CS212 GBA

Introduction to C

Spring 2007

Preprocessor Definitions

Preprocessor definitions are definitions that are processed prior to compilation of a C program. A preprocessor is a program takes input data and produce outputs that are to be used as inputs to another program. In C, preprocessor definitions include file inclusion, macros, conditional inclusion, customized syntax definitions, and many more. Preprocessor definitions can be very powerful as they can be used to "modularize" a program, extend a language or specialize a language. However, here we will only introduce the most common used definitions in C.

#include

The keyword `#include` includes files into a C program. It takes the form of `#include <file_name>` or `#include "file_name"`. Here's an example:

```
#include <stdio.h>
#include "gba.h"
```

When reading this piece of code, the preprocessor copies the entire content of the files you indicate into the current program. For example, in the preprocessor phase, the first line is replaced by the content of the standard I/O header file, and the second is replaced by the content of `gba.h` that we defined. The brackets and the double quotation marks are essentially the same, but conventionally, `<stdlib_file.h>` is usually used for header files from the standard C library and `"user_defined.h"` is for user defined header files.

#define

The keyword `#define` defines macros that serve as alias in a C program. With `#define`, we can define constants and simple functions, or even replace syntax. It has the form of `#define id tokens` or `#define id(param) tokens`. What `#define` essentially does is replace the identifier with the replacement tokens in-line in the program. Here's an example:

```
#define PI   3.14
#define circ(r)   (PI)*(r)*(r)
```

So whenever the preprocessor sees `PI` in the code, it would be replaced with `3.14`. Similarly, `circ(r)` would be replace by `(PI)*(r)*(r)`. Note that `r` is a parameter the macro takes in. So in a piece of code like:

```
a = circ(3);
b = circ(a*PI);
```

after the preprocessor phase, it will look like:

```
a = (3.14)*(3)*(3);
b = (3.14)*(a*3.14)*(a*3.14)
```

In GBA programming, we use `#define` a lot to define constants, register addresses, and simple functions in the header files that we have provided. With these constants and macros, it is much easier to write code that is human readable.

Types

Primitive Types

The following table shows the primitive types in C. The size of an integer is actually determined by the processor and the compiler. Here in the table are the sizes specific to the GBA processor and compiler we're using.

type	size
`char`	8
`short int`, `short`	16
`long int`, `long`, `int`	32
`float`	32
`double`	64

Note that there is no type `string`. Also, types `float` and `double` are just for reference here. We will only use integral types in GBA programming. For these integral types, we can use keywords `signed` and `unsigned` to decide the signedness of an integer. `unsigned short` ranges from 0 to $2^{16} - 1$. `signed long` is a two's complement 32 bit integer. Note that `int` implies `signed int`.

typedef

Since we use the integral types described above a lot, it would be convenient if we give them names that make sense and easier to use. The keyword `typedef` helps us to do "rename" the types. For example, in the header file `gba_types.h`, we have

```
typedef unsigned short   u16;
typedef signed char    s8;
```

This piece of code defines the type `u16` as `unsigned short`, `s8` as `signed char`. So `u16 x` is equivalent to `unsigned short x` after the `typedef` statements. `typedef` is very useful in defining convenient alias for types specific to a program. We will see this to be applied again in structures later.

Structs

A structure (keyword `struct`) in C is like a class in Java without multiple constructors and methods. A struct is a user defined type that can be composed of several fields of different types specified in its definition.

Defining a struct

Let us start with an example:

```
struct point {
  s8 x;
  s8 y;
};
```

This defines a type `struct point` that has two fields `x` and `y` of type `s8`[1]. To initialize or modify the fields in `point`, we would do something like:

```
struct point p;
p.x = 2;
p.y = -5;
```

The way we access the fields in a struct is a bit like what we do with an object in Java: `struct_instance.struct_member`. Also note that `point` is only a ***struct identifier*** and the type of `p` is `struct point`, so everytime we want to create a new point, we have use `struct point`.
To get away with writing `struct point` everytime, we can make use of `typedef` and make our life simpler:

```
struct pt {
  s8 x;
  s8 y;
} point;
```

Note that we can omit the struct identifier `pt` as it is not necessary here. So now we can instantiate `point` easily:

```
point p;
p.x = 3;
p.y = 4;
```

Recursive Definition

We can also define a struct recursively:

```
typedef struct pt{
  point p;
  struct pt *next;
} point_list;
```

Here is where the struct identifier cannot be omitted, because we refer the struct itself before it's been defined.

A Note on Pointers to Structs

If we have a pointer pointing to a struct, we can use an operator `->` to access its members:

[1] `signed short`, followed from the example in the typedef section

```
point_list* list = (point_list*) malloc(sizeof(point_list));
list->p = p;     //these two statements
(*list).p = p;   // are equivalent
```