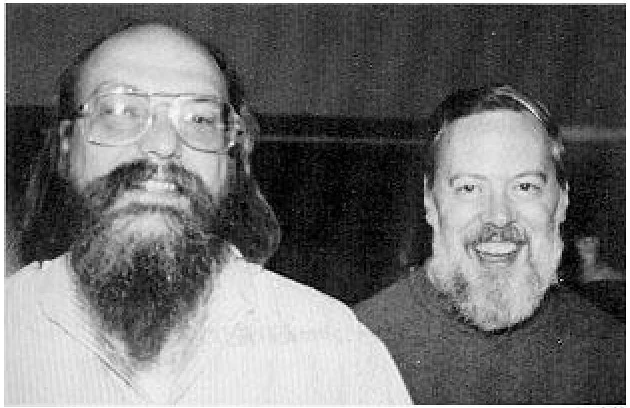
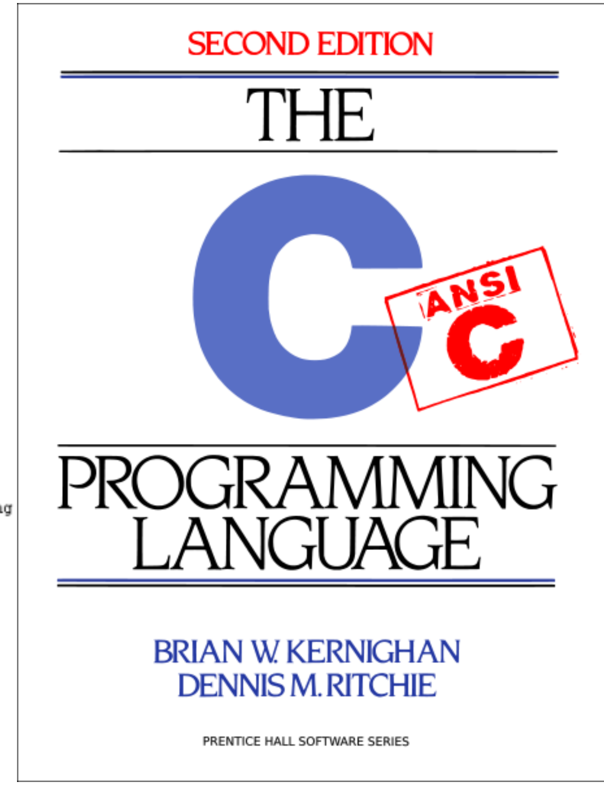
An Intro to C



```
{
        WARN_ON_ONCE((cpu_has_apic && !disable_apic));
        return 0;
}

/*
 * right after this call apic->write/read doesn't do anything
 * note that there is no restore operation it works one way
 */
void apic_disable(void)
{
        apic->read = native_apic_read_dummy;
        apic->write = native_apic_write_dummy;
}

void native_apic_wait_icr_idle(void)
{
        while (apic_read(APIC_ICR) & APIC_ICR_BUSY)
                cpu_relax();
}

u32 native_safe_apic_wait_icr_idle(void)
{
        u32 send_status;
        int timeout;

        timeout = 0;
        do {
                send_status = apic_read(APIC_ICR) & APIC_ICR_BUSY;
                if (!send_status)
                        break;
                udelay(100);
        } while (timeout++ < 1000);

        return send_status;
}
```



“C should be every programmer’s first crush”

Links

Source Code: https://github.com/gnu-user/intro-c

Basics

- Comments
  - // for single line comments
  - /* *…* */ for multiline comments
- Variables
  - letters, digits, underscores
  - naming convention uses underscores (my_variable)
  - C is a Spartan language, and so should your naming be
- Sections of code (code blocks) are enclosed using curly braces { … }

Basics

- C has many operators, many of which you are familiar with from other languages
  - assignment: =
  - arithmetic: +, -, *, /, %
  - augmented assignment: +=, -=, *=, /=, %= …
  - boolean logic: !, &&, ||
  - equality testing: ==, !=
  - increment and decrement: ++, --
  - **member selection: ., ->**
  - **object size: sizeof**
  - order relations: <, <=, >, >=
  - **pointer, reference, and dereference: &, *, [ ]**

Basics

- Data types, C has the following base types
  - char
  - int
  - long
  - float
  - double
  - void
- Each data type can have the following modifiers
  - signed, unsigned (+/- or only + values)
  - const (a constant value such as pi in your code)
  - static (makes the variable stay in memory and accessible without regard to scope)

Basics - Arrays & Pointers

- A Collection of data, syntax similar to other languages
- Arrays need to be null terminated (strings) or have the length stored somewhere
    - If you create a character array (strings) they are **automatically null terminated**

```
int ar[10]; // This has no data in it yet
int point[6]={1,2,3,4, 5,0}; // should know the length!
char str[] = “hello” // auto null-terminated
```

- Pointers are conceptually like a street signing “pointing” you to where something in memory is located
- We will discuss pointers later when we talk about dynamic memory

Basics - Other Data Types

- Structs, allow you to create your own data types
  - Often used with **typedefs**, which allow you to define a “type” as a different name, a shortcut for structs

```
typedef struct {
    char name;
    int age;
} my_struct;
```

- Unions, like a struct, can only be one data type of the struct at a time…
- Enums, enumeration of data, used a lot to simplify code

```
typedef enum { red, green, blue } colours;
```

Basics - Scope

- Scope, a very important concept, easiest to remember that variables defined inside **{ … }** cannot be accessed outside of it
- Variables and methods declared in other header files can be used when you include the header file
- Generally you have access to everything included from the header file (functions, constants, structs) anywhere in your code

Basics - Control Flow

- If, else-if, else statements

```
if (5 < 6) {
    printf(“Five is less than six!”);
} else {
    printf(“What?!”);
}
```

- You have the standard for, while, and do while loop like in other languages

```
for (int i = 0; i < 10; ++i) {
    printf(“Hello World!\n”);
}
```

Basics - Control Flow

- Switch-case statement, much cleaner than if-elseif-elseif…-else statements, especially when dealing with numeric/enumerated data

```
switch (grade) {
 case 1:
    printf("A\n");
    break;
 case 2:
    printf("B\n");
    break;
...
 default:
    break;
}
```

Basics - Functions

- Functions are the building blocks of C, they allow you to organize code into **conceptual blocks of execution**
- Functions without arguments in C **must have void** as the argument, this is different from other languages
- Functions are often broken into two steps the ***declaration*** which is just the function name and parameters in a header file and the ***definition*** which is the actual code (body) of the function
- Functions in C are pretty standard, except for a few quirks you must have **void** as an argument if there are none, always specify the types and variables of functions (even though the code compiles if you don't)

Basics - Functions

```
double square(double value); // The declaration

double square(double value) { // The definition
    return value * value;
}

// This takes no arguments and returns nothing
void say_hello(void);

void say_hello(void) {
    printf("Hello\n");
}
```

Basics - Other

- You can utilize the pre-processor of the compiler to create macros in C, these should be used rather than putting in “hard-coded” values in your programs
- The ***#define*** keyword is used to define macros which the compiler will substitute throughout your code

```
#define MY_CONSTANT 123456
```

- Typecasts are important for converting between data types in your program, however you must be careful that you are typecasting between compatible types using **(type_name) expression**

```
double value = (double) 5 / 3;
```

I/O

- Performing input and output operations are very useful in C for reading and writing data as well as printing out content to the terminal.
- The most common methods for performing I/O in C are the following
  - printf
  - puts
  - fopen
  - fclose
  - scanf
  - fprintf

Printing to Console

- The **printf** function takes a formatted string containing the content to print and takes the content to print as arguments, for a newline add **\n** to the end.
- The most common placeholders for printing variables are
  - **%d** -- for integers
  - **%f** -- for floating point values
  - **%s** -- for strings

```
   printf(“Hi %s you’re %d and %f feet\n”, “sue”, 10,
4.2);
```

- **puts** is similar to printf but only prints text and adds a newline to the end automatically.

Opening and Closing Files

- The **fopen** function is used to open a file given the path provided, the second argument it takes is the mode the most common are the following:
  - **“r”** -- read the file
  - **“w”** -- write to the file
  - **“r+”** -- read and write to the file
  - **“a”** -- append to the file
- The fopen function returns a pointer to the file (more on this later), **fclose** takes the pointer and closes the file.

```
File *fp = fopen("outfile.txt", "w");
fclose(fp);
```

Reading Input

- The **scanf** function is most commonly used to read input from the terminal provided by users, the scanf function takes a formatted string of the expected input, and the address in memory (reference).
- The following example is for a prompt asking the user to enter their age and year of birth

```
int age, year;
printf("Enter your age and year of birth:");
scanf("%d %d", &age, &year);
```

Reading From a File

- The **fscanf** function is similar to scanf, except that it takes the pointer to the file as the first argument, followed by the formatting, and the references to the variables to store the results.

```
int age, year;
File *fp = fopen("age_year.txt", "r");
fscanf(fp, "%d %d\n", &age, &year);
fclose(fp);
```

Writing to a File

- The **fprintf** function is similar to printf, except that it takes the pointer to the file as the first argument, followed by the formatting, and the variables for the formatted string.

```
int age = 5, year = 2010;
File *fp = fopen("age_year.txt", "w");
fprintf(fp, "%d %d\n", age, year);
fclose(fp);
```

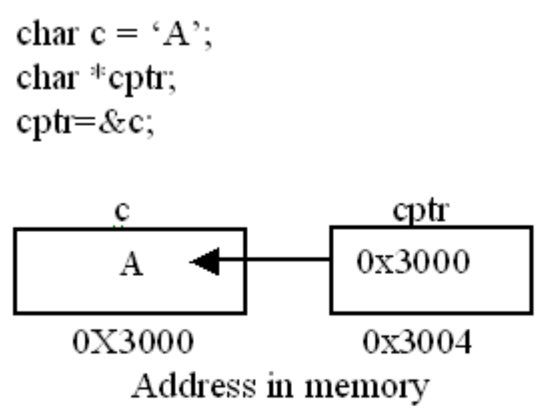
```c
#include <stdio.h>
#define AGE_INCREMENT 5
double square(int val);
typedef struct {
    char* name;
    int age;
} person;
typedef enum { red, blue, green } colour;

int main(void){
    person student;
    student.name = "Bob";
    student.age = 20;
    for (int i = 0; i < 10; ++i) {
        switch (student.age) {
        case 20:
            printf("young\n");
            break;
        case 30:
            printf("forever 21\n");
            printf("Their age doubled: %f\n", square(student.age));
            break;
        default:
            break;
        }
        student.age += AGE_INCREMENT;
    } return 0;
}
double square(int val) {
    return (double) val * val;
}
```

Memory Management

- Anytime you need to create something that you don't know ahead of time (at runtime) you generally need to use dynamic memory allocation
- Data that is declared in your code (e.g. variables, constants, strings) are placed on the stack, whereas data that is allocated when the code is executed is on the heap
- The main functions used for dynamically memory allocation are **malloc, calloc, realloc, free**
- Pointers are used as "sign posts" that tell you where data is located in memory

Pointers

- Pointers are variables that point to data in memory
- Pointers store the memory location, they are dereferenced to access the data in memory
- Pointers can be updated to “point” to a new location in memory just as any other variable can be changed
- Asterisk “*” used to create a pointer, my personal convention is to group it with the variable (e.g. *int *var*)



Pointers

- Remember scope? It's important, when you have a pointer to something in memory you have to keep track of it and free it when you're done using it
- Just because the pointer variable doesn't exist anymore it **doesn't mean the memory it's pointing to is now free**
- Arrays are actually pointers! The language just has a nice syntactic sugar for accessing them (e.g. my_array [4] is getting the 4th value from the base memory address in the pointer my_array)
- Always null initialize your pointers, this makes it easy to check if a pointer is actually pointing to valid data!

Dynamic Memory

- When you dynamically allocate memory you need to specify how many bytes of memory you need using **malloc** or **calloc**
- **malloc** is just “memory allocate” and allocates any memory (often full of garbage data that was already sitting in memory)
- **calloc** is the same except it fills the memory with null
- **realloc** is useful if you need to get more memory for the memory you allocated originally using malloc/calloc
- Memory allocated is returned as a void pointer, you must ***typecast*** the pointer returned to the correct type

Dynamic Memory

- Often you may not know the size in bytes of something such as a struct, this is where **sizeof** comes in handy
- sizeof will tell you the size of the data in bytes, if you need to make an array, just multiply sizeof by the size of your array
- When you are done using the memory you must free it using **free**, this is the most difficult part of dynamic memory and leads to memory leaks if you don’t free memory that you are no longer using
- Whenever you are dealing with strings **ALWAYS use calloc**, as it will fill the memory will null characters. Otherwise you will have a lot of headaches...

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFFER 128

void mem_leak(void);

int main(void) {
    char *name = NULL;
    int *age = NULL;
    name = (char *) calloc(BUFFER, sizeof(char)); // Allocate array using
calloc
    age = (int *) malloc(sizeof(float)); // Allocate a float using malloc
    mem_leak(); // Creates memory leak allocated memory not freed
    printf("Enter your name: ");
    scanf("%s", name);
    printf("Enter your age: ");
    scanf("%d", age);
    // Arrays are pointers, print each character of the name
    for (int i = 0; i < strlen(name); ++i) { // strlen fail if didn't use
calloc!
        printf("%c\n", name[i]);
    }
    // Don't forget to free the memory. Can't free munch don't have pointer!
    free(name), free(age);
    return 0;
}
// This function creates memory leaks, it should return the munch pointer!
void mem_leak(void) {
    char* munch = NULL;
    munch = (char*) calloc(BUFFER * 1000, sizeof(char));
}
```

Compiling Your Code

- If you are compiling a single source file it is relatively easy and can be done just using gcc

  ```
  gcc -Wall -std=c99 -o example_2 example_2.c
  ```

- **Always use -std=c99**
  - No "for (...) style loops" in older C versions!
  - ANSI standard added a lot of useful improvements
- If you have multiple source files chances are you will need a Makefile
- Makefiles are how their name sounds, they are used to “make” your source code

Compiling Your Code

- Makefiles are simple if you make them simple
- It is often much more practical to reuse and change an existing Makefile instead of making one from scratch
- You can use cmake to generate one for you, cmake is being used a lot now for newer projects
- Googling helps a lot, if you’re using eclipse they will be auto-generated for you, however you often have to customize them

```
# MAKEFILE TEMPLATE
SOURCE_FILES := example.c useful.c other_stuff.c
HEADER_FILES := useful.h other_stuff.h
OBJECT_FILES := main.o useful.o other_stuff.o
FLAGS := -Wall -O0 -g3 -std=c99
EXEC_NAME := example
INSTALL_DIR := /usr/bin/
RM := rm -rf

default: example
debug: example-debug
all: example

example: $(SOURCE_FILES) $(HEADER_FILES)
    @echo 'Building target: $@'
    gcc $(FLAGS) $(SOURCE_FILES) -o $(EXEC_NAME)
    @echo 'Finished building target: $@'

example-debug: $(SOURCE_FILES) $(HEADER_FILES)
    @echo 'Building target: $@'
    gcc $(FLAGS) -DDEBUG $(SOURCE_FILES) -o $(EXEC_NAME)
    @echo 'Finished building target: $@'

install:
    @cp $(EXEC_NAME) $(INSTALL_DIR)

clean:
    @$(RM) $(OBJECT_FILES) $(EXEC_NAME)
```

Debugging

If debugging is the process of removing bugs, then programming must be the process of putting them in.

--Edsger W. Dijkstra

Debugging

- Debugging is an essential part of programming, no one writes code that is bug free
- Most modern IDEs (Eclipse, Visual Studio, IntelliJ) have extensive debugging functionality allowing you to step through every line of your code and view everything as it happens
- We will get into these later using an example program with a bug in it.
- The easiest method can be to simply add “sanity checks” which print out information at parts of your code
- Can use **#define**, **#ifdef** to have debug messages

Debugging

```
#include <stdio.h>
#define DEBUG

int main(void) {
#ifdef DEBUG
     printf("Debugging program\n");
#else
     printf("Running normally\n");
#endif
     return 0;
}
```

Advice

- Become familiar with the C library
  - See the references provided
- **Don't reinvent the wheel!** Use the C library, if it doesn't have what you need find a library that does
  - GSL (Scientific computing)
  - GMP (Scientific computing, infinite precision)
  - SGLIB
  - Gnulib
  - CCAN (Collection of code snippets for C)
  - KLIB

Common Mistakes

- **Always use -std=c99**
  - No "for (...) style loops" with older versions!
  - ANSI standard added a lot of useful improvements
- Multiple header inclusions error when compiling, solution is to just add each new header file using Eclipse (File → New → C Header)
- Segfaults, this is caused by a pointer trying to access memory it doesn’t have access to
  - NULL initialize your pointers is a good habit
- Not null terminating strings (**use calloc!**) or sending wrong array length, remember 0 is first item in array

Resources

- Excellent and Concise C References
  - http://en.cppreference.com/w/c
  - http://www.cplusplus.com/reference/clibrary/
  - http://users.ece.utexas.edu/~adnan/c-refcard.pdf
- Online Books / Tutorials
  - http://gribblelab.org/CBootcamp/
  - http://c.learncodethehardway.org/book/
  - http://publications.gbdirect.co.uk/c_book/
  - http://en.wikibooks.org/wiki/C_Programming
  - http://www.cprogramming.com/
  - http://www.tutorialspoint.com/cprogramming/index.htm

Resources

- Detailed C References
  - http://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html
  - http://www.gnu.org/software/libc/manual/html_mono/libc.html
  - http://www.acm.uiuc.edu/webmonkeys/book/c_guide/
  - http://www.c-faq.com/
- POSIX/UNIX Reference
  - http://www.gnu.org/software/libc/manual/html_mono/libc.html
  - http://en.wikipedia.org/wiki/Unistd.h

Resources

- POSIX/UNIX Overview
  - http://cplus.kompf.de/posixlist.html
  - http://www.cs.cf.ac.uk/Dave/C/CE.html
  - http://www.tutorialspoint.com/cprogramming/index.htm