Chapter IV
Introduction to C for Java programmers

Now that we have seen the native instructions that a processor can execute, we will temporarily take a step up on the abstraction ladder and learn the C programming language, which is widely used for systems programming. C is a high-level language but you should be able to gain an insight[1] on how its statements, use of variables, parameter passing in functions etc. are translated to machine instructions. It is because C 'exposes' the processor to the systems programmer and because of the high-speed of its compiled programs, that the language is so widely used for low-level programming.

This introduction to C presumes that the reader has a good grasp of Java and, thus, puts emphasis on the differences between the two languages. These notes are not intended to be a complete tutorial of the language, just a brief summary of the main differences with Java. There are numerous good books and Internet resources on C, see the course web page for some suggestions.

IV.1 Major differences with Java

Although Java borrows much of its syntax from C, they are fundamentally different languages. The main difference is that C is *not* object oriented; the object is an unknown concept in C and a C program is a collection of *functions* (the equiv. of Java methods), which process data stored in global or local variables. Program execution starts from a special function called `main`.

Another difference is that C programs are compiled into machine code which can directly execute on the processor. Java compiles programs into a special type of code (bytecode) which cannot be executed directly, but has to be interpreted by another piece of software. The interpreter does use native machine instructions to perform the actions that each bytecode specifies; in a sense, it translates the bytecode into machine code instructions and executes them on the spot. Thus each time a bytecode is encountered, e.g. in a loop, it gets interpreted from scratch.

Memory management is another major difference between C and Java. In Java, objects are always allocated dynamically, while in C data structures (objects without methods) can be allocated either dynamically or statically, the latter meaning that their size is fixed at compile time and cannot change when the

[1] The details are covered in the compilers course.

program runs. Memory space, which is dynamically allocated to a data structure, has to be released explicitly by a C program, while in Java the memory space is reclaimed by the run-time system automatically.

Runtime errors are caught in Java and exceptions are thrown. In C there is 'no-one' to catch the runtime errors, so when they occur, the program behaves strangely and, in most cases, it crashes. This combined with the fact that the C compiler is much more liberal (it trusts that the programmer knowns what they are doing, so it does not complain for things that may seem completely wrong to a Java programmer), make C less 'safe' but a lot faster.

Finally C has the infamous pointers! Pointers are similar to Java's references, which are special variables that reference (or point) to another variable. References can be used only for objects in Java. C pointers are more general, they can point to any type of variable and they can be used in a number of ways since special arithmetic can be performed on them. We have actually seen pointers in assembly programming already. Remember the base register in data transfer instructions `sw`, `lw`? That was a pointer and we saw arithmetic being performed on it to get the next item of an array/string. This is exactly what C pointers are too.

IV.2 The Hello world program

Here is the Hello world program in C, `hello.c`:

```
1 #include<stdio.h>
2
3 /* This is a
4   comment */
5 int main(void)
6 {   // This is a comment too
7     printf("Hello world!\n");
8     return 0;
9 }
```

In DICE machines the program is compiled with the command:
`gcc hello.c`
and run by:
`./a.out`
The output is:

```
Hello world!
```

The first line of the program is a C *pre-processor* directive, which includes a *header file*, called `stdio.h`. This is similar to the Java `import` statement. Header files contain function, constant and global variable declarations that are defined

in another `.c` file. In this case, the library function `printf` is used in the program, so we need to include the header file that declares it. The (already compiled) code of `printf` is *linked* with the hello world program to create the final executable file `a.out`.

The pre-processor[2] is a program that runs just before the C compiler, modifies the `.c` file according to the directives included in the file and provides the final source code to the C compiler. All lines starting with `#` are pre-processor directives. Note that pre-processor directives do *not* end with `;` as C statements do. The basic functionality of the preprocessor will be discussed later.

Line 5 defines the function `main`, the starting point for every C program. Its return type is `int`, i.e. integer, which is used is used to indicate the program status: `0` means the program terminated correctly, while any other value means an error occurred and the returned value specifies the type of error. Line 7 is a function call to `printf`. `printf` is used for displaying formated text on the screen, which in this case is a simple string. Line 8 is the last statement of the main function, which simply returns the value `0`.

IV.3 Data types and variables

Since all operators and control statements of C are familiar from Java, they are not described here. We will see them in action later on when we will consider some example programs.

IV.3.1 Built-in data types

The following table describes the built-in data types of C and their typical bit sizes. Note that the size of an `int` is equal to the word size of the machine, rather than fixed to 32 bits as in Java. Characters are encoded using the ASCII code, therefore a `char` in C is 8 bits compared to 16 bits for Java which uses Unicode encoding. The built-in C function `sizeof()` returns the size, in bytes, of a data type (or a variable).

char	8
short	16
int	16, 32, 64 (depends on machine's word size)
long	32 or 64
float	32
double	64

There is no boolean type in C. Instead an `int` is used whenever a logical expression is required. Value `0` means false, while any other value means true.

The C data types are signed, i.e. use 2's complement encoding, by default.

[2] Also available as a stand-alone program: `c pp`.

The language supports unsigned types; just add the keyword `unsigned` before a data type name (`char`, `short`, `int`, or `long`).

C's choices in data types show that the language is not that much detached from machine code. All C built-in data types, with the possible exception of `long`, can be used directly by the processor instructions. We can clearly see that a processor's word type is visible to the C programmer by the `int` data type. As most processors do not have instructions that operate on individual bits, C uses words (`int`) for performing boolean operations. Finally, direct support for unsigned data types is provided since most processors have instructions operating on unsigned numbers.

IV.3.2 Variables

There are two categories of variables in C, *global* and *local*. Global variables are declared outside a function and can be accessed by any function defined in the same source file as the variable declaration. For a global variable to be visible to a function defined in another file, it has to be declared as an external variable (e.g. `extern int x;`) in that file. In order to protect a global variable in one file from being accessed in other files, we can declare it with a `static` prefix.

Local variables are declared inside a function. The compiler requires them to be declared before any statements. Local variables are accessible only within the function and 'live' for as long as the function is active, i.e. they are created when the function is called and destroyed when the function returns. This behaviour can be modified by declaring a local variable as `static`. This causes the variable to retain its last value when the function is called again.

Recall the use of a stack for passing parameters and returning values for functions in machine code. In addition to holding parameters and return values, C uses the stack to store the (non-static) local variables of a function. This explains why the local variables are destroyed when the function returns, because all the values on the stack are popped.

IV.4 The preprocessor

We have already seen one of the main uses of the C preprocessor, to include header files. In the hello world program, a library header file was included. These headers are stored in a special location in the file system and the compiler knows where to look for them. When user header files are to be included, the file path needs to be specified. For example, to include `header.h` located in the same directory as the `.c` file which includes it, we use the directive `#include "header.h"`. Notice that the angular brackets have been replaced with double quotes. The former imply a system header file while the latter a user header file.

Another common use of the pre-processor is to perform text substitution. When the directive `#define NAME replacement_string` appears, the pre-processor will replace `NAME` with `replacement_string`. This is a very commonly used way to define constants in C.

You can also use defines with a `NAME` that looks like a function `NAME(X)`. There can be as many parameters as needed. This is called a macro; whenever `NAME(whatever)` appears in the source code, it is replaced with `replacement_string` and `X` in the `replacement_string` is substituted with `whatever`.

Macros are used as *inline* functions. Because a function call is a relatively expensive operation (push values onto the stack, jump and link, pop values out) for simple operations it is faster to actually replace the function call with the expanded code in its declaration. Because when C was developed the compilers were not smart enough to do this automatically, pre-processor macros were used instead and are still commonplace today.

Finally the third common use of the C pre-processor is to perform conditional compilation, i.e. selectively exclude source code from being compiled. When developing a program we usually include code to hold and print out information useful for debugging. A common trick is to use a boolean variable called `debug` and at key program points, check if `debug` is true and print out the information. This is clearly inefficient because the compiled code is larger and slower as it has to check `debug` frequently. In reality `debug` does not change after the program has started, so it is known at compile time. Therefore the preprocessor can be used to hide the debugging-related lines of code before the compiler actually sees them.

The following program demonstrates the use of the above features of the pre-processor:

```
// #define DEBUG
#define N 50
#define MIN(A, B)  (A < B ? A : B)

int func(int a[])
{
    int i;
#ifdef DEBUG
    printf("Entering function: func\n");
#endif
    for (i = 1; i < N; i++)
        t = MIN(a[i-1], a[i]);
    return t;
}
```

Run cpp on the above to see what you get after the preprocessor does its job. Then un-comment the first line and do it again.

IV.5 Complex/composite data structures

IV.5.1 Structures and unions

Structures

Structures are composite data structures, similar to objects without methods. Here's how a point structure may be defined and two variables of this type declared:

```
struct point {
    int x, y;
} p1;
struct point p2;
```

Anonymous structures can also be defined, but no more variables of this type can be declared afterwards:

```
struct {
    struct point c;
    unsigned int rad;
} circ1, circ2;
// No way to declare another variable of the same struct now
```

In an expression the component data of a structure can be accessed using the structure member operator, "."

```
p1.x = 2;
dist_x = cir1.c.x - circ2.c.x;
```

Unions

Unions are declared and used in expressions similarly to structures, but they are essentially a way of interpreting the same memory space as two or more, different data types. For example the geomObject union can store *either* one point or one circle:

```
union geomObject {
    struct point;
    struct circle; // assuming circle struct above is named
} g_obj;
```

Example of using unions in expressions:

```
g_obj.point.x = 1;
g_obj.circle.c.y = 9; // overwrites the point field
```

Structures, once defined can be used exactly as built-in types. For example, they can be used as arguments or return types from functions and the whole structure will be copied in this case. This is different to Java where only references to objects are passed to (or returned from) methods rather than full copies of the objects themselves.

New types can be defined in C using the typedef statement. For example to define a new type circle: *Type definitions*

```
typedef struct {
    struct point c;
    unsigned int rad;
} circle;

circle circ1, circ2;
```

IV.5.2 Arrays

Arrays in C are significantly different to Java's arrays, although they 'look' similar in syntax. An array in C is just a contiguous block of memory containing a *fixed* number of data items of the same type. The size cannot change at run time and must be clearly defined at compile time. Here are some array declarations in C:

```
circle c_array[3];    // Array of 3 circle structures.
int arrayName[5][4];  // 2-D array
int n[] = {3, 4, 5, 6}; // Array with initialiser
```

Although the last declaration might seem to define a variable size array, the compiler fixes the size according to the number of elements in the initialiser. C can have data structures that can change dynamically, but this is done using pointers. As we will see shortly, pointers can also be used as arrays.

Other major differences with Java are that array bounds are not checked and functions are not allowed to return arrays (but arrays are accepted as function parameters). Whenever we need to return an array, we can do the same using pointers.

IV.5.2.1 Strings

Strings in C are arrays of characters (type `char`) with the, sometimes implicit, convention that the end of the string is marked with a special character `'\0'`, called the *null* character which has an ASCII value of `0`. Therefore the string declaration:
char string[10];
contains up to 9 characters because one space is taken up by the `'\0'`.

The way string literals, like `"this is a string"`, are used can be confusing to the newcomer to C. Assignment of string literals to character arrays using the common assignment operator, '=', is not permitted and equality comparisons ('==') do not work as might be expected. Library functions `strcpy(dest, source)` and `strcmp(s1, s2)`, defined in `string.h`, must be used instead. However, when a string is being declared, assignment using a string literal is allowed:

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

char str[10] = "string";
char str1[] = "another string!";

int main()
{
    char s[10] = "me", s1[11] = "me";

    //s = "Aris";  // The compiler does not accept this

    if (s == "me") // This is false!!!
        printf("s is equal to \"me\"\n");
    if (s == s1)   // false too: s, s1 are different arrays
        printf("s is equal to s1\n");
    if (strcmp(s, s1) == 0) // strcmp ret. 0 when strings match
        printf("strcmp: s is equal to s1\n");

    printf("%s\n", s);
}
```

IV.6 Pointers and dynamic memory allocation

From what we have seen so far, one might think that C does not provide much support for dynamic data structures. The capability is actually there but

one has to use pointers to use it.

We have already seen what a pointer is from a machine language point of view. A C pointer is essentially the same, a variable holding the memory address of a data type, but its use in a high-level language needs some effort to familiarise with. A pointer variable is declared as:
type *name; where *type* can be anything from built-in data types to structures; even pointers to pointers and pointers to functions (the syntax is slightly different) are allowed. From the above declaration we see that C pointers need to know what type of data they point to.

Consider the following C program:

```
1 #include <stdio.h>
2 int main()
3 {
4     int i;
5     int *p;  // p is a pointer to an int
6
7     p = &i;  // p get the address of i;
8     *p = 5;  // assign 5 to whatever p points to.
9     printf("i = %d, p points to %d", i, *p);
10 }
```

This code introduces two operators that have to do with pointers. The `&` operator returns the *address of* a variable which can then be assigned to a pointer. The *dereferencing operator*, `*`, is used to access the data that a pointer points to. So in the above program, after line 6, `*p` is equivalent to `i` both at the left side of an assignment (line 8) and at the right (line 9).

Pointers as function arguments

In C, like Java, function arguments are *passed by value*. This means that a function gets a copy of the values of its arguments rather than the original arguments themselves. The return value is also copied back to a variable (or used in an expression, etc.) when the function returns.

But in many cases we want a function to change the values of the original arguments; then we have to use pointers. When a function has a pointer to the original data it can manipulate the data directly. Note that the pointers themselves are copied by value! But this is OK, pointers are just memory addresses, so it is this address that is useful, not the pointer variable. The most common example to demonstrate the use of pointers as function arguments is the code for the swap function which exchanges the values of its two arguments.

```
#include <stdio.h>

void swap_wrong(int a, int b)
```

```
{
    int t;

    t = a;
    a = b;
    b = t;
}

void swap_right(int *a, int *b)
{
    int t;

    t = *a;
    *a = *b;
    *b = t;
}

int main()
{
    int x = 3, y = 5;

    swap_wrong(x, y);
    printf("x = %d y = %d\n", x, y);
    swap_right(&x, &y); // must use the address of operator here
    printf("x = %d y = %d\n", x, y);
}
```

pointers and structures

When a pointer, `p`, points to a structure, we use `(*p).rad` to access a structure member. As this is a very common operation[3], C has a structure pointer operator '`->`', so the previous expression can be written as `p->rad`.

IV.6.1 Pointer arithmetic and arrays

Up to now we have seen pointers that point to a single `int`, `struct`, etc. But pointers are very useful when they point to arrays. Consider the following lines:

```
int a[10];
int *p;
p = &a[0];
```

C allows arithmetic on pointers, so when `p` is assigned to point to the first element of the array, `p+1` points to `a[1]` and, in general, `p+i` points to `a[i]`.

[3] Forgetting the brackets is a common mistake; `*p.rad` means `*(p.rad)`.

Obviously `*(p+i)` is the same as `a[i]`; the compiler even accepts pointer arithmetic with array names, so `a[i]` is the same as `*(a+i)`. Note that the address of `a[1]` is actually `&a[0] + 4`, assuming 32-bit integers. The compiler converts `p+i` to the correct address because it knows what type of data `p` points to from its declaration.

The result of a pointer arithmetic expression can be stored to a pointer variable. For example, `p++` makes `p` point to the next element and `p = a+i` makes `p` point to the i-th element of the array `a`. The expression `*p++` is also very commonly encountered in C programs. It means: use the value pointed to by `p` and then increment `p` so that it points to the next element. Likewise, `*++p` first increments and then accesses the data. To increment the, say integer, pointed to by `p`, use `(*p)++`. What does `(*++p)++` do?

Special value `NULL`, a pre-processor defined constant, is used to show that a pointer is not pointing to anything. Traditionally `NULL` is `0`, so in many programs you find expressions like: if (!p) ... which means if (p == NULL) ... The above implies that comparisons with pointers are allowed in C, and are used, for example, when one pointer points at the end of an array and another pointer traverses the array, to determine when to stop.

IV.6.2 Dynamic memory allocation

So we can do all this fancy arithmetic with pointers, but for what purpose? Apart from being facilitating access to variables by reference from within a function, we haven't seen any extra functionality offered by pointers. Pointers become really useful in C when combined with dynamic memory allocation, i.e. acquiring memory space for a data structure that we do not know its size before the program is actually executing. For this purpose C provides the library function `malloc(s)` which returns a pointer to a newly allocated memory area of size `s`. `stdlib.h` must be included before `malloc` can be used. In order to make source code re-usable in other architectures, C provides the `sizeof` operator to determine the size of a data type. For example to allocate space for `n` integers one can use:

```
int *p;
if ((p = malloc(n*sizeof(int))) == NULL) {
   // Error
}
```

Notice how an if statement is used to detect the rare case when it is not possible to allocate the space required. Also notice how an assignment can be done inside an if statement.

Memory space can be released when it is not needed by calling the library

function `free`. It is always a good idea to release unused memory, even when modern computers have gigabytes of it.

IV.6.3 Pointers to pointers

Since a pointer is just a variable, we could make another pointer point to it! Would that be of any use? Most certainly yes!

Consider the case of an array of strings. It can be declared as:

```
char *strTable[10];
```

As you can see, we would have to know in advance the (maximum) table size. The only way to have both the string size and the table size be determined (or change) at run-time is to use a pointer to a pointer of type `char`. The declaration of this is:

```
char **strTable;
```

Notice that following the above declaration there is no reserved space for the strings nor the table itself. To allocate space we would use a piece of C code like the following:

```
strTable = malloc(n*sizeof(char *));
for (i=0; i < n; i++) {
  // s gets a string of length l
  *(strTable+i) = malloc(l*sizeof(char));
  strcpy(strTable[i], s);
}
```

The first call to `malloc` allocates space for a table of `n` strings. Then, space is allocated for each of the strings individually. Notice the use of pointer arithmetic. `strTable+i` points to the i-th entry of the table; to get the pointer to the first character of the string, we need to dereference it, `*(strTable+i)`. To get a specific character, say character j (remember counting starts from 0) of the i-th string, we would write: `*(*(strTable+i)+j)` or `strTable[i][j]`.