EE4144: Brief Overview of C Programming Language

EE4144

Preprocessor macros, Syntax for comments

- Macro definitions

```
// define M to be 1
#define M 1
// define FILENAME to be "file.txt"
#define FILENAME "file.txt"
// define a macro max to calculate max of two quantities
#define max(X,Y)  ((X) > (Y) ? (X) : (Y))
```

  - Whereever these macro names are referred to in the C/C++ code, the names are simply replaced by the corresponding substitutions.

- Including other files (e.g., header files). Example: `#include <math.h>`
- Syntax for comments:
  - `// Single line comments`
  - ```
    /* Multi-line comments
    * Comments that span over multiple lines
    Still within the comment
    */
    ```
    ```

Conditional compilation using the preprocessor

- Include a piece of code only under a certain condition : preprocessor directives `#ifdef` , `#endif`, `#if (defined )` , `#else` , `#elif`, etc.
  - Examples:
    ```
    #ifdef INCLUDE_FUNCTION1_DEFN
    // only if INCLUDE_FUNCTION1_DEFN is defined
    void function1();
    #else
    // only if INCLUDE_FUNCTION1_DEFN is not defined
    void function2();
    #endif
    #if (defined INCLUDE_MATH_FUNCTIONS && \
         defined USE_DOUBLES_FOR_TRIG_FUNCTIONS)
    double sin(double);
    double cos(double);
    #endif
    ```
  - \ at the end of a line is used for line-continuation (i.e., to split a long line into multiple smaller lines).
- A frequent use of conditional compilation is for include guards in header files (so as to include the contents of the header file only once even if the header file is `#include`'d multiple times). Example (e.g., for a file, file1.h):
  ```
  #ifndef FILE1_H
  #define FILE1_H
  // function prototypes, etc., in file1.h
  #endif
  ```

Variables and Datatypes

- Remember that C is case-sensitive. `int a;` is different from `int A;`
- Basic datatypes include `char`, `int`, `float`, `double`, etc.
- `signed` and `unsigned` variants
- Aggregates of variables of basic data types can be defined as structures (`struct`).
- Arrays of multiple values of a data type, e.g., `int arr[10];`
- Pointer types, e.g., `int *p_int; float *p_float;` // pointers to values of different data types are themselves different data types, i.e., an `int *` is different from a `float *`; also, function pointers are different data types.
- To define an aggregate of multiple variables, but using the same memory for each of the variables, use `union` data types. Example:

```
union {
  int i;
  float f;
} u;
```

  - `u` is a union of an `int` and a `float`, i.e., `u` can be regarded as an `int` in which case the data is accessed as `u.i` or `u` can be regarded as a `float` in which case the data is accessed as `u.f`.

Operators

- Arithmetic operators: +, -, *, /, %, ++, --, etc.
- Comparison operators: ==, !=, >, >=, etc.
- Logical operators: !, &&, ||, etc.
- Bitwise operators: ~, &, |, ^, >>, <<, etc.
- Assignment operators: =, +=, -=, &=, etc.
- Array subscript and pointer operators: [], *, &, ->, ., etc.
- Other operators: (), ternary operator (? :), sizeof, type cast operator, etc.

Operators in C and C++:
http://en.wikipedia.org/wiki/Operators_in_C_and_C%2B%2B

Functions

- A group of statements; takes a set of parameters (could be `void`) and returns a value (could be `void`).
- Function declaration is just the *prototype* (often in *header* files); function definition is the *implementation*.
- Example:
  ```
  int sum(int n1, int n2) { return (n1+n2); }
  ```
- A function can call itself (*recursive* functions). For example, factorial of a number can be calculated using the following recursive function:
  ```
  int factorial(int k)
  {
    if (k == 1) return 1;
    else return k * factorial(k-1);
  }
  ```
- Since function calls push the link register value on the stack and can also push local variables and/or function parameters on the stack, recursive functions can result in stack overflow if the function call nesting becomes too deep. For example, calling `factorial(10000);` with the recursive implementation of `factorial` might result in a stack overflow. Depending on the expected sizes of the function arguments, it might be better to use a non-recursive implementation.

Pointers

- Pointers "point" at a memory location (i.e., a pointer is like an address to a memory location). Pointer variables hold the address of a memory location. Example:
  - `int k1; int *k_ptr = &k1;` // `k_ptr` is a pointer to the memory location that contains `k1`
- An address is taken using the symbol `&`.
- Pointers are dereferenced using the symbol `*`.
- Example:

```
int k1 = 5;
int *k_ptr = &k1;
// k_ptr now contains address of k1 (i.e., points to k1)
int q = *k_ptr; // q now contains 5
int *k_ptr2 = k_ptr; // k_ptr2 also now points to k1
```

- You can have pointers to pointers. Example:
  - `int *k_ptr; int **s = &k_ptr;` // `s` now contains the address of `k_ptr` (i.e., points to `k_ptr`).

Pass-by-value and pass-by-pointer

- Arguments to C functions are passed by *value* (i.e., copies of the arguments are passed to the function). Example:
  - `int f(int i, int k);`
  - When `f` is called, for example, as
    `int i1 = 1; int k1 = 2; f(i1,k1);`
    copies of `i1` and `k1` are passed to `f`. Modifying the values of the passed arguments (`i` and `k`) inside the function `f` will not modify the values of `i1` and `k1`.
- If an argument to a function is a pointer, then the value of what the pointer *points to* can be changed from within the function using the pointer. Example:
  - `int g(int i, int *p_int);` // `*p_int` can be changed from within the function g
  - Passing a pointer as an argument is still pass-by-value (the pointer is passed by value), but approximates pass-by-reference since the value of what the pointer *points to* can be changed from within the function.

typedef

- The keyword `typedef` is used to define an alias for a type.
- Example:
  - `typedef int AliasForInt;` // `AliasForInt` is now an alias for `int`
  - `AliasForInt a;` // `AliasForInt` can be used anywhere `int` can
  - `void f(AliasForInt i);` // `AliasForInt` can be used anywhere `int` can
  - `typedef int * IntPtr;` // `IntPtr` is now an alias for pointer to `int`
  - `IntPtr k;` // `k` is now of type IntPtr, i.e., `k` is a pointer to an `int`

Function pointers

- The address of a function can be stored in a function pointer. Examples:
  - If `f` is a function declared as `int f(int);`, we can define a pointer to this function as:
    `int (*f_ptr)(int) = &f;` // `f_ptr` is now a pointer to the function `f`
  - The `&` is optional when defining a function pointer, i.e., you could also write `int (*f_ptr)(int) = f;`
- A function pointer can be used to call the function that it points to. For example, with `f_ptr` defined as shown above, you can write:
  `int k = f_ptr(3);` // call the function pointed to by the function pointer `f_ptr`
- Function pointers are commonly used to implement *callback* functions. The entries of an interrupt vector table can also be thought of function pointers (addresses to functions). Function pointers can also be used to implement generic functions. See `http://crrl.poly.edu/EE4144/generic_integrator.c` for an example.
- Typedefs can be utilized to simplify function pointer syntax, especially when function pointers are used as arguments for functions. Example:
  - `typedef float *(*SomeTypeOfFunction)(float, float);`
    // SomeTypeOfFunction is now an alias for the type corresponding to "pointer to a function that takes two float arguments and returns a pointer to a float".

Structs

- Structs can be used to define a group of variables.
- Example:

```
struct MyStruct
{
    int a;
    double d;
    char c;
};
```

  A variable of type `struct MyStruct` can be defined as:

```
struct MyStruct s;
```

- A struct can be a function parameter, e.g.,

```
void f(struct MyStruct s1);
```

- We can have pointers to structs, e.g.,

```
struct MyStruct *ps;
void f1(struct MyStruct *ps1);
```

Typedefs for structs

- A `typedef` can be used to avoid having to keep typing `struct` whenever we want to refer to the datatype.

```
typedef struct _MyStruct
{
    int a;
    double d;
    char c;
} MyStruct;
```

With this typedef, we can use `MyStruct` as an alias for `struct _MyStruct`, i.e.,

```
MyStruct s;
void f(MyStruct s1);
MyStruct *ps;
void f(MyStruct *ps1);
```

`extern`

- The `extern` keyword is used to access a symbol defined in another translation unit
- Example: if an int is defined in a file a.c as `int k;` , this int can be accessed from another file b.c as:

  ```
  extern int k;
  ```

Without the `extern` keyword, the `int k;` lines in a.c and b.c would define two separate global variables with the same name, resulting in a linker error.

static

- The `static` keyword has two different meanings depending on the context.

Static variable inside a function

- Example:
```
void f()
{
   static int a = 0;
   a ++;
}
```
- The variable `a` is initialized once at the beginning and retains its value between multiple calls to the function `f`. Without the keyword `static`, the variable `a` would be allocated each time the function is called (and would be set to 0). With the keyword `static`, the variable `a` increments each time `f` is called, so, after `f` is called 5 times, `a` becomes 5.

Static global variable or function

- With keyword `static`, a global variable or function is only visible within the translation unit (file scope) in which it is defined.
- Examples:
  - `static int k;` // global variable (defined outside any function); due to `static`, this variable is only visible at file scope
  - `static void f(int a);` // `f` is only visible at file scope

register

- Example:
  - `register int k;`
    // tells the compiler to try to put this variable in a processor register
- A register is much faster to read/write than a location in memory.
- The `register` keyword tells the compiler that this variable is used often and would be good to have in a register to reduce time for read/write operations for this variable.
- The compiler may not be able to accommodate the request to put the variable in a register and may *ignore* this keyword.
- Since the variable might actually be allocated in a register, and since you cannot take the address of a register, taking the address of a register variable (as, for example, `&k`) is not allowed.

`volatile`

- Example:
  - `volatile int k;` // tells the compiler that this variable might change outside of the control of the program
- tells the compiler to not try to optimize accesses to this variable (i.e., actually access it from memory whenever a line of code refers to it even if it appears that the variable cannot have been changed since its last access)
- `volatile` is often used to access I/O peripheral registers (e.g., a peripheral status register for an analog-to-digital converter indicating that data is ready, etc.); here, the data in these variables (memory-mapped I/O) would change due to other hardware events (i.e., outside the control of the program running on the processor)
- `volatile` is also used for variables that are changed from within an interrupt service routine; from the point of view of the main loop of the program, these variables appear to change outside its control.
- Example:
  - `volatile int new_data_received;`
    // global variable updated from within an interrupt service routine

const

- The `const` keyword defines a variable as a *constant*, i.e., tells the compiler that it will not be modified in the program.
- Example:
  - `const int k = 5;` // `k` will remain 5 for the complete duration of the program
- However, a `const` variable might be modified by using a pointer to the variable.
- Writing `const` with a pointer specifies that the data pointed to by the pointer is a constant. Example:
  - `const int *q;` // `q` is a pointer to a constant integer, i.e., `*q` cannot be modified via the pointer `q`.
- A `const` pointer to a `const` : the data pointed to by the pointer cannot be changed and the pointer value itself cannot be changed (i.e., the pointer cannot be made to point to something else). Example:
  - `const int *const q = &k;` // `k` cannot be changed via the pointer `q` and `q` cannot be made to point to something else.
- If the `const` keyword is used for some parameters in a function declaration, it specifies that the function does not change those arguments. Example:
  - `void f(const int *r1, int *r2);` // function `f` might change data pointed to by `r2`, but does not change data pointed to by `r1`

Variable and function scopes

- Variables can have various scopes (i.e., visibilities).
  - Global scope: a variable defined outside any function body, e.g.,
    `int a; // outside any function body`
    - A variable at global scope is visible everywhere (*globally*) in the sense that if there is any other global variable with the same name somewhere else, there will be a symbol conflict. To actually access the variable from another translation unit, use the keyword `extern`.
  - File scope: To restrict the visibility of a variable to only the translation unit in which it is defined, use the keyword `static`.
  - Local scope (block scope): a variable defined inside a brace-enclosed block is only visible within the block. Example:

```
void f() {
   int a = 3; // a is only visible inside function f
   if (a > 1) {
      int b = 4;  // b is only visible inside the if block
   }
}
```

- Functions can have *global scope* or *local file scope*. Functions have global scope by default and are visible from other translation units in the program. To make a function have file scope (i.e., visible only within the translation unit), use the `static` keyword. Example:
  - `static int sqr(int a) {return a*a;}` // sqr is visible only within this translation unit.