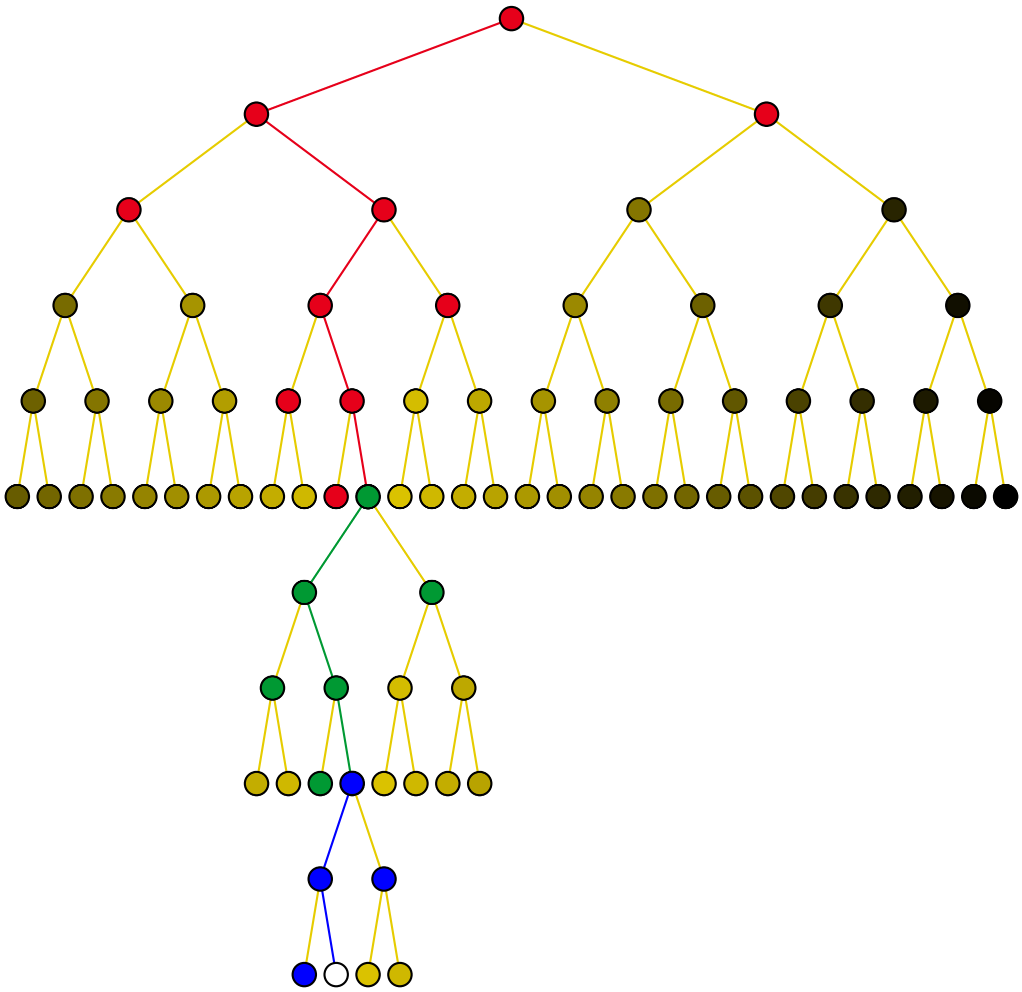
# From Java to C
# A Supplement to *Computer Algorithms*, Third Edition

Sara Baase

Allen Van Gelder



October 30, 2004

## OVERVIEW

This is a brief guide for converting the subset of Java used in the text into C. The purpose is to help readers who want to implement some algorithms, but prefer to work in C. Alternatively, if a prototype Java implementation has been developed and debugged, readers might wish to convert the program to C to improve its performance. Java has many constructs and features that the text does *not* use, and we do not address their conversion into C.

Familiarity with C is a prerequisite for reading this guide. For excellent treatments of the C language, see Gehani (1998) and Roberts (1995). This guide uses the terminology of "declare" and "define" as set forth by Gehani. On Unix systems, C library functions are documented in the "man pages"; for example, "**`man printf`**" elicits the specifications for **`printf`**.

It will also help if Section 1.2, Chapter 2 and the appendix of the text have been looked over (**`InputLib`** is in the appendix). References in this guide refer to the text. Here are some basic correspondences between Java and C.

| *Java* | *C* |
|---|---|
| `public static void main(String[] argv)` | `int main(int argc, char* argv[])` |
| `argv.length` | `argc` |
| `argv[i]` | `argv[i+1]` (`argv[0]` holds the program name.) |
| `System.in, System.out, System.err` | `stdin, stdout, stderr` |
| `print` or `println` | `fprintf` or `printf` |

## DYNAMIC ALLOCATION: new BECOMES calloc

The **`new`** operator in Java corresponds most closely to **`calloc`** in C because **`calloc`** initializes the allocated memory to binary zeros; **`malloc`** may also be used. Note that **`p`** below is technically a pointer in C. However, aside from the "constructor" call to **`calloc`**, **`p`** can and should be treated as an array.

| *Java* | *C* |
|---|---|
| `int[] p = new int[n];` | `int* p = calloc(n, sizeof(int));` |
| | `int pLength = n;` |
| `p.length` | `pLength` |
| `p[i]` | `p[i]` |

## ORGANIZER CLASSES BECOME STRUCTS

In general, organizer classes in Java (a term introduced in the text) should be structs in C. Pointers to such structs are not routinely needed, but arrays of structs are appropriate wherever arrays of an organizer class would be used in Java. In C the assignment statement operates on structs, copying the contents of the struct, not just a reference to it, so the **`copy`** methods of Java organizer classes are not needed in C. We illustrate the ideas by translating the **`Date`** class in Section 1.2 of the text, Exercise 1.1, into C.

```
typedef struct /* no nested typedefs in C */
    {
    int number;
    int isLeap; /* no boolean type in C */
    } Year;

typedef struct
    {
    Year year;
    int month;
    int day;
    } Date;

...
Date dueDate, noticeDate;
...
    noticeDate = dueDate;
    noticeDate.day = dueDate.day - 7;
```

Unlike the Java example in Section 1.2, the above assignments work as expected: Changing **noticeDate.day** does not also change **dueDate.day** because the previous assignment copied the *contents* of **dueDate**, not a reference to it.

It is perfectly proper for a function to return a struct, and this is the recommended way for a function to produce several outputs. Because certain common programming needs are much simpler to code with structs, we would not be surprised to see structs added to some future version of Java, as an extension to the eight built-in primitive types.

## ADT CLASSES

Although organizer classes should become structs, ADT classes in Java should almost always become pointer types in C. Of course, the pointer type will point to some struct, but that struct will normally be *hidden from the ADT clients*. The design philosophy is explained in Chapter 2 of the text (see Figure 2.1, e.g.). The *separate compilation* feature of C allows the privacy features of Java to be simulated to a limited extent. We'll walk through an example, using the **IntList** ADT of Chapter 2.

The general idea for a C implementation of an ADT is that the public declarations and definitions go in a "header" file, conventionally named **IntList.h**, while the private declarations and definitions go in the file **IntList.c**. Both **IntList.c** and the ADT client files "include" **IntList.h**.

A salient point is that the pointer type for the ADT is normally public (so appears in **IntList.h**), while the details of the struct to which it points are private (so appear in **IntList.c**). The *name* of the struct (its *tag*) is public, however.

In line with the foregoing remarks, **IntList.h** will have a declaration like this near its beginning:

```
typedef struct IntListStruct * IntList;
```

This *declares* (names, but does not *define*) a struct named **IntListStruct** and defines the type **IntList** to be a pointer to this type of struct, whatever it is.

Now function prototypes and global constants for the ADT may be declared in **IntList.h**. Notice that prototypes are very similar to the function headers in Figure 2.4 of the text, but the words **public** and **static** are absent. (The meaning of **static** in C is quite different from its meaning in Java, and is discussed later in this guide.)

```
int first(IntList L);
IntList rest(IntList L);
IntList cons(int newElement, IntList oldList);
extern const IntList nil;
```

A point that is not widely appreciated is that the keyword **extern** in the declaration of **nil** means that the *type* of **nil** is being declared in this statement, but the actual definition of **nil** is elsewhere, *possibly in the same file or compilation unit*. In this example, **IntList.c** will "include" **IntList.h** and the above **extern** declaration, and it will also contain the statement that defines (reserves storage for) **nil**, which becomes a globally accessible constant. This will be seen in a few paragraphs.

Finally, let us remark that all the documentation (javadoc) comments that appear in Figure 2.4 of the text should appear in **IntList.h** (as opposed to **IntList.c**). The theme of ADT encapsulation and information hiding is that **IntList.h** contains the public interface information for the ADT clients, whereas **IntList.c** may well not be available to those clients (they would link in the separately compiled **IntList.o**).

Now let us turn to the contents of **IntList.c**, which *implements* the ADT. After the customary **#include** statements, the first order of business is to complete the declaration of **IntListStruct**. Then **nil** is defined. Again, it is instructive to compare with Figure 2.4 and observe the correspondences.

```
#include <stdlib.h>
#include "IntList.h"

struct IntListStruct
     {
     int element;
     IntList next;
     };

const IntList nil = NULL;
```

We might also include the following bizarre-looking statement:

```
typedef  struct  IntListStruct  IntListStruct;
```

This makes **`IntListStruct`** a type as well as the name of a struct. It happens that the **`IntList`** ADT does not have a use for this type, but it might be convenient in some more elaborate ADTs. Please refer to the C references cited in the overview for more information about this syntactic issue.

The actual C code in the function bodies is quite similar to Java (for the subset of Java used in the main part of the text). A very important difference, however, is that pointer variables are followed by the arrow operator (**`->`**) in C, not the dot operator, to access instance fields of the object to which they point. Thus the C code will have **`L->element`**, **`L->next`**, etc. We include one example.

```
IntList  cons(int newElement, IntList oldList)
{
IntList  newList = calloc(1, sizeof(struct IntListStruct));
     newList->element = newElement;
     newList->next = oldList;
     return newList;
}
```

One of the advantages of Java for large projects that is not enjoyed by C is that function names are qualified when they are used outside their class. For example, if a program has list ADTs of several types, in C it cannot use "**`first`**" in all of them as a function name. A common solution in C is to define a two- or three-letter prefix for each ADT and begin all public names with that prefix. Even if name clashes are not a problem, it helps other programmers to determine which functions are in which ADTs, modules, or libraries. In translating the **`IntList`** ADT to C, it might be appropriate to rename "**`first`**," "**`cons`**," and so on to something like "**`il_first`**," "**`il_cons`**," and so on.

## Visibility of Functions and Data Structures

Although we don't have an example of it in **`IntList`**, it often happens that we want to define a subroutine or data structure inside an ADT and have it not be visible to the outside world. If nothing else, having every subroutine visible throughout a large program invites name clashes and accidental misuse when a function with a similar name was intended.

The mechanism in C for limiting visibility to the functions that comprise an ADT is to use the **`static`** keyword. This word is rather misleading, but we are stuck with it in C. When "**`static`**" appears in front of a declaration or definition that would otherwise be globally visible it directs the compiler and linker to make the name visible only in the current "compilation unit" or file being compiled. Thus by combining **`static`** and separate compilation, a degree of privacy for an ADT can be achieved. The correspondences between Java and C are as follows:

1. A nonpublic static Java method becomes a static C procedure. Its prototype appears in the .c file, not in the .h file, for the ADT.
2. A public static Java method becomes an extern (globally visible) C procedure or function. Its prototype appears in the .h file and its body appears in the .c file for the ADT.
3. A nonpublic static Java instance field becomes a static C variable. Its definition appears in the .c file, not in the .h file, for the ADT.
4. A public static Java instance field becomes an extern C variable. Its declaration, with the keyword **`extern`**, appears in the .h file and its definition, without **`extern`**, appears in the .c file for the ADT.

If these technicalities seem too complicated, readers should remember that limiting visibility is not necessary for correct functioning unless there are name clashes. In most moderate-sized programs, name clashes can be avoided simply by using unique names.

## Reading Input

C offers many—perhaps too many—ways of reading files. Our purpose here is just to sketch the features of C that correspond to the method of reading input described in the appendix of the text, primarily in the class **`InputLib`**.

The C functions involved are all thoroughly documented as part of the `stdio` and `string` libraries. No special library need be defined in C by the programmer. The following includes should be near the beginning of a C file that processes input. We also typedef `Cstring` for convenience.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
typedef char * Cstring;
```

A *file pointer* (type `FILE *`) is returned by `fopen`, and the standard file pointer `stdin` is automatically available to a C program. Other input functions will use the file pointer as a parameter to indicate which file is to be processed. We will designate the file pointer by `inbuf` in the examples. For example, `fclose(inbuf)` closes a file previously opened by `fopen`.

The primary input function is `fgets`, which gets one complete line of input into a buffer (a character array, previously allocated). It is the analog of `getLine` in `InputLib`. The `getLine` calls in Figure A.2 would translate something like this:

```
char line[1024]; /* A generous overestimate for any one line */
char* fgetsRetn;
...
    fgetsRetn = fgets(line, 1024, inbuf);
    while (fgetsRetn == line)
        {
        Edge e = parseEdge(line);
        ...
        fgetsRetn = fgets(line, 1024, inbuf);
        }
    if ( ! feof(inbuf)) { /* Some error occurred */ }
```

Note that `fgets` will not overflow `line` even if the input line is more than 1023 characters, so it is "safe" in this sense.

## PARSING A LINE OF INPUT AND OTHER STRING OPERATIONS

At this point we want to point out that objects in the `String` class in Java are *immutable*; their contents cannot be changed. So each `getLine` call in Java necessarily returns a new string. However, in C `fgets` writes into an array allocated by the caller, and typically this array is used repeatedly for this purpose, as in the example above. Most of the string library functions in C work in a similar fashion; the caller is expected to allocate the space for the result, then use a parameter of the call to inform the string function where that space is. An exception is `strdup`, which allocates the needed space, and returns its address to the caller.

The following table summarizes some correspondences between Java and C. We will see additional details about string library functions later.

| *Java* | *C* |
|---|---|
| `fp = InputLib.fopen(fname);` [1] | `fp = fopen(fname, "r");` |
| `line = InputLib.getLine(fp);` | `lPtr = fgets(line, len, fp);` |
| `nextToken` | `strtok` or `strtok_r` (see discussion) |
| `Integer.parseInt` | `atoi` |
| `Double.parseDouble` | `atof` |
| `String s` | `char s[`$n_{const}$`]` or `char s[]` or `Cstring s` (see discussion) |
| `s.length()` | `strlen(s)` |
| `String s = s1 + s2 + s3` | `Cstring s = calloc(len, sizeof(char));`<br>`strncpy(s, s1, len);`<br>`strncat(s, s2, len - strlen(s));`<br>`strncat(s, s3, len - strlen(s));` |
| `toString` | `sprintf` |

[1] Assumes `fname` $\neq$ `"-"`.

Since C does not allocate space for strings like Java, the programmer needs to be sure that string operations occur only within allocated space. The library functions `strcpy`, `strcat`, `gets`, and many others are unsafe because they might write beyond the bounds of allocated space. Their counterparts, **strncpy**, **strncat**, **fgets**, include a length parameter to limit the number of characters the function will write. That parameter might be set to any value by the caller, but the sensible value is the number of bytes actually available in the allocated space for the target string.

As in the Java version of **parseEdge** in the text appendix (see Figure A.3), it is the job of the C version of **parseEdge** to get the data fields out of **line**. First, we typedef **Edge** as a struct, in accordance with the earlier discussion of organizer classes.

```
typedef  struct
     {
     int  from;
     int  to;
     double  weight;
     }  Edge;
```

Now we give two versions of **parseEdge**. The first version is much simpler, and uses the knowledge of what is expected in the input line. It uses **sscanf** with parameters that are designed for the line to contain two integers, possibly followed by a floating-point number.

The second version is more complicated, and uses the general function **strtok**, to accommodate input strings of unknown format.

## parseEdge with sscanf

First, we'll give the code, then explain it.

```
Edge  parseEdge(char line[])
     {
     Edge  newE;
     int  numFields;
          numFields = sscanf(line, " %d %d %lf %*s", &newE.from, &newE.to, &newE.weight);
     if (numFields < 2 || numFields > 3)
          {
          printf("Bad edge:  %s", line);
          exit(1);
          }

     if (numFields == 2)
          newE.weight = 0.0;
     return newE;
     }
```

The **sscanf** function extracts data items from the **char** array that is its first parameter (**line** in our example). The extraction is governed by the pattern given in the second parameter, which is usually an explicit string enclosed in double quotes, as in our example. Remaining parameters are *addresses* at which to store the extracted data items; the "**&**" prefix operator can be read "address of". A full explanation of the **scanf** family may be found in the "**man** pages" of your Unix system, or in a book that describes the C library functions. We confine our explanation to this example.

The pattern string for extraction includes a space to indicate where an arbitrary number of "white space" characters may occur. A field in this string beginning with "**%**" specifies a data item to be extracted from the input. The codes we use are "**%d**" to extract an "**int**", "**%lf**" to extract a "**double**", and "**%s**" to extract a string.

But we actually specify "**%*s**" to specify that, although the string is extracted, it is discarded. That is, there is no parameter in the **sscanf** call that specifies a storage location for the string.

So the technique here is a trick to see if there is extra unexpected stuff on the line that was read and passed into **parseEdge**. If **sscanf** *does* find any significant characters after extracting three fields, it will return the value 4; this will lead to an error message.

We emphasize that a field "**%s**" must correspond to a later parameter that gives the address of a **char** array in which to store a string found in **line**. The programmer must ensure that adequate space has been allocated to store that string.

For most applications where the input is organized by lines, the combination of **fgets** and **sscanf** provides a safe and adequate method to read and parse the input data.

**parseEdge with strtok** First, we'll give the code, then explain it.

```
Edge parseEdge(char line[])
    {
    Edge newE;
    Cstring lineTmp = strdup(line);
    Cstring w1 = strtok(lineTmp, " \t\n");
    Cstring w2 = strtok(NULL, " \t\n");
    Cstring w3 = strtok(NULL, " \t\n");
    Cstring w4 = strtok(NULL, " \t\n");

    if (w1 == NULL || w2 == NULL || w4 != NULL)
        {
        printf("Bad edge:  %s", line);
        exit(1);
        }

    newE.from = atoi(w1);
    newE.to = atoi(w2);
    if (w3 != NULL)
        newE.weight = atof(w3);
    else
        newE.weight = 0.0;
    free(lineTmp);
    return newE;
    }
```

The **nextToken** function in Java translates generally to the **strtok** function in the C string library. First we make a new copy of **line** in **lineTmp** because **strtok** will change the contents of its first argument. The first call to **strtok** requests (the address of) the first "word" or "token" in the **Cstring lineTmp**. Because **lineTmp** $\neq$ **NULL**, **strtok** remembers it and returns the address of the first token. The second through fourth calls to **strtok** have **NULL** as the first parameter, indicating that the caller wants subsequent tokens from the same **Cstring** that it is remembering, i.e., **lineTmp**.

The second parameter of **strtok** specifies a list of characters that are to be treated as delimiters for the token requested. In this case we specified space, tab, and newline, a popular set of delimiters. Extra delimiters are discarded. For example, there might be spaces before the first token.

If no characters remain in **lineTmp** or the only characters that have not been converted to tokens are delimiters, **strtok** returns **NULL**. Thus the first **if** test in the code is true if **lineTmp** contains fewer than two tokens or more than three tokens. The second **if** is true if there is a third token.

Whenever **strtok** finds the requested token, it writes a binary zero character immediately after the end of it in **lineTmp** and returns the address within **lineTmp** where the token begins. Thus the returned address is a pointer to a 0-terminated string. We copied **line** into **lineTmp** so **line** would remain intact. If we need to remember the string returned by **strtok** for any length of time, we must make a safe copy of it with **strdup**. In the example, we use **atoi** and **atof** to interpret these tokens as ints and doubles, after which we do not need them. As a rule, you want to process all tokens returned by **strtok** with one of the functions **strdup**, **atoi**, or **atof**, depending on whether it is a string, int, or double (doubles might be converted to floats). Then **lineTmp** can be safely freed so its space can be reused.

Notice that it is not necessary to make a dynamic allocation of the struct **newE** in C; the **return** statement copies its *contents* back into the caller's storage area. Notice how this is different from the construction of an ADT object, as illustrated in the example of **IntList.c**.