APPENDIX A

C Primer

This book assumes a basic knowledge of C. This appendix is a quick primer for people who know languages with C-like syntax, such as Java or JavaScript.

People new to C might refer to *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie, which is a concise but complete introduction to C.

Functions

C programs are built from functions. Each function takes a number of arguments and can optionally return an answer. There is no concept of objects as used in Java. The function called to start a program is called `main()`. `main()` takes as argument an array of strings specifying the command-line arguments used to start the program. For instance, if you type **`man strlen`**, man's `main` function will see `numberOfArguments` as `2` and `arguments` as `{"man", "strlen"}`:

```
void main(int numberOfArguments, char* arguments[])
{
  // Code ...
}
```

Primitive Types

C supports a limited number of primitive types: `char`, `int`, `float`, and `double`. These can be qualified by `signed` or `unsigned`, and `short` or `long`. The values you can store on the iPhone with these types are as follows:

```
signed   char             -128 to 127
unsigned char                0 to 255

signed   short int       -32768 to 32767
unsigned short int           0 to 65535

signed   long int  -2147483648 to 2147483647
unsigned long int            0 to 4294967295
```

Primitive types are passed by value in function calls: A copy of the variables passed as arguments to a function is made when the function is invoked.

Functions that do not return an answer have a "void" return type. Similarly, functions that take no arguments have a void argument list. For instance, on the Mac, `NSBeep` just makes a sound, taking neither arguments nor returning a value:

```
void NSBeep(void);
```

Basic Operators

C operators can only be applied to primitive types. For instance, unlike JavaScript strings, C strings are not primitive types, and cannot be added together using the + operator.

Function Signatures

A function's signature specifies the function's arguments' types and the function's return value type. Unless a function's signature is specified before the function is used, the compiler assumes the arguments and return values are all `int`s. For instance, to specify the cosine function takes and returns a `double`, we must specify the following signature:

```
double cos(double theta);
```

Interface Files

Function signatures are commonly placed in interface files, whose filenames have a `.h` extension. They are included into the C file using the `#include` preprocessor command:

```
#include "filename.h" // will include filename.h from the current directory
#include <filename.h> // will include filename.h from a system include directory
```

During compilation, a copy of the source file is made, in which every `#include` source line is replaced by the content of the included file.

System Libraries

Calls to system libraries are also functions. For instance, the standard `printf()` function is used to print information to the terminal. Manual pages (invoked by

typing **man 3 *function name*** from the terminal) explain the syntax of each function call. `printf` supports a subset of the format strings of `NSLog()`, which are documented on page XX.

Composite Structures

The primitive types can be combined into composite types by defining a structure. The variables named in a structure are accessed using the `.` operator:

```
struct Point         { float latitude, longitude; };
struct Point paris = { 48.856667, 2.350833 };
printf("Paris is at latitude: %g and longitude: %g\n",
                                          paris.latitude, paris.longitude);
```

Notice that the type is `struct Point`. This can be shortened using `typedef`, which lets you rename types. For instance, renaming `struct Point` to `Location` would look like this:

```
typedef  struct Point { float latitude, longitude; } Location;
Location paris = { 48.856667, 2.350833 };
printf("Paris is at latitude: %g and longitude: %g\n",
                                          paris.latitude, paris.longitude);
```

Memory Access

C programs allow direct access to memory as a fundamental operation. All the data your program manipulates is stored in memory. The location of a datum in memory is given by its address.

Pointers are variables that contain addresses. A pointer's type specifies the type of the data located at the address. Adding a `*` to a type indicates it is a pointer. The `&` operator returns the address of a variable:

```
unsigned int           i = 0;
unsigned int* pointerToI = &i;
```

A pointer can point to another pointer:

```
unsigned int** pointerToPointerToI = &pointerToI;
```

The `*` operator lets you read or write the data identified by the pointer:

```
printf("i : %d\n", i);  // prints 0
*pointerToI = 100;
printf("i : %d == %d\n", i, *pointerToI);  // prints 100, 100
```

Arrays

C arrays are blocks of memory. The array's items are arranged consecutively in the memory block. Fixed-sized arrays can be specified by adding the `[]` operator to the array declaration:

```
int prime_numbers[10] = { 2, 5, 7, 11, 13, 17, 19, 23, 27, 29 };
```

Array elements can be accessed using the `[]` operator:

```
int i;
for (i = 0; i < 10; ++i)
  printf("%d", prime_numbers[i] );
```

They can also be accessed directly with the dereference operator `*`:

```
int i;
for (i = 0; i < 10; ++i)
  printf("%d", *(&prime_numbers[0]+i) );
```

Notice that adding an integer *i* to a pointer *p* of type *t* advances *p* by *i* elements of type *t*.

Watch Out!

Nothing prevents you from writing past the end of an array in C, and nothing prevents you from reading from invalid addresses in memory. In the best case, the program will crash close to the error. In the worst case, data will be corrupted.

Dynamic Memory Allocation

Call `malloc(size)` to reserve a block of memory of *size* bytes. Call `free()` to release it:

```
void* memory = malloc(1024);
free(memory);
```

Freeing memory that was `malloc`'ed is the programmer's responsibility. Not doing so can lead to memory exhaustion, which on the iPhone will terminate your program.

Strings

Strings are arrays of `chars`. By convention, the string is terminated by a `'\0'` character. In this manner, the end of a string can be found by searching for the `'\0'` character. All C library functions that manipulate strings assume this convention, as

does the compiler for strings in the source code:

```
unsigned char* name = "John";
printf("The name %s has %d characters", name, strlen(name));
```

The `strlen(string)` function returns the number of characters in a *`string`*.

The standard C library string functions do not provide any special support for Unicode, although the UTF-8 encoding is somewhat compatible: C library functions should not mangle it too badly.

Types

The C compiler enforces type restrictions. For instance, you cannot write:

```
char* name = "john" + " " + "smith";
```

Variables' types must be declared, and once declared, they cannot be changed:

```
int    x = 5;
double y = 3.14;
x = y;
```

After running the code, `x` is 3.

Explicit type conversions are specified by using `(type)` as an operator. For instance, to create a pointer to characters from the `void*` memory returned by `malloc`, we cast to `(char*)`:

```
char* string = (char*) malloc(stringSize);
```

New Kinds of Errors

In many ways, C is a thin wrapper on top of assembly language. Good C programmers are aware of the assembly language the compiler will produce for each function they write. This makes C a good language for lean programs that run on small devices such as the iPhone. It also introduces unfamiliar errors:

- C compilers do not require variables to be initialized. Uninitialized variables contain random values.
- Local variables in C functions do not survive when the function returns. If your function returns a pointer to a local variable, the data pointed to will be overwritten some undetermined time later.