float --> double --> long double
  ```

- The rules for integer types are

  ```
  int --> unsigned int --> long int --> unsigned long int
  ```

Arithmetical Type Conversion III

- When a signed operand is combined with an unsigned operand, the signed operand is converted to an unsigned value. The conversion involves adding or subtracting a multiple of $n+1$, where n is the largest representable value of the unsigned type. This rule can cause programming errors.

  Suppose that the int variable i has the value -10 and the unsigned int variable u has the value of 10. If we compare i and u using the ¡ operator, we might expect to get the result 1 (true).

  Before the comparison, however, i is converted to unsigned int. Since a negative number cannot be represented as an unsigned integer, the converted value won't be -10. Instead, the value 4294967296 is added, giving a converted value of 4294967286. The comparison $i < u$ will therefore produve 0.

  Because of traps like this one, it is best to use unsigned integers as little as possible, and never mix them with signed integers.

Type Conversions in Assignments I

The usual arithmetic rules don't apply to assignments. Instead, C follows the simple rule that the expression on the right side of the assignment is converted to the type of the variable on the left side.

```
char c;
int i;
float f;
double d;

i = c;  /* c is converted to int */
f = i; /* i is converted to float */
d = f; /* d is converted to double */
```

Other cases are problematic.

Type Conversions in Assignments II

```
int i;

i = 842.97;   /* i is now 842 */
i = -842.97;  /* i is now 842 */
```

Some assignments may be meaningless:

```
c = 10000;
i = 1.0e20;
f = 1.0e100;
```

A narrowing assignment may elicit a warning from the compiler.

Implicit conversions in C99 are somewhat different from the rules in C89, because C99 has additional types
`-Bool, long long types, extended integer, complex types.`

Casting I

- We can use explicit type conversions. For example

```
float f, fract_part;

frac_part = f - (int) f;
```

- Casts are sometimes necessary to avoid overflows:

```
 long i;
 int j = 1000;

 i = j * j;  /*overflow may occur when storing j * j */
             /*temporarily in memory */

/* do it in the following way */

 i = (long) j * j;
```