CIS 343 - Structure of Programming Languages

The C Programming Language

History

You can read this history, from the author!

https://www.bell-labs.com/usr/dmr/www/chist.html

History

- Was written in the early 70's by Dennis Ritchie at Bell Labs.
- While some people think Brian Kernighan helped write C, this is not the case. He merely helped write the book on it (he has clearly stated this himself!).
- Was written for Unix development (original Unix was written in Assembly)
- Was based on BCPL, a language with no types, and B. C added types and other features.

History

- Contrary to what some 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 - Instruction Sets

- 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.

Compilation - Instruction Sets

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

- Intel Sample
- ARM Sample

Compilation - Compilers

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.

> **Note**: I prefer clang as it provides better errors and warnings by default

Compilation - Libraries

- 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**!).

Compilation - File Types

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

- 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:

Compilation

- 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.
- The compiler **does NOT** need the implementation of `handyFunction()`.

Compilation

We will say my library is called `library`. So I should have

- `library.h` - the interface file. May include declarations data structures, typedefs, as well as function signatures and anything else that should be shared between multiple files.
- `library.c` - the implementation file; i.e. the function bodies and variables needed for them, etc.

Compilation

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`.

Compilation - Including Code

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>
```

Compilation

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?

Compilation - Include Guards

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

- The code would be sourced into the project more than once.
- Bloats the code. Not good.

Compilation - 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
```

Compilation - Include Guards

> **Note**: Some compilers support the `#pragma once` directive. This accomplishes the same thing as an include guard but is non-standard and best avoided.

Compilation

We can compile this code with the command (notice both *.c files!):

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

or

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

This will output a file called `a.out` (`a.exe` if you are on Windows). We run that with the command

```
./a.out
```

Compilation

- `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

On the command-line we could have called clang individually per file:

```
clang -c library.c
clang -c main.c
```

This would create two files called `library.o` and `main.o`. We could link these together with:

```
clang library.o main.o
```

- We DO NOT pass the *.h files - they are pulled-in via the #include statements.

Compilation

- Most systems pre-compile common libraries. This gives us multiple benefits:
  - Saves space.
  - Saves time.
  - Allows updates to be more easily applied.

Linking

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.

Linking

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

There are two types of linking:

- **Static Linking**
- **Dynamic Linking**.

Static Linking

- 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.

Dynamic Linking

- 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 to 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.

Linking

- 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!

---?code=./c-lectures/samples/errno-base.h

`errno-base.h`

---?code=./c-lectures/samples/errno.h

`errno.h`

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. Usually called `argc`.
  - 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. Usually called `argv` (v for vector).
- **NOTE:** the first argument, argv[0] is the program name!
  ---?code=./c-lectures/samples/args.c

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, ...repeat as needed...)
```

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);
```

- **NOTE:** This is called **overloading** (not overriding - C can't do that.)

Data Types

C provides a few basic categories of data types.

- **Void**
- **Integer Types**
- **Floating Point Types**
- **Aggregate Types**
- **Pointers**
- **Arrays**

Data Types - Void

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:** There are quite a few more types than listed here; types such as `long long`, `long long int`, `signed` and `unsigned long long int`s, 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

- C doesn't dictate the maximum number of bits in a byte (minimum is 8 bits on current standard though could have been less in past standards).
    - Allows compiler developers to choose fastest for underlying system.
    - POSIX dictates 8 bits in a byte though!
- It does however provide minimum and maximum values each type must hold.
    - i.e. `short` must be able to store **AT LEAST** -32,767 to 32,767.
    - Can hold a bigger range though, as long as it includes this range.
- Also provides fixed size integers such as `uint8`, etc.

Data Types - Fixed-Width Integer Types

- Guarantee the width (number of bits) for a data type.
- Are **OPTIONAL**. Compiler developers can choose to support them or not. Makes *less* portable code.
- Typically only used if we need to represent fixed-width data (perhaps for reading a file that has several 8-bit chunks, etc.).
- If you don't have a reason to use them, you probably don't need to.

Data Types - Floating Point Types

C provides for

- `float`
- `double`
- `long double`

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 - Aggregate Types

C provides two mechanisms for storing aggregate types, the

- `struct`
- `union`

Aggregate types 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:

Data Types - Structs

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

You would declare one as such:

```
struct student s;
```

- **NOTE:** This is NOT an object! (Why not?)

Data Types - Structs

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;
```

Data Types - Structs

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';
```

- **NOTE:** This is creating one on the stack (statically at compile time).

Data Types - Structs

- Or, we could create one dynamically at run-time on the heap:

```
// Here, s is NOT a student.  It is a pointer to a student.
// The student variable has no name but exists at the address
// returned by malloc.

student * s = malloc(sizeof(student));
```

- Then, we can access members using pointer notation:

```
s->g_number = 300746283;
s->gpa = 3.874;
s->class = 's';
```

Data Types - Structs

Pointer notation is just shorthand (syntactic sugar) for:

```
(*s).g_number = 300746283;
(*s).gpa = 3.874;
(*s).class = 's';
```

- **NOTE:** The parentheses cause the pointer to be dereferenced *before* finding the member. (What would it mean otherwise?)

Data Types - Unions

- `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.
- If a union held a member for an `int` and a `double`, it would only occupy enough space for a `double` (the larger data type).
- Unions harm readability and writability of code. Don't use them unless you must.

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.
- This way the compiler can help us catch type errors.

Data Types - Pointers

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 of an `int`.

Data Types - Pointers

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.

Data Types - Pointers

We can access the data in the memory location that the pointer points to. We 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** pass data by value (also called pass-by-copy).
    - Data must be passed between functions through parameters and returns.
- Many of us are more used to a pass-by-reference model.
- Passing by reference means that we don't pass a data structure to or from a function - we pass a reference to that data. This keeps things fast as the size of the data structures increase. 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.

- **NOTE:** This is the ONLY way C can pass data. It does not matter what type of data we are passing

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 of `x`.

Data Types - Pointers

It is sometimes easier to think of this by mentally 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... maybe...)

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!
- Easy to do since code and data are housed in the same memory space! (Remember the Princeton Model?).

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 arrays cannot be reassigned later in the program's runtime, so their size is fixed until recompiled.

Memory Management

Recall from 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. These go on the stack.
- Data allocated at run-time goes on the heap.

Memory Management

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:** Both of these functions return a pointer to the memory allocated (if the call was successful).

Memory Management

We use malloc as follows:

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

or

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

- 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

Notice 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.