EE 3171 Lecture 7

A Brief Review of C

Lecture 7 Concepts

You should know C++ coming into this class. This is just a quick lecture to bring all that back for the lab, as well as a discussion on things relating to embedded development.

> Also, we're going to talk about some special libraries TI distributes called *TivaWare* in a later lecture. This is just good old, plain vanilla C.

Basic data types

Allocating memory

Pointers

Structures

A Few Software Engineering tips

Disclaimer: This lecture contains a high level of opinion from a variety of sources. I believe these are best practices, but it's important that you follow the corporate standard when one exists.

Basic Data Types

`int` - Integer. Size depends on architecture.

Variants: `unsigned`, `long`, `short`, `long long`

Be careful with `long long`. Not all compilers treat them equal.

What is more important here than the actually sizes is recognizing that these are not universally the same. Know the representation size on YOUR target device.

`char` - Character. 8, 16, 32 bits (depending on character encoding)

Variants: `unsigned`, `signed`. Why?

Variables of the `char` datatype in embedded systems are almost always 8-bit ASCII values. The 16- and 32-bit Unicode characters are much less common.

Portability. Different architectures default to different signed or unsigned representations of characters. When your code may be run on a different platform, and you're doing funky things with the characters, be explicit.

More Basic Types

`float` - Floating point. Size usually depends on architecture.

Usually IEEE 754 Standard (32-bit single precision, 64-bit double precision)

Floats on microcontrollers may be flaky, depending on the existence of an FPU.

The Cortex-M4 actually has a single-precision FPU.

`double` - Double precision floating point. Size is usually 2x float.

`void` - Unassociated with a particular data type.

`bool` - Boolean true/false.

Only supported by C99 compilers, and requires `stdbool.h` header.

Otherwise, just use `#define TRUE 1`…

Alternate Data Types

Define data types that explicitly identify length

Can easily be done in a header file

Like the industry-standard `stdint.h`

What does `stdint` give you?

Signed	Unsigned
`int8_t`	`uint8_t`
`int16_t`	`uint16_t`
`int32_t`	`uint32_t`
`int64_t`*	`uint64_t`*

So use these as much as possible.

* Your platform may or may not support 64 bit values.

Declaring & Initializing Variables

Varies somewhat between compilers, so be careful.

You can declare and assign on the same line:

```
int32_t j = 0;
char c = ‘a’;
float pi = 3.14;
```

You cannot declare a loop index inside the loop unless you’re using a C99 compiler.

```
for (uint32_t i = 0; i < 10; i++);// Bad news.
```

Things to watch out for:

Some compilers will not let you assign with the declaration.

But do it if you can.

Some compilers will auto-initialize, but don’t count on it.

Some compilers force all variable declarations to be at the top.

External Linkage

If a variable or function is considered “external”, that means it is:

Declared with global scope in one file, but referenced in another

extern keyword is used to inform the compiler the variable has been declared elsewhere, so do not allocate memory for it.

Example:

```
File_1.c              File_2.c
uint16_t var_name;    extern uint16_t var_name;
```

`File_1.c` will allocate a memory location for `var_name`

`File_2.c` will not allocate memory

The linker will map references to `var_name` in `File_2.c` to the address in `File_1.c`

Static Variables

`static` implies:

- Only one instance of the variable, and
- It needs to stay in memory

In particular,

- If declared in a function, the value is retained between calls

```
int Counter(void)
{
    uint16_t cnt = 0;
    ...

    return cnt++;
 }
```

```
int Counter(void)
{
    static uint16_t cnt = 0;
    ...

    return cnt++;
 }
```

Volatile Variables

`volatile` implies:

the variable may change values

without the program doing it.

Necessary for:

input devices

timers

shared memory in multi-processors

Turns off compiler optimizations for this variable.

```
B = A;     ldw r9 [A]
            stw r9 [B]


...        ...


B = A;     ldw r9 [A]
            stw r9 [B]
```

The bottom reference would likely be deleted by an optimizing compiler -- unless A was declared to be `volatile`.

Bitwise Operators

A tremendously important set of operations in embedded systems are called “bitwise”, because they allow you to manipulate individual bits in a variable.

These are the AND/OR/XOR operators.

AND = & (Good for clearing bits)

OR = | (Good for setting bits)

XOR = ^ (Good for toggling bits)

More Bitwise

Let’s say we have a 32-bit variable and we only care about the low 8 bits.

This is a more common situation than you might imagine.

We use AND with a *mask* to isolate just the bits that we’re interested in:

```
interestingBits = allTheBits & 0xFF;
```

1’s in the interesting bit locations.

More Bitwise

Now let’s say there is a 32-bit variable out there that needs bit 8 set without modifying anything else.

We use OR with a mask to isolate just the bits that I’m interested in:

```
turnedOn = turnedOff | 0x100;
```

Remember, start counting at 0 on the far right of the number.

More Bitwise

Now let’s say there is a 32-bit variable out there that needs bit 0 toggled without modifying anything else.

We use XOR with a mask to isolate just the bits that we want to change:

```
flopped = flipped ^ 0x01;
```

Looking Forward

One pattern that will be very important to us in the future is writing if statements looking at whether or not a single bit is set.

The AND operation is crucial here.

```
// Only call the function
// if bit 4 is set
if (0x10 == (allTheBits & 0x10))
{
    AnInterestingFunction();
}
```

Arrays

0-based indexing

- Off-by-one Error
- Seg Faults

Declaration:

```
uint16_t myArray[50];// Static Array

uint16_t* myArray; // Dynamic Array, needs
malloc
```

Access:

```
arrayElement = myArray[10];   // 11th element
```

Strings

Just a character array -- no basic `string` type as in C++.

Declaration:

```
char myString[80]; // 80 character static string

char* myString;   // Dynamic string

char* myString = “A constant string.”;
```

Crucially, that last one is actually *19* characters long — all the letters, spaces and punctuation, plus the ASCII null character (`\0`) that marks the end of the string.

The “ and “ are *not* included.

Be comfy with `string.h` functions

Recipe for Iterating Through a String

```
char* myString = “Bow ties are cool.”;
char* currChar = myString;

while (*currChar != ‘\0’)
{
 // Do a thing to
 // the current character
 currChar++;
}
```

Structures

Custom-designed data types

```
struct struct_name
{
   uint32_t whatever;
   char somethingElse;
  struct struct_name* next;
}
```

Then to instantiate:

```
   struct struct_name myStruct;
```

Generally like typedefs to simplify things.

```
   typedef struct struct_name
   {
      …stuff …
   } MYSTRUCT_T;
```

Then instantiation is just: `MYSTRUCT_T myStruct;`

There are a couple hidden conventions here. The `typedef` name is all-caps and ends with the `_T` suffix. These are not required, just conventions.

Unions

Allows you to access a single memory location by different variable names, even if they are of different types.

Can be very useful in embedded programs

- I/O port data often arrives 1 byte at a time (slooooowly!)

- Moving data between buffers is much more efficient as words

Example:

```
union
{
  uint32_t   u32[6];
  uint8_t    u8[24];
} msg_buffer;
```

The actual size of the union is dictated by the largest element.

Know your processor’s byte-endian convention to know where things overlap.

Functions

Reusable bits of code invoked from your main program.

```
return_type FunctionName(arg_type arg1,
arg_type arg2, ...)
```

e.g., `int FindMax(int32_t num1, int32_t num2)`

Note: By default, all arguments are passed by value, not by reference!

What does this mean? Next slide, please!

Passing Arguments

Let's say I have a little function:

```
void FindBiggest(int32_t anInteger, int32_t
anotherInteger, int32_t theBigOne)
{
  if (anInteger > anotherInteger)
   theBigOne = anInteger;
  else
   theBigOne = anotherInteger;
  return;
}
```

If `anInteger = 6`, `anotherInteger = 8` and `theBigOne = 0` when I call the function, what will be the value of `theBigOne` when it returns?

The answer: `0`.

Why? Again, next slide, please!

Pass By Value

By default, C programs pass all values to functions by *value*.

This means a copy is made and sent to the function. Changes to the copy are not committed back to the original variable.

What if you want to change the original variable?

Three Options:

- Pass by Reference (How Java does it)
- Global Variables
- Assign it using the returned value.

Pass By Reference

Step 1: Define the function prototype differently:

```
void FindBiggest(int32_t anInteger,
int32_t anotherInteger, int32_t*
theBigOne);
```

Step two: Call the function differently:

```
FindBiggest(speed1, speed2, &maxSpeed);
```

This passes a *pointer* to the variable, which effectively means you're modifying the original variable.

Global Variables

If you have a variable defined outside the scope of the main function, you can simply use that variable in the main function and in other functions.

Don’t bother even putting them in the argument list.

In most environments, global variables are frowned upon.

In embedded programming, they are used a lot.

Be very thoughtful about how you should apply the `volatile`, `const` and `static` modifiers to these variables.

Return Values

Change the function again:

```
int32_t FindBiggest(int32_t anInteger, int32_t
anotherInteger)
{
 if (anInteger > anotherInteger)
  return anInteger;
 else
  return anotherInteger;
}
```

Then call the function as:

```
maxSpeed = FindBiggest(speed1, speed2);
```

Obviously only works for one return value.

Well… it’s complicated. There are ways of getting around this which are outside the scope of this class.

Variables

Variables are simply a way to name the memory addresses that contain certain types of data.

An `int` contains an integer, a `char` contains a character and so on.

A *pointer* is just another kind of variable that contains an address.

Pointers

Declaring a pointer:

```
 int32_t* pThing;
```

Assigning a pointer:

```
 pThing = someAddress;  // But
you’ll almost never do this.
```

Assigning to the variable pointed to by a pointer:

```
*pThing = someValue;  // Called
dereferencing the pointer
```

Pointers

What is the value of `pThing`?

```
int32_t* pThing = (int32_t *) malloc(sizeof(int32_t));
```

The system decides. We don't necessarily know.

Is this assignment okay?

```
pThing = myIntegerVariable;
```

No! (Note: It is *technically* legal [as long as the `sizeof()` the RHS = `sizeof()` LHS.])

Do these statements do the same thing?

```
*pThing = myIntegerVariable;

pThing = &myIntegerVariable;
```

Answer: Sort of. Short-term, yes. Long-term, no.

Various and Sundry

Pointer arithmetic

```
int32_t counter = 100;
int32_t* pCounter = &counter;

pCounter++;
```

What is the value of `pCounter`?

What is the value of `*pCounter`?

Pointer arithmetic might be one of the worst of the necessary evils in C. Why?

So *very easy* to screw up.

So *very valuable* for iterating through arrays.

Pointer arithmetic is the only reason pointers have types. Otherwise, an address is an address.

Various and Sundry

Pointer declaration syntax:

Some people use:

```
int32_t *pThing;
```

Some people use:

```
int32_t* pThing;
```

Various and Sundry

Pointer declaration syntax:

Some people use:
```
int32_t *pThing;
```
← These people kick puppies.

Some people use:
```
int32_t* pThing;
```
← These people work at Google.

Various and Sundry

Okay, so one of these is an exaggeration. The other one is true!

Pointer declaration syntax:

Some people use:

```
int32_t *pThing;
```

These people kick puppies.

Some people use:

```
int32_t* pThing;
```

These people work at Google.

Various and Sundry

Linking files with `#include`

- `<file.h>` is some kind of system library.
  - The compiler knows where to look.
- `“file.h”` is something you created.
  - Supply the relative path.
- `“file.c”` is allowed, but terribly bad form.
- Always put a sequence of:

```
#ifndef FILE_H
#define FILE_H
…
#endif
```

  in your `.h` files.

Various and Sundry

Libraries you should know:

- `stdlib.h`
    - Lots of fundamental things.
- `stdio.h`
    - `printf`, `fprintf`, etc.
- `math.h`
    - exponents, square root, etc.
- `time.h`
    - Anything related to time.

Various and Sundry

Generating pseudo-random numbers:

Include `time.h`

```
srand(time(NULL));
myRandomNumber = rand();
myRandomNumber = rand() % 5;
```

Do NOT use this method on the Tiva!

However, it may be of more use to grab some of the low-order bits of a free-running counter.

Besides, `srand` on the some processors always returns the same value.

Printing

Use `stdio` and functions like `printf` and `fprintf`.

Outline of a Generic C Program

Begin with a block comment

Name, date, version, purpose and so on...

Then all your `#include` statements and `#define` statements.

If you have global variables, they go here too.

Then function prototypes (if they're not in a `.h` file).

E.g., `int32_t UsefulFunction(int32_t myArray);`

Outline of a Generic C Program

Then the `main()` function.

Normally will be `int main()`.

Hey, Java programmers: It's not `void main()`.

Might occasionally see:
`int main(void)`,
`int main(int argc, char* argv[])` or
`int main(int argc, char** argv)`

In embedded programs, there are no arguments, so leave out the `argc`/`argv` stuff.

Outline of a Generic C Program

Inside the `main()` function:

- Local variables (try to define all of them at the top)
- Your program behavior.
- Last line: `return 0;`
    - Some people leave this off, but it’s bad form.

Outline of a Generic C Program

After the main:

Functions.

Some people put the functions before the `main`. My *opinion* is that they should go afterwards.

Here Be Dragons

This is the break between the factual and the opinionated.

We begin with issues of readability and then go into other issues of things the language will let you do, but you never should.

Braces

Bracing style in this class is whatever you want it to be. However, you will probably be told which style to use at your company. Just deal with it.

As a survey:

> 1TBS: Opening brace goes on same line as conditional, final brace goes on new line.
>
> Allman/BSD style: Opening brace goes on new line, at same tab depth as control statement.
>
> > This one is my preference, and is the default for Visual Studio (among other editors). On the other hand, MS gave us Hungarian Notation, so maybe it's not all sunshine and roses.
>
> Whitesmiths style: Opening brace goes on new line, at same tab depth as body of the code segment.
>
> GNU style: Opening brace on new line, indented two spaces. Code is indented another two spaces.
>
> And some others that are minor variants.

Indentation and Line Spacing

Indentation – Again, nobody can agree.

Every editor and printer on planet has different tab settings.

Therefore, never use tabs to indent, use spaces to indent!

Use (2…4) spaces per level, but *Be Consistent*!

Most modern editors will insert spaces when you hit the Tab key, so this isn't such a big deal anymore.

Line Composition

Use parentheses and white space as needed to improve readability and clarify the meaning of the statement.

Even when the compiler does not need them, e.g.:

```
a = b+c*d+e*f;   // difficult to read

a = b + c*d + e*f;   // easier due to spaces

a = b+(c*d)+(e*f);   // easier due to parens
```

Line Spacing

Excessive blank lines spread the code over more pages and make it harder to read as a whole.

> We're talking *excessive* blank lines. Use whitespace liberally!

As a general rule:

> Skip lines between logical blocks of code
>
> Do *not* skip lines within a logical block of code
>
> A "logical block of code" might only be two or three lines long.

Commenting

The `/*…*/` notation is standard in C

The `//…` notation is a C++ addition (not standard in C)

But, most modern C compilers seem to have adopted it, so go ahead and use it.

I prefer to use standalone full-line comments <u>everywhere</u>. Examples include:

At the beginning of a logical block of code or to describe the block as a whole.

Full-line comments should preceded by a single blank line.

I don't like end-of-line comments, especially if your explanation requires multiple lines of end-of-line comments.

If it causes the line to wrap, make it a full-line comment. It's obviously important.

On a more idealistic note, comments reveal the *purpose* of the code. They should never simply rehash what we can understand by reading the code.

Things You Can Do, But Shouldn't

Conditional operator:

```
x ? y : z;
```

I would explain it, but it's stupid. Don't use it. Ever.

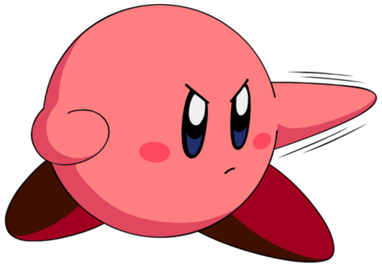
Don't use `goto`. (unless it's absolutely necessary)

Always remember that variables are stored internally in binary.

Use shift operators (<< and >>).

```
<(^_^)<   <(^_^)>   >(^_^)>   <(^_^)>
```

Dance, Kirby, dance!



Things You Can Do, But Shouldn't

Assignments in conditional statements

```
    if (fopen = RandomValue())…
```

Use "Magic Numbers"

```
    if (myVariable == 9)…
```

Use integers like bizarre Booleans

```
    while (loopCounter) {…loopCounter--;}
```

Drop the braces on one-line loops/if statements.

```
    if (stopValue == STOP)
      printf("Stopping!");
```

Use `printf` as your debug tool.

If you absolutely have to, use `stderr` instead of `stdout`.

There is a good exception to this first one: spawning new processes with `fork()`.

The difference between `stderr` and `stdout` is that `stdout` is usually buffered. If there are any timing issues, you will miss them using `stdout`.

Things You Can Do, But Shouldn’t

Recursion:

- Elegant representation of *recurrence relation* functions
- Bad idea in real-time embedded software
  - Each iteration = a procedure call (expensive in both time & memory)
  - It’s too easy to write a non-terminating recursion
- A simple loop is faster & less of a memory hog

Complex Expressions involving “`++`” or “`--`”

- e.g. `++j  = --i + i++ * --j;`
- Compiler may allow all sorts of crazy unary ops

Things You Can Do, But Shouldn't

Leave out the `return` statement in `void` functions (or the `main` function).

The key is: just because the syntax lets you do it, doesn't mean you should.

Summary

C is a very flexible and sophisticated language

- Very Powerful ⇒ Very Dangerous
- A great deal of effort has been spent on “taming” C
- Some companies have very detailed & strict style guides
  - Google is an extreme example

In embedded programming, you need to limit what you do

- finite amount of memory
- finite amount of time

Summary

Remember that in C, it's still true that TMTOWTDI

- Choice 1: simple and obvious, but slow
- Choice 2: clever & fast, but not so obvious
- How do you choose?

The K.I.S.S. Principle is usually your best option

- Not the way basic CS courses are taught
  - Taught to handle complex code to solve complex problems
  - Tend to write complex code to solve simple problems
- Embedded programming is a major shift in paradigm
  - Need to write simple code to solve simple problems
  - Maybe even simple code to solve complex problems

This is largely Dr. Kieckhafer's commentary, which I'm leaving here because I tend to agree with it (mostly). The simple solution is usually the best.

Similarities

Other than aforementioned, C works mostly like Java and C++. If you try it, it will probably work.

Pointers are the key. If you can understand pointers, you’re ahead of the game.

Last Things

Obviously there's a lot more to C than covered in 46 slides.

Lots of good tutorials online and in books.

C hasn't really changed in 20+ years.

It's a hugely valuable skill for any engineer.

Next, we'll talk about TivaWare and how it makes your Cortex-M4 programming easier.