ECE 252: Systems Programming and Concurrency Spring 2021

Lecture 1 — Introduction and Our C Toolkit

Jeff Zarnett *2021-05-02*

About the Course

We'll start by reviewing the highlights of the class syllabus. Please read it carefully (it is available in Learn under Content → Overview). It contains a lot of important information about the class including: the lecture topics, the grading scheme, contact information for the course staff, and university policies.

Some Background on Operating Systems

An operating system (often abbreviated OS) is a piece of software that sits between the hardware of a computer and the applications (programs) that run on that computer. The OS does many different things and often has many (occasionally-conflicting) goals.

To use an analogy, you may wish to think of the operating system as the "secretary" of the computer. It is a resource manager, and it allocates resources as necessary. The most significant difference between a secretary and an operating system is that while a secretary can work in parallel with you, operating systems often work when programs are sleeping, and operating systems must put themselves to sleep to allow programs to execute. It is responsible for seeing to it that other programs can operate efficiently, providing an environment for other programs, and collecting and reporting data about what is and has been happening.

An operating system is also responsible for resource allocation. In the real world, the resources we have to work with, such as CPU time or memory space, are limited. The OS decides how to allocate these resources, keeps track of who currently owns what, and, in the event of conflicting requests, determines who gets the resource.

The OS usually enables useful programs like Photoshop or Microsoft Word to run. Any computer has various pieces of hardware, such as the CPU, memory, input/output devices (such as monitors, keyboards, modems). The OS is responsible for abstracting away the details of this, so that the authors of programs do not have to worry about the specifics of the hardware. Imagine how painful it would be to write even a simple program, like the Hello World example, if we had to write our program differently for every combination of hardware.

In most cases there will be multiple programs running on the computer. This implies the sharing of various resources. When this is the case, there is the potential for conflicts to arise. An operating system creates and enforces rules to make sure all the programs get along and play fairly. Of course, not all interaction between programs is competitive; sometimes they want to co-operate, and the OS helps them do that, too.

Systems Programming

The operating system is, however, just the backdrop to what we are going to do in this course. We will often be doing *Systems Programming*. This is the next layer above the operating system itself. System programs ship with the operating system and they are useful tools, but they are not part of the kernel.

Some examples of things that might fall under systems programming (from [SGG13]):

- **File Manipulation** - Programs to create, delete, copy, rename, print, and manipulate files and directories.
- **Communication** - Remote login capability, file transfer, download utilities, and messaging.
- **Processes and Thread Management** - Creating processes and threads, working with them, interacting with others, and cleaning them up.

Programming at this level is more difficult than writing regular programs. It may require knowledge of the hardware, or perhaps programming facilities like debugging are limited. Furthermore, systems programs must take concurrency into account; multiple (regular) programs may try to use the systems program at the same time. An e-mail client may choose to refuse to allow two instances of the program to be open at the same time, but users would not accept a system that did not permit more than one file operation at a time.

A key thing that distinguishes systems programming is that we want to do certain operations that involve the operating system. In short, there are some things that the operating system does not allow programs to do, and what they have to do is ask the operating system to do it instead. Learning how to use the functionality provided by the OS is a key part of writing any complex program (i.e., one that is non-trivial).

Concurrency

The previous section mentioned concurrency, but didn't really introduce what it is. A program is said to be concurrent if it can support two or more actions in progress at the same time; it is parallel if it can have two or more actions executing simultaneously [Bra09]. Soon enough we will spend a great deal of time examining the differences between parallelism and concurrency in the program.

It is already the case that many programs you use are to a greater or smaller degree concurrent. Depending on your level of programming experience, you may have already written a concurrent program, intentionally or without knowing it. We will learn about how to take a program and make it concurrent, as well as how to write it with concurrency in mind from the ground up.

Concurrency comes with a number of advantages, chief amongst them the ability to speed up your program and get more work done in the same amount of time. It does, however, come with a number of drawbacks. On top of being somewhat more difficult to write, it is more difficult to be sure that the program is correct. We have tools to find and deal with the problems that arise, but we will need to use them carefully.

To give you a taste of the sort of problem that we may encounter, consider a program that performs a simple calculation given some input. If the program has a concurrency problem, then the answer could be (1) consistently the wrong answer every single time, (2) different on consecutive runs with the same input, or (3) correct some of the time but incorrect some of the time. As you can imagine, none of these options are acceptable. If you bought a simple four operation calculator that only gave the correct answer 99% of the time, would you be satisfied with your purchase?

Without going too much into the statistics of this, if your company's payroll system got it right 98% of the time, and the company paid you every 2 weeks (ie 26 pay periods a year), then statistically they would likely fail to pay you within your first two years working there. Ouch. Correctness matters.

Good to Great

You know the what of this course, but how about why? This is a mandatory course for computer engineering students (which is most of you). Fine, but what will you get from it? To steal the title of a book: "Good to Great".

The purpose of the course is to give you the tools and skills to be a *great* programmer. What does it mean to be great, though? Good programmers implement a feature efficiently; great developers improve the program architecture to solve a whole class of problems at once. Good programmers can use a network communication library; great programmers can write that library. Good programmers can find answers to their questions on Stack Overflow; great programmers can write those answers.

Past students will confirm that employers ask interview questions about the topics in this course. If you master these topics, it opens all kinds of doors for you in the workplace. Most likely, you don't want to be a junior or intermediate developer all your career. Whether you want to advance your career down the individual contributor track or the technical lead and management track, you need this knowledge and these skills.

You want to be a great programmer, don't you? Still, I have to be honest. This course is going to be hard.

Our C Toolkit

This is a programming course, after all, and we will be programming in C. There's a baseline assumption that you have experience with a C-like language (if not C itself), such as C++, Java, C#. While the purpose of this class it not to teach you how to program in C, past experience (from ECE 254) suggests that it is helpful to cover a number of key concepts and conventions in C programming to make sure that everyone is on the same page, and to streamline explanation of future code examples. It is also probably desirable to put all of these things together in the same place for quick reference.

C is a procedural language: that is to say it has functions (but that doesn't make it a functional programming language). It is not Object-Oriented or anything like it. We will write functions and functions work as they do in all C-like languages. They have a return type, function name, and then a list of arguments (or parameters) in parenthesis.

```
int function_name( double arg1, int arg2, char arg3 ) { ... }
```

Header files, whether our own or for provided libraries are imported into the program using a precompiler directive such as below, where two of the standard headers are imported.

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

Many of the functions we wish to call are included in these header files, but there are many many more. If you are attempting to use a library function and the compiler says it doesn't know what you mean, then first check your spelling, as well as the number of parameters... and then check to be sure you have included the correct header file(s). If you get a compiler warning about "implicit declaration" of a function it means that a header is missing but the compiler can guess about what you want. You should still fix this by including the right header, which you might have to google.

C has no `bool` type by default. In many contexts an `int` is used where 0 represents false and 1 represents true. Or you can include the header `stdbool.h` which provides you the type as well as the explicit values of true and false.

But speaking of the compiler not knowing how to find a function you might have the compiler tell you that it can't find the definition for `bar` when compiling `foo`.

```
int foo() {
  return bar( 7, 11 );
}

int bar( int v1, int v2 ) {
  return v1 * v2;
}
```

What gives?! It's RIGHT THERE!

Some of the rules of the C compiler date back to the original design of the language and in these ancient and primitive times the compiler was unwilling to take two passes compiling source code because it would take too long. This can be solved in one of two ways. Either move `bar()` in the source so it appears before `foo`, or, instead, provide a function prototype for `bar()`.

```
int bar( int v1, int v2 );

int foo() {
  return bar( 7, 11 );
}

int bar( int v1, int v2 ) {
  return v1 * v2;
}
```

The function prototype appears earlier in the file and promises the compiler that at some point there will be an implementation of this function, but be aware that it exists. This will now allow compilation to succeed. Function prototypes are frequently declared all at the top of the file.

In another ancient rule based on no second passes, versions of C before the C99 standard do not allow declaring of variables in a function anywhere but at the very start of the function. This means that in the example below, `loop1` will not compile but `loop2` will. Fortunately we will always assume (at least) C99 standard so this problem is avoided.

```
void loop1() {
  for ( int i = 0; i < 10; i++ ) { ... }
}
void loop2() {
  int i;
  for ( i = 0; i < 10; i++ ) { ... }
}
```

Comments. In C comments always begin with `/*` and end with `*/`. The style of comment like `// This is a comment` may be permitted by your compiler but this is not supported everywhere. Please use C-style comments.

Structures. Perhaps the most striking difference in C for those coming from an Object-Oriented Programming language is that C has no such concept. There are no classes, and no class hierarchy, no interfaces or abstract classes, none of that. There are no access control modifiers (`public, private`, etc).

What we have instead for programmer-defined types is the *structure*, the `struct`. A structure is the forerunner of the class but is much simpler: it is a grouping of variables, and nothing more. A structure is composed of a group of variables as one might expect. A simple structure representing a 3-dimensional cartesian coordinate:

```
struct point {
  double x;
  double y;
  double z;
};
```

In the example, all three elements are defined as `double` but there's no reason why they have to be the same (they can be anything). The following short example shows creating a `struct point` and assigning some values. As you would expect, the attributes of the structure are accessed with the dot operator (`.`):

```
struct point p1;
p1.x = 1.9;
p1.y = 2.5;
p1.z = -1.0;
```

Type Names. In the previous example when we want to use the `point` structure as defined, we have to write `struct point` as the full name of the type. This can result in a lot of typing of the keyword `struct` such as in a function like this:

```
struct point add( struct point p1, struct point p2 ) { ... }
```

Fortunately, we are provided another keyword that will reduce the amount of typing that we have to do. This is the `typedef` keyword. The keyword precedes a structure definition, and after the definition there is the new name you'd like to use.

```
typedef struct point {
  double x;
  double y;
  double z;
} point_t;
```

This would allow you to rewrite the previous function call as a more compact form:

```
point_t add( point_t p1, point_t p2 ) { ... }
```

You can (if you want) use `typedef` on other things, such as `typedef int pid_t` which allows you to use your own names for certain things if you have a reason (as we'll see later).

A small note on the naming convention: it is now discouraged to use the `_t` ending on the name of a programmer-defined type. Hopefully the UNIX police will not arrest me based on the previous example. Nevertheless, when we do some future examples, we will see that there are types (like `pid_t`) where this convention holds. But regardless, it's just convention, not law, and using it (or not using it) isn't wrong.

You may also notice that in the definition there is the `point` name after the `struct` keyword and this is not used anywhere once we have finished the `typedef` definition. It can be omitted and the definition would just be:

```
typedef struct {
  double x;
  double y;
  double z;
} point_t;
```

And this would be exactly equivalent to the previous definition.

Memory Allocation, Deallocation, and Pointers Your program has three kinds of memory: global variables, stack, and heap. Memory that is allocated on the stack is local to the function currently running and will automatically be thrown away when the function returns. Memory that is allocated on the heap is explicitly asked for and must also be explicitly returned as well. Stack memory is somewhat limited; heap is very large (although not infinite). In some programming languages one may get away without knowing the difference, but this is C...

Stack allocated memory is fairly simple. In the example below, `bar` is a variable allocated on the stack. The lifetime of this variable is just as long as `foo` runs.

```
void foo() {
 int bar;
 ...
}
```

It's important to remember that the variable `bar` has no initial value. It has to be initialized to some value, otherwise it simply contains garbage. This is a key step and a frequent source of error in programs. Even students in 4th year with a great deal of programming experience sometimes forget to initialize variables and spend many hours debugging their programs.

Global variables are variables declared outside of any function. Although they can be used for good, excessive use of them is a warning sign of poor program design. Global variable space is also not infinite. While we might use global variables in the class for simple examples or to make an exam question simpler, in real life it is usually better to avoid them. Global variables may be initialized for you to zero or something like it, but it is best not to count on this and to initialize the variables.

In C, heap memory is allocated in a very simple way: you ask for the amount of memory that you want. The function for this is `malloc()` and it takes one parameter – the number of bytes of memory that you want. So if we want to allocate an integer on the heap, we need to call `malloc()` and request the number of bytes that correspond to an `int`. But how many bytes is that?

You may think the answer is 4 bytes, but a quick look at the C standard says that while there are are exact sizes for some types, other types have only a minimum size. The rest is implementation specific. That is, it can vary from one system to another. Yikes! Fortunately we have `sizeof`, which ask the compiler to help us out here and substitute in the correct size where it is needed. So we would call `malloc()` with the argument `sizeof( int )`. At run time, the correct amount of memory will be requested.

This is also very helpful for when you want to allocate a structure. Instead of needing to work out the size yourself (well, this is three `doubles`, and a `double` is 8 bytes...), `sizeof( point_t )` will allocate the correct size... as a nice bonus, if ever the definition of the structure is changed the amount of memory requested is updated automatically.

The return value of `malloc()` is a pointer to the memory that is allocated. Unlike C++, the value does not need to be cast (and probably shouldn't be). Consider the following allocation:

```
void example( ) {
  int * x = malloc( sizeof( int ) );
  ...
}
```

The variable `x` is a pointer to an integer, and it is allocated on the stack. The value contained in `x` is the address of the memory that was allocated by `malloc()`. That memory currently contains garbage because it has not been initialized. To initialize it, of course, we just assign it a value as below:

```
void example( ) {
  int * x = malloc( sizeof( int ) );
  *x = 0;
}
```

When we are finished with memory, it is proper to deallocate it using `free()`. This function takes a pointer to the memory that is to be deallocated and lets the operating system know that we are finished with this memory. We don't have to specify the number of bytes being returned, because it will be the same as was originally allocated and this is tracked by the system's memory allocator. So the complete example:

```
void example( ) {
  int * x = malloc( sizeof( int ) );
  *x = 0;
  ...
  free( x );
}
```

If we forget to deallocate some memory that is allocated, this is a *memory leak* and this is a bad thing. It may not have immediate bad effects, but it is wrong and in the long term (such as a program that runs 24/7) memory leaks will slow down the program and can eventually lead to a crash. When memory is being allocated, you should always take a moment to consider where it is to be deallocated. The call to `free()` should be when the memory is no longer needed (ideally as soon as possible after this is known), but it may be in a different function altogether.

A provided function `a()` might, for example, allocate the memory and return a pointer to it for your use and it is then your responsibility to deallocate this memory when you no longer need it. Reading the documentation (or source code) of a function will tell you if this is the case. Or, you may pass memory between two (or more) functions you have written in your program; just be sure to check the "flow" of the data to be sure that anything that was allocated gets deallocated.

It is worth noting that `free()` just marks the memory as being available; it doesn't actually erase what is there. So you might be able to use memory after marking it as free. Murphy's Law suggests that this will work just fine when you test your code but will crash the program every time when it is in production (or the TA is marking it). To reduce the likelihood of this, some coding conventions ask you to assign a pointer to 0 after it has been freed. This guarantees that a future attempt to use that same pointer will crash.

Also noteworthy: if you try to call `free()` on the same pointer twice, this will very likely crash your program immediately. Similarly, if you try to free memory that was allocated on the stack rather than on the heap, this also leads swiftly to a crash.

Dereferencing, Address-Of, and The Arrow. You will recall from some basics about how pointers work that the `*` symbol is used to declare something a pointer, and to *dereference* the pointer (follow the directions to the memory on the heap). This is also used when we are expecting a value. If we have a function defined like this:

```
int work( int x, int y ) {
  return x + y;
}
```

And our variables are allocated on the heap then we know how to use them in a function call as below.

```
void baz() {
  int* a = malloc( sizeof( int ) );
  *a = 2;
  int* b = malloc( sizeof( int ) );
  *b = 4;

  int z = work( *a, *b );
  free( a );
  free( b );
}
```

But suppose the situation is reversed; the function `work()` is now defined like this:

```
int work( int * x, int * y ) {
  return *x + *y;
}
```

And our variables `a` and `b` are stack allocated? To invoke this function we use &, the address-of operator.

```
void baz() {
  int a = 2;
  int b = 4;

  int z = work( &a, &b );
}
```

In short, the function is expecting a pointer to an integer but we have a regular integer allocated on the stack. We can feed the function `work()` a pointer to that stack-allocated memory using the address-of operator. It needs to be used with caution, however, because of the fact that stack allocated memory does not outlive the function it is allocated in... and sometimes does not outlive a particular iteration of a loop. We will use this in a number of examples, but if your program is not behaving as expected you should check places where this is used as it can be a source of issues.

The "Arrow" operator is a nice notational shorthand. If we allocate a structure on the heap, and want to assign one of the members of that structure, we write something like this: `(*p1).x = 99.9;` . An equivalent statement that results in the same outcome but looks a little neater: `p1->x = 99.9;`. These two statements are perfectly interchangeable. If you really prefer one style, use that style. If the Arrow makes no sense to you[1] you don't have to use it.

Arrays. Arrays can be either stack or heap allocated, as you would expect. A stack allocated array is fairly simple, where you use the square brackets to specify the size that you want, as below:

```
int array[10];
```

Now the stack has an array named `array` of capacity 10 integers, all of which, remember, contains garbage because it has not been initialized. For a small array, or one that doesn't need to last very long, this suffices. But it is much more likely that we need to allocate arrays on the heap:

```
int* array = malloc( 10 * sizeof( int ) );
```

This looks weird, but this is how we do it we asked for enough memory to hold 10 integers and this will be our array. But yes, an integer pointer could point to only one integer or to an array of integers. When deallocating this memory, the call is still `free( array )` (unlike C++ there's no difference when deallocating an array).

Either way, to assign elements of the array we use the square brackets operators, like `array[0] = 5;`. There's no difference in whether or not it is stack or heap allocated when we do this.

As said, the array needs to be initialized, regardless of whether it is stack or heap allocated. The simple way would be a for loop:

[1] Then you have failed this city...

```
int * array = malloc( 10 * sizeof( int ) );
for ( int i = 0; i < 10; i++ ) {
  a[i] = 0;
}
```

This is nice but as you might realize this could take a long time for a sufficiently large array. There is a nice shortcut you can use (if you include the `string.h` header). It is called `memset`. It takes three arguments: a pointer to the memory you want to set, the value you want to initialize it to (the function does unsigned char conversion here, so pretty much this is only good for initializing to 0), and then how many bytes should be set:

```
int * array = malloc( 10 * sizeof( int ) );
memset( array, 0, 10 * sizeof( int ) );
```

There are no automatic bounds checking for arrays, and C will gladly let you try to read the 12th element of a capacity-10 array. You will either read garbage or perhaps crash your program. When iterating over an array, you will need to know in advance how long the array is so you know when to terminate the loop. For that reason, when an array is passed as an argument to a function, another variable is used to specify the length.

Strings. As far as C is concerned, a string is just a character array (and is generally just `char*`. In C, a string is of arbitrary length and terminates with a byte of zero (or if you prefer, `NULL`). This is distinct from the character "0" which has a nonzero value. If you put a string literal in some code like `"Hello World"` then the compiler will graciously add the null-terminator for you. If you are composing a string, however, then you need to remember to add it (and the char for it is `'\0'`). And either way you should also remember to make room for the null terminator when allocating memory or calculating the length of a string.

Function Convention. Because there are no member functions for data objects the convention is generally that the first parameter to a function is modified. Thus, an initialization routine will take several arguments: the first being whatever it is we wish to initialize, then further arguments that are used to initialize this structure. Or, if a function modifies a struct, it is typically the first one that is changed:

```
int initialize_header( struct header * h, int v1, double v2, struct blah * v3 );
void addToPoint( point p1, point p2 );
```

This is not a firm rule and there are examples that contradict this. But it is convention and will be seen frequently.

In this same vein, many functions have integer return types. Because the first parameter structure is being modified, the return value is an integer that is the "result code". If everything went well this is usually zero. If there was a problem, then it's a nonzero number and the number can help you figure out what's wrong. For example, a return value of 36 might be something you can look up in the documentation and find out that the value used for the third argument is too large.

Also noteworthy: there is a header `errno.h` that defines a variable `errno`. Many functions and library calls set the value of this variable if something goes wrong during their execution. Knowing what precisely went wrong can be very valuable when debugging your program.

If you want to read `errno` (or print it) then you include the header and declare `extern int errno;` in your program. Then, when a function call that uses it has a problem, you can use `errno` by, for example, printing an error message. The header file contains definitions of numerous standard error values such as `EIO` for "Error in Input/Output". Check the documentation to get the full picture – there are a lot of them. But learning that your network call is code 111, `ECONNREFUSED` (connection refused) is a lot more helpful than just getting back `-1` as a return code.

Printing. Writing to the console is done using the `printf()` function. This function formats a string and writes it to the console. In a simple example, the Hello World program would contain the following print statement:

```
printf("Hello␣World\n");
```

Note that the `"\n"` in there is a newline character, which should probably not be forgotten if you want your printout to look right.

In addition to this, `printf()` does *formatted* printing. When we have values that need to appear in the output, we use a *format specifier*. The format specifier tells the printf how to format the value. The value is then given as an argument after the string. Order matters: the first format specifier encountered will match to the first argument after the string.

```
int x = -25;
...
printf("The␣value␣of␣x␣is␣%d\n", x);
```

Results in printing to the console `The value of x is -25` (and a newline character after it). In this example the format specifier `%d` is used for a signed integer (or `%i` would do the same). There are different format specifiers for different types: unsigned integers, floating point numbers, scientific notation, even a string. There are many options but we do not wish to belabour the point by putting them all here. But be sure you chose the right one: if you have a `double` and you try to print it with an `int` format specifier, it will look like the data contains garbage (yes, I have helped someone debug this).

Or another example:

```
point p1 = malloc( sizeof( point ) );
p1->x = 9.5;
p1->y = -5.4;
p1->z = 0.1;
printf("(%f,␣%f,␣%f)\n", p1->x, p1->y, p1->z);
```

Results in printing `(9.5, -5.4, 0.1)`. As you see, the `printf()` routine takes an arbitrary number of parameters. If the string contains 5 format specifiers, then the string is followed by five arguments. If the string contains 10 format specifiers, then there are 10 arguments after the string. If you have 3 format specifiers but only 2 arguments after the string, `printf()` will take whatever value is next on the stack which may be garbage.

Define Directives. It is possible to use a precompiler directive for fixed values in your program. This is the `#define` directive. Placing them at the top of the program is common. They are then used throughout the program to avoid "magic numbers" (i.e., numeric values that have no obvious meaning). Creation is like this:

```
#define BUFFER_SIZE 1000
```

And then later in the program it will be used:

```
char* buffer = malloc( BUFFER_SIZE );
```

When the program is compiled, the compiler replaces `BUFFER_SIZE` everywhere with its definition. If used in many places, changing the definition in the central location saves you from having to change it everywhere.

There is a small potential pitfall in this though; the compiler is VERY literal about how it makes the replacement. So if the directive is `#define VALUE 2 * 25 + 4`, then you will get the wrong answer if that's later used in something like `int size = VALUE * 2;`, because the compiler will make it `int size = 2 * 25 + 4 * 2;` for a total size of 58 rather than 108. This can be avoided by putting parenthesis `#define VALUE (2 * 25 + 4)`. Because BEDMAS.

Main and Arguments. A program begins at the start of `main` and the signature is as follows:

```
int main( int argc, char** argv ) {
  ...
  return 0;
}
```

The variable `argc` returns the count of the number of arguments to the program. The variable `argv` has two stars... because it is a pointer to an array of `char*` (if it helps you, imagine `char**` as `(char*)*`). This second variable is a pointer to an array of the actual text arguments. Let's imagine we invoke a program like this:

```
jz@Tyr:~$ ./a.out Hello 17
```

In this case, the value of `argc` is 3. The first argument is always the name of the executable, and thus `argv[0]` is `"./a.out"` (as a "string"). Then `argv[1]` is `"Hello"`, and `argv[2]` is `"17"`. The use of quotation marks is not a coincidence or oversight: it is a string and not the number 17. If we actually want to convert it, we need to parse it to an integer with a function like `atoi()`, such as `int value = atoi( argv[2] );`.

It is convention that the `main` method returns 0 if everything went well in the program and a different value if something went wrong. So if the wrong number of parameters is provided, you might end the program with `return -1;`.

Void Pointers. A construct we will use frequently is `void *`. It's not a pointer to nothing; it is a pointer with no type. This is the way of saying there is a pointer with no specific type. We've already seen an example of how to use it. The function `memset` takes as its first argument a void pointer. That is to say, this function does not care what the type of the pointer that it is provided is. So we can initialize an array of `int, char, double, point_t`, anything we like. This is but one example where untyped pointers will be used; we will see many more.

References

[Bra09] Clay Brashears. *The Art of Concurrency*. O'Reilly Media, 2009.

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.