

Intro to C for Java Programmers

1 High-level Overview

C and Java cater to different programmers. Java is a modern high-level object oriented language. Java employs program analysis to catch many errors, and garbage collection to eliminate memory leaks. Java's target programmer is doing application development in a big team, writing little software modules to plug into the Big Design. C is for low-level systems programmers. C gets compiled straight to machine language. Many of the features that make Java convenient from a software engineering point of view don't exist in C. Sometimes, this is because they didn't exist when C was created. Other times, it's because the C philosophy has been to sacrifice almost anything for the sake of efficiency and control over the hardware. And that's why systems programmers love C. If you're writing an operating system, or a language runtime, you are deeply concerned with exactly what the hardware is doing. And (apart from raw assembly language) C is the only language that gives a programmer this level of control. For instance, it is possible in C to allocate object at specific memory addresses. You can't do this in Java, because most of the time there's no reason to worry about the specific addresses containing objects.

Another huge difference is that in C there are no objects! C was created before Object-Oriented programming and so has no Classes, inheritance, interfaces or virtual methods. C has functions. Functions in C basically exist at the top level, and are the primary way to structure code. Sometimes it may be convenient to have a few global variables (roughly equivalent to `public static finals` in Java), but usually C code exposes functionality with functions. C actually has a way to aggregate different kinds of data, known as a `struct`. However, there's no protection (everything's basically public), no methods, and no inheritance. A `struct` is basically a one-off definition of a collection of data.

2 Basic Stuff

2.1 Building C code

C code is stored in text files (with the ".c" extension). The C compiler is called `cc`. And you invoke it on the command line much like `javac`. (For instance to compile "foo.c" I would type `cc foo.c`. Of course, it can't be that simple. Unless told otherwise `cc` will write the compiled code to a file named `a.out`. So you should use the `-o` flag to tell `cc` what to name the file. For instance, if I wanted to compile "foo.c" into an executable named "MyProgram" I would type: `cc foo.c -o MyProgram`. Note, that unlike in Java, C files don't have to have the same name as the output binary.

2.2 Man pages

C was created to make writing UNIX easier. This means that UNIX (and therefore Linux) is a C programmer's paradise. In fact, most C library functions have their own man pages! If you need

to know how to use a C function (say `mmap`) just get the man page (`man mmap`), and it'll tell you everything you need to know (although not always in the friendliest way).

3 Surface Differences, syntax

Now it's time to look at a chunk of C code.

```
#include<stdio.h>

void main(){
    printf("Hello World\n");
}
```

This is the hello world program. Right off the bat, the first line looks completely unlike Java. `#include` serves a purpose in C similar to `import` in Java. `stdio.h` is a *header* file that defines standard I/O (hence the name) functions. Unlike Java, in C basic I/O and string operations are actually library functions and need specific header files to be included. The definition of the main function looks similar to Java. This is because Java syntax appropriated a lot from the C/C++ family of languages. `printf` is an I/O function that outputs a string to standard out. Invoking functions in C is almost identical to Java, except in C there are NO EXCEPTIONS! Error handling in C is the programmer's responsibility and is part of the conventions of the platform.

3.1 Variables, Initialization, Assignment

C variables are declared similarly to their Java equivalents. In fact, the type names are almost identical. For instance, the following code is equivalent in C and Java:

```
int x;
int y = 4;
x = y;
```

C variables are not initialized however! In Java, if you don't assign a value to an `int`, it is automatically assigned 0. In C, declaring a variable just allocates space for it. Whatever data was in those bytes before will still be there, and if you read from an uninitialized variable, you could be getting garbage. For instance:

```
int x,y;
int z = x + y;
```

In Java, `z` would be 0. Always. In C, `z` could be anything. Sometimes it might be 0, other times it could be 437.

3.2 If

If statements in C resemble Java `ifs`. However, C has no `boolean` type. C treats all ‘primitive’ types as potential booleans. In C, a variable is false if it is 0 (interpreted as an integer), and a variable is true if it is NOT 0. So 1 is true. So is -987. The following code is legal in both Java and C:

```
int x = 0;
//something happens to x
if(x != 0){
    x = 0;
}
```

However, the following code does EXACTLY the same thing in C, but would generate a compile-time error from `javac`:

```
int x = 0;
//something happens to x
if(x){
    x = 0;
}
```

3.3 Loops

Java stole the syntax for `for` and `while` loops from C, so they look the same. Except that C interprets `ints` as booleans AND you can’t declare variables inside the arguments to `for`. For example, in Java you can write:

```
for(int count = 0 ; count < total ; count++){
    . . .
}
```

Most C compilers would yell at you for that. So, all you have to do is just declare `count` outside the loop like so:

```
int count;
for(count = 0 ; count < total ; count++){
    . . .
}
```

3.4 Functions

Functions in C are the primary way to structure code. C has no notion of classes or packages so C functions are declared at the top level (that is, they aren’t nested inside anything). Function declaration resembles Java’s except that `public`, `private`, `final`, `abstract` don’t mean

anything. First you declare the return type, then the function name, then the argument list, followed by the code (surrounded by {, }). For instance, the following code defines a C function to determine if an integer is even or odd:

```
int isEven(int num){
    if((num % 2) == 0){
        return 1;
    }
    return 0;
}
```

Looks pretty simple, huh? However, there's a catch. C compilers are lazy things and they won't allow a function to be used in a file before it was declared. So, if I wanted to use `isEven` in `main` the following code wouldn't work:

```
#include<stdio.h>

void main(){
    int uninit;

    if(isEven(uninit)){
        printf("%d is even!\n", uninit);
    }else{
        printf("%d is odd!\n", uninit);
    }
}

int isEven(int num){
    if((num % 2) == 0){
        return 1;
    }
    return 0;
}
```

This will fail to compile because `isEven` is defined after `main`. Of course, I could simply move `isEven` up so that it's defined between the `#include` and `void main(){` . However, this can get ugly as you'll have to manually track the dependencies between functions. Don't do that. Instead use what's known as a forward declaration or prototype. Here's an example:

```
#include<stdio.h>

int isEven(int num);

void main(){
    int uninit;
```

```

    if(isEven(uninit)){
        printf("%d is even!\n", uninit);
    }else{
        printf("%d is odd!\n", uninit);
    }
}

int isEven(int num){
    if((num % 2) == 0){
        return 1;
    }
    return 0;
}

```

A prototype is basically a one-liner that lets the compiler know that there's a function named such-and-such with such-and-such types. Convention dictates that prototypes accumulate between the last `#include` and before the first real code.

4 Deeper Differences, semantics

The real messy stuff of C is revealed when you get into the deeper semantics. This is where the biggest differences with Java are located.

4.1 Types and weak typing

Java and C both require type declarations, but Java is strongly typed, which means that the Java compiler makes sure that you use everything 'sensibly' according to its type. C is weakly typed, which means that the compiler expects the programmer to know what they're doing all the time. C will basically let you cast from anything to anything else. More importantly, C will automatically convert some variables from one type to another (this is known as coercion). In Java, you can't compile code with type errors in it. C is much more forgiving, and will usually just print out some warnings and continue compiling your code.

4.2 Pointers

Java has two types of variables: primitives and Objects. All Objects are dealt with through references only, and all primitives are dealt with directly. If you want to have a reference to an `int` you need to box it in an `Integer` object. C doesn't have references, but it does have pointers. A pointer is basically just a variable that holds the address of another variable. A pointer just points to a particular place in memory. Unlike Java, pointers really don't have a type. Although the syntax may suggest it, a pointer is really just an address. The compiler trusts that the programmer will

interpret the address correctly. In general, you can treat pointers like references. In Java function arguments are passed by copy if they're primitives and passed by reference if they're Objects. In C, everything is passed by copy, so if you want to pass a reference to some data (rather than a copy), you'll need to use pointers.

4.2.1 *

Pointers are created by attaching a `*` to a type name. For instance `int*` is a pointer to an `int`. And `int* ptr` is a pointer to an `int` named `ptr`. A generic pointer is usually specified as `void*`, which is basically a pointer to anything. Note that you can have pointers to pointers (which is occasionally useful), you just need to use two `*`'s.

`*` is also the dereference operator. You use it to read out the value pointed to by a pointer. For example `int val = *ptr;` will initialize `val` to have the value pointed to by `ptr`.

4.2.2 &

`&` is used to get the address of a variable. As `*` is used to extract a value from a pointer, `&` is used to extract a pointer from a value. So the following code:

```
int* ptr;
int x = -17;
ptr = &x;
```

At the end of this code `ptr` will be pointing at `x`. `&` is most often useful for getting the addresses of local data when calling a function that expects pointers.

4.2.3 NULL

`NULL` is a special value, and basically represents an invalid pointer. It is always a bad idea to dereference `NULL`. Often `NULL` is used by functions that return pointers to indicate an error. However `C NULL` and `Java null` are very different. In Java all uninitialized references are set to `null`, not so in C! So, in C not all uninitialized pointers may be `NULL`. In fact, it's a virtual certainty that some of them won't be, so don't try and do equality testing with `NULL` in C.

4.3 C Arrays

C supports arrays. However, these arrays are far less structured than in Java. First off, there's no `.length` field. The programmer has to know how large the array is (this has led to a whole raft of program bugs over the years, by the way). In C, an array is just a contiguous chunk of memory that all contains the same type, and what an array really is is just a pointer to the first element of the array. Arrays are dereferenced similarly to Java. However, unlike Java, if you reference past the end of an array you won't get an exception. You may read out garbage, or you may crash your program! The syntax for declaring an array is a little different (the `[]` occur after the variable name, not the type). Consider the following code:

```
int arr[4] = {0,0,0,0};
printf("arr[10] = %d\n", arr[10]);
```

This creates an `int` array of size 4 (initialized to all zeros). The second line dereferences the array at index 10, which would cause an `ArrayIndexOutOfBoundsException` exception in Java, but in C it just reads some random bits out of memory (on my laptop, it's usually -1073744200).

So C arrays are unsafe. How do you use them safely? Two ways: firstly, you can introduce a special value that indicates the end of the array. This is the C standard for strings (which are just `char[]`), where the special character `'\0'` terminates the string. Secondly, you can pass around the array with an `int` that holds the length of the array. This is usually the cleaner solution, and is used when handling command line arguments and byte streams.

4.4 Structs

In C, if you want to aggregate arbitrary data types, you use a struct. A struct has named fields (just like a class), and they are accessed with the `'.'` operator (just like in Java). Like functions, structs should be declared before they're used. The following code declares a struct for a Java-style string

```
struct String{
    char* str;
    int length;
};
```

NOTE the ending semicolon! Now, with this style of declaration, everytime you wanted to use a `String` you'd have to type `struct String` rather than just `String`. In order to avoid that, you can use the `typedef` keyword like so:

```
typedef struct{
    char* str;
    int length;
} String;
```

Note that the identifier (`String`) moved to the end of the structure definition. Now with this bit of voodoo, you can just use the typename `String`.

4.4.1 `->`

Usually, one is dealing with pointers to structs, not structs themselves. In that case, it is often convenient to use the `->` operator rather than having to dereference the struct pointer and then use `'.'`.

4.5 Memory Management

4.5.1 `malloc`

Java has a heap, C has a heap. In Java you allocate things in the heap with the `new` keyword. In C, you call a function named `malloc`, technically `malloc`'s signature is `void* malloc(int)`.

Unlike `new`, `malloc` doesn't care about the type of the data you're allocating. The only argument is an `int`, which holds the number of bytes you need. Of course, you don't have to figure out exactly how many bytes a particular data type occupies, you can use the built-in `sizeof` function to have the compiler calculate it for you. For instance, to allocate an array of 15 chars: `char* array = (char*)malloc(sizeof(char) * 15);` . To allocate a struct named `ext2_block` would look like: `ext2_block* block = (ext2_block*)malloc(sizeof(ext2_block));` . The following code allocates an array of 20 strings, with each string having space for `len` characters (excluding the trailing `'\0'`):

```
int len = . . .;
char** str_array = (char**)malloc(sizeof(char*) * 20);
int idx;
for(idx = 0 ; idx < 20 ; idx++){
    str_array[idx] = (char*)malloc(sizeof(char) * len);
}
```

4.5.2 free

In Java, a garbage collector is running in the background and deallocating unused objects. However, for various technical and historical reasons, garbage collection is not standard in C. Normally, the programmer is responsible for deallocating memory when it is no longer needed. The function to do this is named `free`. `free`'s signature is: `void free(void*)`, so all you do is hand `free` the pointer to the memory you want to deallocate and it'll take care of the rest. Calling `free` is fairly simple, but figuring out where to call `free` is the hard part (if you've ever heard of a memory leak, it's when the programmer should have freed something, but didn't and now can no longer access the memory). This problem can be so difficult in fact, that garbage collection was added to Java specifically to avoid having programmers worry about this.

5 Examples

5.1 Simple Cat

This code basically emulates the `cat` binary on most UNIX systems. The POSIX standard for C defines lots of handy methods for opening files. The function `fopen` is defined in `stdio.h` and as the man page reveals, `fopen` takes a file name, and a string specifying how you intend to use the file. In this case, we'll just want to read it so our access string will be `"r"`. Additionally, `fgetc` is a function that reads the next character out of an open file. However according to the man page, the type signature of `fgetc` is: `int fgetc(FILE* stream)`. Why on earth does it return an `int`? It's because `fgetc` really needs to return two values. One is the next character in the file, and the other is a boolean that tells the programmer if the end of the file (EOF) has been reached. In Java, situations like this are usually resolved with the use of an exception, C has no exceptions, so the solution is to return an `int`. Remember, C's type system is weak, so there's no

problem casting an `int` to a `char`. `feof` is a convenience function that tests whether or not the end of a file has been reached.

```
#include<stdio.h>

void main(int argc, char** argv){
    if(argc < 2){ //make sure we have enough arguments to continue
        return;
    }

    char* fname = argv[1]; //grab the filename

    FILE* my_file = fopen(fname, "r"); //open for reading

    if(my_file == NULL) //make sure the file exists
        return;

    char ch = (char)fgetc(my_file); //grab the first character

    while(!feof(my_file)){ //while there's still stuff in the file
        printf("%c", ch); //print out the character
        ch = (char)fgetc(my_file); //grab the next one
    }
}
```

5.2 Doubly-Linked List

This code describes a simple implementation of a doubly-linked list. Of note are the declaration of `ListNode`, which had to be separated from the `typedef` statement because of the self-reference to `ListNode*`.

```
#include<stdlib.h>
#include<stdio.h>

struct ListNode {
    void* data;
    struct ListNode* next;
    struct ListNode* prev;
};

//typedef here so that everyone else can just call it an Element
typedef struct ListNode Element;
```

```

typedef struct {
    Element* head;
    Element* tail;
} List;

List* makeNewList(); //creates an empty list
int listLength(List* ls); //tells you how long the list is
void insert(void* data, List* ls); //appends a new node to the list

void main(){
    List* ls = makeNewList();
    printf("ls has %d Elements\n", listLength(ls));
    insert((void*)0xDEADBEEF, ls);
    printf("ls has %d Elements\n", listLength(ls));
}

List* makeNewList(){
    List* ls = (List*)malloc(sizeof(List));
    ls->head = NULL;
    ls->tail = NULL;
    return ls;
}

int listLength(List* ls){
    Element* node;
    int count = 0;

    if(ls->head == NULL)
        return 0;

    for(node = ls->head ; node != ls->tail ; node = node->next){
        count++;
    }

    return count + 1;
}

void insert(void* data, List* ls){
    Element* elem = (Element*)malloc(sizeof(Element));
    elem->data = data;
    Element* last = ls->tail;
    elem->prev = last;
}

```

```

elem->next = NULL;

if(last != NULL){
    last->next = elem;
}else{
    ls->head = elem;
}
ls->tail = elem;
}

```

Now, assume that we want to read the first command-line argument out and store the string as a linked list of characters. It turns out that C allows for two immediate solutions. The first (and most obvious and clean) is to treat the `data` field in a `listElement` as a `char*` and allocate characters on the heap. The following code does that (and prints out the string at the end):

```

void main(int argc, char** argv){
    //A string as a linked list of characters
    //strings inputted via the cmd-line (make sure to use "" around shell
    //seperators)
    //First, we treat data as a char*
    List* ptr_ls = makeNewList(); //an empty list
    char* str;
    char* char_data;
    char ch;

    if(argc < 2)
        return;

    str = argv[1];
    ch = str[0];
    int index = 1;
    //putting the string into the linked list
    while(ch != '\0'){
        char_data = (char*)malloc(sizeof(char));
        *char_data = ch;
        insert(char_data, ptr_ls);
        ch = str[index];
        index++;
    }

    printf("Treating data as a char*\n");

    Element* node = ptr_ls->head;

```

```

while(node != NULL){
    printf("%c", *((char*)(node->data)));
    node = node->next;
}
printf("\n");
}

```

Of course, there's still something missing. Because C has no garbage collection, we need to manually de-allocate the list we created. Heres some code to do that (this code would be located at the end of the main function):

```

//clean up the mess
Element* old;
node = ptr_ls->head;
while(node != NULL){
    free(node->data);
    old = node;
    node = node->next;
    free(old);
}

free(ptr_ls);

```

It may seem wasteful to have to allocate a byte on the heap just to store a measly character. It might also seem wasteful to have to use all this indirection to get at a simple primitive value. Well, the reason it may seem that way is because it is. Remember, in C, you can basically cast anything to anything else, so it is actually possible to store the character value *AS* the data value even though data is a void* and not a char! The following code does this:

```

void main(int argc, char** argv){
    //Now we store characters AS the void* data pointer!
    List* void_ls = makeNewList(); //an empty list
    char* str;
    char ch;

    if(argc < 2)
        return;

    str = argv[1];
    ch = str[0];
    int index = 1;

```

```

//putting the string into the linked list
while(ch != '\0'){
    insert((void*)ch, void_ls);
    ch = str[index];
    index++;
}
printf("Treating data as a char\n");

Element* node = void_ls->head;

while(node != NULL){
    printf("%c", (char)(node->data));
    node = node->next;
}
printf("\n");
}

```

Hopefully, this code has sent your Java mind into cringe-mode. Yes, this violates type-safety. Yes it is ugly, but it is slightly faster and uses less space than the previous version. And it's for exactly these kinds of machine-level tweaking problems that C was designed. Again, we'll have to manually de-allocate our list, but since `data` isn't actually pointing at any heap data our deallocating code is simpler:

```

//cleaning up
Element* old;
node = void_ls->head;
while(node != NULL){
    old = node;
    node = node->next;
    free(old);
}

free(void_ls);

```