CIS 343 - Structure of Programming Languages

Presented by: Nathan Bowman

Slides from: Ira Woodring

The C Programming Language

History

You can read this history, from the author!
https://www.bell-labs.com/usr/dmr/www/chist.html

History

History

Contrary to what we think now, it was not written for portability!

Interest in portability came later.

History

- Popularized syntax we use still today, such as brackets for array accesses (which was uncommon at the time!).
- Introduced the operators ++ and - - (invented by Ken Thompson, also at Bell Labs).

History

- It is an imperative procedural language
  - Imperative means it gives commands to change state
  - Procedural means it provides the ability to structure code and call procedures

History

- By around 1973 the basics were complete enough to rewrite the Unix kernel in the language.
- Toward the end of the 70's work began on making it portable.
    - This means we can take the same code and compile it on different architectures to the same effect.

History

- After it became portable the popularity soared (mostly due to the popularity of Unix).
- During the 80's compilers were written for nearly every architecture and OS.
- In 1983 work began on standardizing the language.

Overview

C is a high-level language that provides very low level capabilities.

Data types are very closely related to underlying hardware.

Many higher-level languages don't provide the capabilities that C has for interacting so closely with hardware. This makes C a better choice for systems programming and speed-intensive tasks.

Compilation

Every computer processor has an instruction set. This set of instructions are the ONLY things a processor knows how to do. Different brands and types of processors have different instruction sets.

C requires the use of a compiler to create the libraries and executables needed to run a program.

The compiler translates higher-level C code into machine code that a processor can understand.

Here are two sample instruction sets, should you be interested:

Intel Sample

ARM Sample

Compilation

In this class we will use either the Clang compiler or the GCC compiler. Both are on EOS, and freely available should you wish to install them on your own machines.

Compilation

A program written in C is some executable code (usually) linked to other libraries.

These libraries are often pieces of code written by a community of advanced programmers, and provide facilities such as I/O, memory management, etc.

Likely, these libraries have been in development for many years, are very robust, efficient, and secure code. We should use them.

But, we can write libraries as well (and should!).

C gives us facilities to separate out our code into interface and implementation files.

Compilation

Interface files describe data types, functions, etc. that may be used in a library. The do not usually provide the implementation. Interface files are *.h files.

Implementation files are the actual code files that provide the instructions needed to complete some task. They are provided in *.c files.

Compilation

The C compiler compiles every file separately. So if we write a program that makes use of a library, we need to tell C that we are using the library correctly. For instance:

Say I have a file called `my_prog.c`. I wish to use a function called `handyFunction(42);`.

While compiling `my_prog.c` the compiler needs to know if I am using `handyFunction()` correctly. Am I providing the correct type and number of parameters? Does it return the expected type?

The compiler is given this information from the *.h file.

We will say my library is called `library`. So I should have `library.h` and `library.c` files. The *.h file, the interface, might look like this:

```
int handyFunction(int num);
```

This tells the compiler that there is a definition for `handyFunction` that takes an `int` and returns an `int`.

At the top of my `my_prog.c` file, I can tell the compiler about the library file with the command:

```
#include "library.h"
```

Note: when importing libraries we use "" around personal libraries. If we were importing a system library we would use <> such as:

```
#include <library.h>
```

So... we know that each file is compiled separately.

We provide the interfaces via the `#include` statements.

This causes the code to be pulled into the project.

Can anyone see a potential problem?

What happens if we `#include` a library, and then a library `#includes` the same library?

The code would be sourced into the project more than once.

Bloats the code. Not good.

Enter **include guards**.

Include guards allow us to define some name in memory. If it has already been defined, we don't redefine it. So we surround our *.h file code with something like this:

```
#ifndef     __H_OUR_NAME__
#define     __H_OUR_NAME__

... *interface code* ...

#endif
```

```c
/*
 * my_prog.c
 * Ira Woodring
 *
 * A trivial example of how to write a basic C program,
 * with the use of external libraries.
 *
 */

// The # symbol is a compiler directive.  These are not
// lines of code really, but instructions for the compiler.
// This one says we are using the standard I/O library, so
// the compiler knows we are using functions like printf
// correctly.
// Notice we are using <> around the library name as this
// is a system library.
#include <stdio.h>

// This is a library we created, of code we may wish to use
// in multiple projects.
#include "library.h"

// This is the program's entry point
int main(int argc, char** argv){
    printf("Howdy world!\n");
    handyFunction(42);
}
~
~
~
~
~
~
~
~
```

```
// When compiling an executable, the compiler needs
// will pull in the *.h files and underlying code
// everytime it is asked to with an #include.
// But, we may wish to use the library in multiple
// different files.  We don't want the code to be
// included in the final executable multiple times,
// as that would be redundant and lead to bloat.
// Solution?  Include guards!

// Here we are saying:
// If the name __H_MY_LIBRARY__ has not been defined
// already, then define it by loading in this code.
// Otherwise, do nothing.
#ifndef         __H_MY_LIBRARY__
#define         __H_MY_LIBRARY__

// As this is an *.h file, this is an interface.  No
// code is provided for how the function should work.
void handyFunction(int num);

#endif
~
~
~
~
~
~
~
~
```

```c
// In the implementation file, we must #include the
// interface file as well.
#include "library.h"

// Even though we source stdio.h into our prog.c,
// we still MUST include it in this file as well.
// Remember!  Each file is compiled separately.
#include <stdio.h>

void handyFunction(int num){
    for(int i=0; i<num; i++){
        printf("=");
    }
    printf("\n\n");
}
~
~
~
~
~
~
~
~
```

We compile this code with the command

```
clang my_prog.c library.c
```

or

```
gcc my_prog.c library.c
```

This will output a file called `a.out`. We run that with the command

```
./a.out
```

By the way, `a.out` stands for "assembly out". It is a holdover from the early days at Bell Labs. It was kept through the B, BCPL, and early C phases because folks were used to it.

You can provide an optional executable name with the `-o FILENAME` flag such as:

```
clang my_prog.c library.c -o my_executable
```

Same syntax applies to GCC.

Compilation

Notice that on the command-line we pass the names of ALL the *.c files to the compiler. We **DO NOT** pass the *.h files!

However, there is no need to pass the name of the system libraries. The <> around their names indicate to the compiler to search through the system library path.

It turns out that most systems pre-compile common libraries. This gives us multiple benefits:

Compilation

- We save disk space by not having to include these libraries in our own projects.
- We save compilation time by not having to re-compile these prebuilt libraries.

Compilation

Which brings us to a very important point:

The compiler doesn't actually create executable files. It creates object files. The **linker** links together the files into an executable.

Compilation

The linker is often automatically called by the compiler (the commands provided earlier will automatically cause this to happen).

It turns out though, there are two types of linking:

Static Linking and **Dynamic Linking**.

Compilation

Static linking pulls in all of the code needed for a program to run. This results in a larger executable, but guarantees that the program can run on the target system.

Compilation

Dynamic linking uses **method stubs** when linking to a system library. These executables are not as portable. The target system must provide the required libraries the stubs point to or the program can't run. This is the default linking method.

Compilation

When should I use each method?

Most of the time you will just want to link dynamically. This keeps the executable small and relies upon the system libraries. Since system libraries are patched often to fix bugs/security holes, this is a good practice.

Compilation

If you happen to be coding and need very specific versions of a library for your code to work correctly, you may wish to compile statically. This will pull in all of the code and libraries needed for the executable to run on the target system.

Program Layout

Every C program has an entry point called `main`. This function should look like this:

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

Note that you may see programmers write similar but different function signatures for `main`, but those are not best practice.

Program Layout

Let's dissect the signature a bit.

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

The first `int` is the return type. The `int argc` is a parameter to the function `main` of type `int`, and the `char** argv` is a parameter to the function `main` of type pointer to a pointer to a `char`.

Program Layout

The return type of `int` means that a C program will return a numeric value. This value corresponds to an error code. This allows us to tell the operating system if the program exited cleanly, or if it exited due to some error state.

This allows us to automate program runs!

```
#ifndef _ASM_GENERIC_ERRNO_BASE_H
#define _ASM_GENERIC_ERRNO_BASE_H

#define EPERM        1  /* Operation not permitted */
#define ENOENT       2  /* No such file or directory */
#define ESRCH        3  /* No such process */
#define EINTR        4  /* Interrupted system call */
#define EIO      5  /* I/O error */
#define ENXIO        6  /* No such device or address */
#define E2BIG        7  /* Argument list too long */
#define ENOEXEC      8  /* Exec format error */
#define EBADF        9  /* Bad file number */
#define ECHILD      10  /* No child processes */
#define EAGAIN      11  /* Try again */
#define ENOMEM      12  /* Out of memory */
#define EACCES      13  /* Permission denied */
#define EFAULT      14  /* Bad address */
#define ENOTBLK     15  /* Block device required */
#define EBUSY       16  /* Device or resource busy */
#define EEXIST      17  /* File exists */
#define EXDEV       18  /* Cross-device link */
#define ENODEV      19  /* No such device */
#define ENOTDIR     20  /* Not a directory */
#define EISDIR      21  /* Is a directory */
#define EINVAL      22  /* Invalid argument */
#define ENFILE      23  /* File table overflow */
#define EMFILE      24  /* Too many open files */
#define ENOTTY      25  /* Not a typewriter */
#define ETXTBSY     26  /* Text file busy */
#define EFBIG       27  /* File too large */
#define ENOSPC      28  /* No space left on device */
#define ESPIPE      29  /* Illegal seek */
#define EROFS       30  /* Read-only file system */
#define EMLINK      31  /* Too many links */
#define EPIPE       32  /* Broken pipe */
```

```
#define EDOM         33  /* Math argument out of domain of func */
#define ERANGE       34  /* Math result not representable */

#endif
~
~
```

```
#ifndef _ASM_GENERIC_ERRNO_H
#define _ASM_GENERIC_ERRNO_H

#include <asm-generic/errno-base.h>

#define EDEADLK     35  /* Resource deadlock would occur */
#define ENAMETOOLONG    36  /* File name too long */
#define ENOLCK      37  /* No record locks available */
#define ENOSYS      38  /* Function not implemented */
#define ENOTEMPTY   39  /* Directory not empty */
#define ELOOP       40  /* Too many symbolic links encountered */
#define EWOULDBLOCK EAGAIN  /* Operation would block */
#define ENOMSG      42  /* No message of desired type */
#define EIDRM       43  /* Identifier removed */
#define ECHRNG      44  /* Channel number out of range */
#define EL2NSYNC    45  /* Level 2 not synchronized */
#define EL3HLT      46  /* Level 3 halted */
#define EL3RST      47  /* Level 3 reset */
#define ELNRNG      48  /* Link number out of range */
#define EUNATCH     49  /* Protocol driver not attached */
#define ENOCSI      50  /* No CSI structure available */
#define EL2HLT      51  /* Level 2 halted */
#define EBADE       52  /* Invalid exchange */
#define EBADR       53  /* Invalid request descriptor */
#define EXFULL      54  /* Exchange full */
#define ENOANO      55  /* No anode */
#define EBADRQC     56  /* Invalid request code */
#define EBADSLT     57  /* Invalid slot */

#define EDEADLOCK   EDEADLK

#define EBFONT      59  /* Bad font file format */
#define ENOSTR      60  /* Device not a stream */
#define ENODATA     61  /* No data available */
#define ETIME       62  /* Timer expired */
```

```
#define ENOSR       63  /* Out of streams resources */
#define ENONET      64  /* Machine is not on the network */
#define ENOPKG      65  /* Package not installed */
#define EREMOTE     66  /* Object is remote */
#define ENOLINK     67  /* Link has been severed */
#define EADV        68  /* Advertise error */
#define ESRMNT      69  /* Srmount error */
#define ECOMM       70  /* Communication error on send */
#define EPROTO      71  /* Protocol error */
#define EMULTIHOP   72  /* Multihop attempted */
#define EDOTDOT     73  /* RFS specific error */
#define EBADMSG     74  /* Not a data message */
#define EOVERFLOW   75  /* Value too large for defined data type */
#define ENOTUNIQ    76  /* Name not unique on network */
#define EBADFD      77  /* File descriptor in bad state */
#define EREMCHG     78  /* Remote address changed */
#define ELIBACC     79  /* Can not access a needed shared library */
#define ELIBBAD     80  /* Accessing a corrupted shared library */
```

Program Layout

The two parameters sent to the `main` function are an `int` telling the program how many command-line arguments were passed to the program, and a pointer to a pointer to a `char`. As we will see shortly, this is how C creates an array of strings. These strings are the command-line arguments.

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

int main(int argc, char** argv){
    printf("Howdy all!\n\n");
    printf("This program was provided with %d command-line arguments.\n\n", argc);
    printf("These arguments are:\n====================\n\n");
    for(int i=0; i<argc; i++){
        printf("\t%s\n", argv[i]);
    }

    return 0;
}
```

Program Layout

If we compile and run this program with different input, we get the following outputs:

```
Iras-MacBook-Pro:samples woodriir$ ./a.out
Howdy all!

This program was provided with 1 command-line arguments.

These arguments are:
====================

    ./a.out
```

```
Iras-MacBook-Pro:samples woodriir$ ./a.out Hi there everybody
Howdy all!

This program was provided with 4 command-line arguments.

These arguments are:
====================

    ./a.out
    Hi
    there
    everybody
```

```
Iras-MacBook-Pro:samples woodriir$ ./a.out "Hi there everybody
Howdy all!

This program was provided with 2 command-line arguments.

These arguments are:
====================

    ./a.out
    Hi there everybody
```

Program Layout

So, a few notes. There is always 1 command-line argument provided. It is the executable's name.

If we wish to group arguments together we need to place them inside of quotes.

Program Layout

C programs are collections of statements that change state of some memory.

Commonly used collections of statements may be abstracted out and placed into units called functions.

Program Layout

A function must be declared before it is used.

We can do that multiple ways:

- By placing the function's declaration before the code that calls that function in the same file
- Or by placing the declaration in an interface file (*.h) and `#include`ing it.

Program Layout

We will often talk about a function's *signature*. This is the return type, name, and parameters a function takes.

```
return_type function_name(parameter_type parameter_name, ...re
```

For example:

```
double calc_tax(float amount, double tax_rate);
```

Program Layout

Functions may take different parameters. This means they have different signatures:

```
double calc_tax(float amount, double tax_rate);
```

is a different signature from

```
double calc_tax(float amount, float tax_rate);
```

Data Types

C provides a few basic data types.

These can be separated into categories:

Void

Integer Types

Floating Point Types

Data Types

When no type is or can be supplied, the type is

```
void
```

Data Types - Integer Types

`char`, `short`, `int`, `long` types hold integer (non-floating point) values. These types may also be `unsigned`, which increases the maximum value they can hold, but removes the ability to store a negative value.

Note that there are quite a few more types than listed here; types such as `long long`, `long long int`, `signed` and `unsigned long long ints`, etc.

Data Types - Integer Types

Wikipedia has a nicely tabled list, including format specifiers for how to print these values:

https://en.wikipedia.org/wiki/C_data_types

Data Types - Integer Types

The C standard does not dictate how many bytes each data type must occupy; it DOES however provide minimum numbers of bytes and relationships between types.

Data Types - Floating Point Types

C provides for `float`, `double`, and `long double` data types. These data types are stored via the IEEE 754 standard (we will talk about this later; hardware class will study it in depth.)

Data Types - Record Types

C provides two mechanisms for storing record types, the `struct` and the `union`.

Both a `struct` and a `union` are aggregate types, meaning they are types made up of other types. For instance, if you wanted to create a data type that could hold student information, you may define a `struct` as follows:

```
struct student {
  int g_number;
  float gpa;
  char class;
};
```

You would declare one as such:

```
struct student s;
```

Alternatively, you can define the struct as a type using the `typedef` keyword:

```
typedef struct student_type {
  int g_number;
  float gpa;
  char class;
} student;
```

And declare it:

```
student s;
```

Either way, you would then be able to use `s` by accessing its members:

```
s.g_number = 300746283;
s.gpa = 3.874;
s.class = 's';
```

Data Types - Record Types

`union` types are very similar to `struct` types, except `union`s only store one member at a time. These are usually only useful to embedded system programmers with lower memory requirements. This is because the `union` only allocates enough memory for the largest of its members.

For instance, if a union held a member for an `int` and a `float`, it would only occupy enough space for a float (the larger data type).

Data Types - Pointers

Pointers are data types that store a memory location. They are extremely useful but often confusing to new programmers in the language.

A pointer may be of type `void*` but will usually be typed the same as the type of data its memory points to.

For instance:

```
int x;
int* y;
```

These are two COMPLETELY DIFFERENT data types. The first declares an `int` variable. The second declares a variable that holds a memory address. It just so happens that the memory address this value points to is an `int`.

This allows us to do something like this:

```
int x = 42;
int* y = &x;
```

The & symbol here can be read as "address of". We are saying that y is a pointer variable that points to an `int`. We are setting the value of y to the memory address of where x is stored.

Additionally, we can access the data in the memory location that the pointer points to. We can do this by using the * (called the dereferencing) operator:

```
int x = 42;
int* y = &x;
*y = 45;          // If we printed x it would now print 45!
```

Data Types - Pointers

But... why do we need pointers?

Data Types - Pointers

C can only return a data type's value.

What we are more used to (coming from a Java background) is a pass-by-reference model (although Java isn't pass-by-reference).

Passing by reference means that we don't pass a data structure to or from a function - we pass a reference to that structure. This keeps things fast as the size of the data structure increases. But C doesn't have this ability.

Data Types - Pointers

C passes the value of the data type to and from functions. So, if we have the following code:

```
int doStuff(int x){ ... }
int main(int argc, char** argv){
  int x = 42;
  int y = doStuff(x);
}
```

x is not passed to the function. 42 is passed to the function.

Data Types - Pointers

What is actually happening is that a new area of memory is setup for the function to work in. In that new area of memory the function gets a copy of the data to work on. All work occurs in the function's memory space, on the function's variables. This means any change you make to x only changes the function's local copy.

Data Types - Pointers

It is sometimes easier to think of this by adding to the variable's name. We can think of the first `x` as `main's x` or `main.x` and the second `x` as `doStuff's x` or `doStuff.x`.

This helps us to remember they are two distinct variables.

Data Types - Pointers

Seriously though... this sounds like a stupid pain in the butt.

What's the point? (no pun intended...)

Data Types - Pointers

Imagine using a large data structure, perhaps an array with millions of elements. If C didn't have pointers, since it must pass by value it would have to produce a copy of the entire array for a function to work on it.

Pointers are much, much faster.

Data Types - Pointers

They also allow us to do some neat things like pass functions as parameters to other functions!

Data Types - Pointers

I have provided some tutorial code for help in understanding pointers. Clone the project here:

https://github.com/irawoodring/pointer_perils

Compile it, and step through it until you understand what is happening.

Your first C assignment will be made VASTLY easier if you understand this code.

Data Types - Arrays

Any data type can also be declared as an array:

```
int my_int_array[100];
student my_student_array[1000];
```

However! This is static creation of the array (compile-time). These array cannot be reassigned later in the program's runtime, so their size is fixed until recompiled.

Memory Management

We will see in our architecture lecture that there are two memory areas our programs make use of, the **stack** and the **heap**.

When C code is compiled, the compiler determines the amount of memory needed for each function call. This memory includes space for local variables and code.

Each time a function is called, the amount of memory needed for it to run (both code and variables) is allocated on the stack. Each function call is placed on top of the last, only being removed when a function returns.

Because of this, the size of variables is fixed.

But what if we need a data structure of changing size? Can't go on the stack...

Memory Management

We have to declare the space on the heap. We do that via the `malloc` or `calloc` (part of `stdlib.h`).

```
void *malloc(size_t size)
```

```
void *calloc(size_t nitems, size_t size)
```

Note that both of these functions return a pointer!

Memory Management

We use malloc as follows:

```
int* my_memory = malloc(50 * sizeof(int));
```

This line asks the system to reserve a segment of memory big enough to hold 50 integers.

You will see people who cast the malloc (I did in the past). You do not need to do this, and it is bad practice. Read more here:

https://stackoverflow.com/questions/605845/do-i-cast-the-result-of-malloc

Memory Management

We can now use this segment of memory just as we would a normal array!

```
my_memory[0] = 42;
my_memory[42] = 100;
```

This works because an array in C is really just a pointer to the first element in an array!

Memory Management

Note that this doesn't just work for basic types. Recall the `struct student` we had earlier?

```
typedef struct student_type {
  int student_number;
  float gpa;
  char class;
} student;
```

We could declare an array to hold 50 of these as such:

```
student* my_students = malloc(50 * sizeof(student));
```

Memory Management

The problem? Because we aren't dealing with data on the stack we have to manage the memory ourselves.

This means it is up to us to return this memory when we are finished with it!

Memory Management

Traditionally not doing so resulted in memory leaks. This can still happen on many operating systems, but most modern ones have protections against this.

Still... it is good practice to clean up after yourself. You may not always know on what type of system your code will run.

Memory Management

We return the memory to the heap with the `free()` function.

```
free(my_students);
free(my_memory);
```

Seriously, always `free` any memory you allocate.

This is by no means a comprehensive tutorial of C. It is quite a complex and powerful language. This is enough for you to begin learning, and to complete many basic and intermediate C tasks.