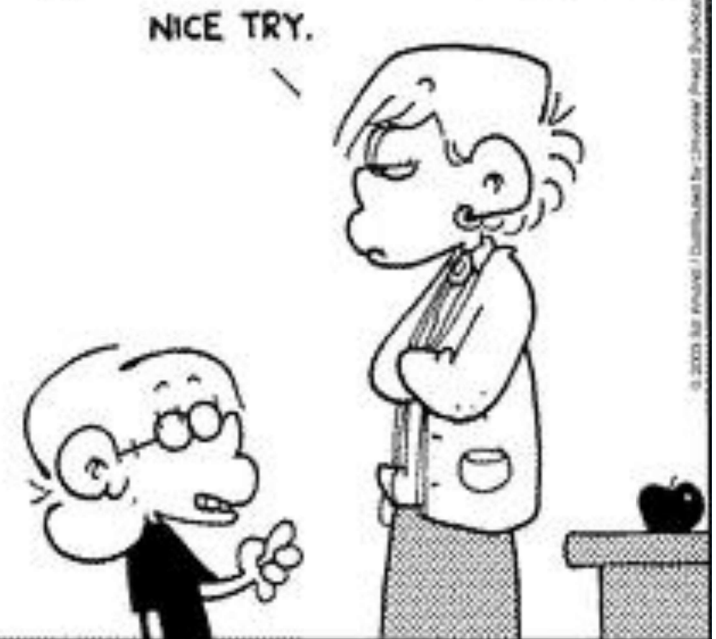


C – The Language

```
#include <stdio.h>
int main(void)
{
    int count;

    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.");
    return 0;
}
```

AMEND 10-3



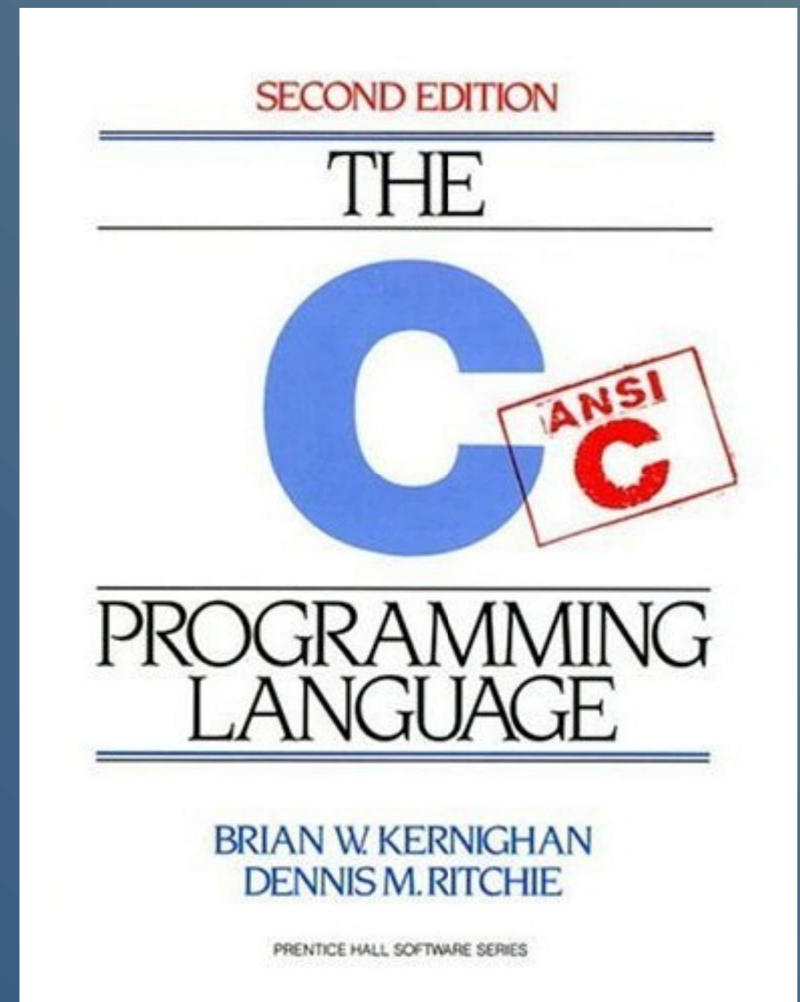
Topics

- 1) Background of C.
- 2) **IDE** for cross development.
- 3) **Modular C** programs and **good design**.
- 4) How to use **printf()**, **strings**, **macros**...

A Brief History Of C

K&R C

- Kernighan and Ritchie published book:
"The C Programming Language" in 1978.
 - Developed at Bell Labs for Unix in 1969 by Ritchie. Note: Ritchie one of original UNIX authors.
 - Designed for writing system software.



Importance of K&R

Linux kernel style guide on where to put the {'s:

"...the preferred way, as shown to us by the prophets Kernighan and Ritchie, is to put the opening brace last on the line, and put the closing brace first, thusly:

```
if (x is true) {  
    we do y  
}
```

...However, there is one special case, namely functions: they have the opening brace at the beginning of the next line, thus:

```
int function(int x)  
{  
    body of function  
}
```

Heretic people all over the world have claimed that this inconsistency is ... well ... inconsistent, but all right-thinking people know that (a) K&R are right and (b) K&R are right...."

Standards

- **ANSI C (1989) / ISO C90**
 - An updated version of K&R C.
 - First agreed on "standard".
- **C99 update added these and more:**
 - inline functions, mix variable declaration in function.
 - // commenting style
- **C11 update added these and more:**
 - threading support, Unicode support,
 - Bounds-checking string functions: `strcat_s()`
- A lot of code still written to ANSI C.
 - It works everywhere!

Important Things Missing vs C++

- **No classes:**
 - use **structs** for data,
module-organization for code.
- No pass **by reference**: use pointers.
- No **overloading** (functions or operators)

Some Differences vs C++

- `true` and `false` defined in `#include <stdbool.h>`
Use type `bool`
- `const` a little different vs C++
(It's not a compile-time constant, so cannot always declare other constants based on previous `const`)
 - C programs often use:
`#define MY_CONST 10`
- C does not strictly enforce function arguments:
`void foo();` // could also be header for:
`void foo(char *msg, int size, double change) {...}`
 -
 - Always include the correct headers & full prototypes.

IDE for Cross Development

Why an IDE?

- Integrated Development Environment (IDE)
 - IDEs have powerful editing features which support your **efficiency**.
 - Syntax highlighting, auto format, auto-complete
 - Integrated build and error display
 - Integrated graphical debugger.
- I recommend you setup an IDE for cross-development.
 - I will support **VS Code** and **Eclipse**.
 - Feel free to pick your favourite.

Modular Design

Modular Design

- Cannot do OOD: no objects!
- Use a modular design where...
 - Each component's interface is its `.h` file.
 - Implementation is `.c` file
- When reviewing the *quality* of a large C program, I first look at how modular its components are.
 - If you do nothing else, learn this!

Naming Conventions

keypad.h

```
// "Public" functions/constants  
#define KEYPAD_NUM_KEYS 10  
bool Keypad_isSomeButtonDown();
```

keypad.c

```
// "Private" functions/variables  
#define DEFAULT_BUTTON 0  
static int buttonState = 0;  
bool Keypad_isSomeButtonDown()  
{...}  
static void initButtons() {...}
```

- - Either inline (C99 `//...`) or block (`/* ... */`).
 - Comment static functions only when needed.
- Pick a consistent indentation style and stick with it.
 - Suggestion: Try the [Linux Kernel style](#).

What happens?

main.c

```
1  #include "other.h"
2  #include <stdio.h>
3
4  int main() {
5      int count = Other_getCount();
6      printf("Count is %d\n", count);
7
8      extern int hidden_count;
9      hidden_count++;
10
11     count = Other_getCount();
12     printf("Count is %d\n", count);
13 }
```

other.h

```
1  #ifndef _OTHER_H_
2  #define _OTHER_H_
3
4  int Other_getCount();
5
6  #endif
```

other.c

```
1  int hidden_count = 0;
2
3  int Other_getCount() {
4      return hidden_count;
5  }
```

- a) Compile-time error.
- b) Runtime error.
- c) Prints 0 and 0
- d) Prints 0 and 1

Linkage



Function or global variable accessible in..

printer.c

```
int badGlobal = 1;  
bool Printer_hasPaper() { ...}
```

other.c

```
extern badGlobal;  
void foo() {  
    badGlobal *= 2;  
}
```



Function or global variable accessible in..

printer.c

```
static int numPages = 0;  
static void updatePaperStatus() { ... }
```



Rule of thumb

- Make functions and global variables static unless..

Fight “Globalization” (C Style)

- Getting rid of externally linked global variables
 - Turn a global variable into..
- Example
 - How could a printer module store the number of pages in the printer?

- Bad (in printer.c)

```
int Printer_pageCount = 0;
```

Other code can read and write this variable.

- Better (in printer.c)

```
static int pageCount = 0;  
int Printer_getPageCount() {...}  
void Printer_updatePageCount() {...}
```

All code to update the value found inside the module..

Card Deck Example

card.h

```
// Represent a single card.
#ifndef CARD_H_
#define CARD_H_

typedef struct {
    // Suit can be one of:
    // 'C', 'H', 'D', 'S'
    char suit;
    // Value can be one of:
    // '2', ..., '9', 'J', 'Q', 'K', 'A'
    char value;
} sCard;

#endif
```

deck.h

```
// Manage a standard deck of cards.
#ifndef DECK_H_
#define DECK_H_

#include "card.h"

#define NUM_CARDS_IN_DECK 52

void    Deck_initialize(void);
sCard    Deck_getNextCard(void);
int      Deck_getNumCards(void);

#endif
```

Card Deck Example (cont)

deck.c

```
#include "deck.h"
#include <stdbool.h>

static sCard cards[NUM_CARDS_IN_DECK];
static bool initialized = false;
static int numCardsLeft = 0;

// Local Headers (for inside .c file only)
static void populateCards(void);
static void shuffleCards(void);
static void removeTopCard(void);
```

..

Need headers so these functions can be called regardless of order in file.

Card Deck Example (cont)

// deck.c continued...

```
void Deck_initialize(void)
```

```
{  
    populateCards();  
    shuffleCards();  
    initialized = true;  
}
```

Call functions with internal linkage; implementations is below here, so must have the headers.

```
static void populateCards(void)
```

```
{  
    for (int i = 0; i < NUM_CARDS_IN_DECK; i++) {  
        /* ... */  
    }  
}
```

Variable in for loop requires C99

```
static void shuffleCards(void)
```

```
{    /* ... */ }
```

...

Example

- Modular design of SFU's electronic lab-door locks.
 - What modules?
 - What functions in each module?

Real C: Example 1

AGC_Processing.h

(AGC is Automatic Gain Control, to make audio volume seem consistent)

```
void Proc_AGC(void);
void Init_AGC (void);

// Global variables exposed by AGC module
extern int  PGAGain;
extern int  AGC_Mode;
extern float DDeltaPGA;

extern float AGC_Mag;
extern int   RSL_Mag;

extern int PGAGAIN0;
extern int dac_gain;
extern int AGC_Signal;

extern int AGC_On;
extern int Old_PGAGAIN0;
extern int Old_dac_gain;

extern int RSL_Cal;
```

Real C: Example 2

options.h

```
// This sequence must be the same as options.c
typedef enum {
    OPTION_RX_AUDIO = 0,
    OPTION_RX_RF,
    OPTION_AGC_Mode,
    NUM_OPTIONS
} OptionNumber;

// Initialization
void Options_Initialize(void);
void Options_ResetToDefaults(void);

// Work with option data
const char* Options_GetName(int optionIdx);
int16_t Options_GetValue(int optionIdx);
void Options_SetValue(int optionIdx, int16_t newValue);
uint16_t Options_GetMinimum(int optionIdx);
uint16_t Options_GetMaximum(int optionIdx);
uint16_t Options_GetChangeRate(int optionIdx);
```

Some C Details

C Dynamic Allocation

- No "new"; use `malloc()`:

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

```
#define NUM_TREES 5
```

```
void foo() {
```

```
    // What's going on here?
```

```
    float *pHeights;
```

```
    pHeights = malloc(sizeof(*pHeights) * NUM_TREES);
```

`malloc()` arg is #bytes to allocate

```
    if (!pHeights)
```

```
        exit (EXIT_FAILURE);
```

```
    ....
```

- Free memory using `free()`:

```
    free(pHeights);
```

```
    pHeights = NULL;
```

For safety.

· 2nd free does nothing.
(no dangling pointer)

printf

C Code:

```
printf("char      %c\n", 'c');  
printf("decimal   %d\n", 100);  
printf("string    %s\n", "Hello");  
printf("float      %f\n", 3.14);  
printf("hex        %x\n", 0xDEADC0DE);  
printf("unsigned   %u\n", 4000000000U);  
printf("size_t     %zu\n", strlen("hi"));
```

```
printf("Cash $%05.2f\n", 0.1);
```

Output:

```
char      c  
decimal   100  
string    Hello  
float      3.140000  
hex        deadc0de  
unsigned   4000000000  
size_t     2
```

```
Cash      $00.10
```

#define, #ifdef

```
// Use #define for constants:
#define NUM_SHEEP 100    //...
#define PROMPT "Hello> "
#define DEBUG_LEVEL 2

// Selective Compilation:
#ifdef DEBUG_BUTTONS
    printf("Button read: \n", daButton);
#endif

#if DEBUG_LEVEL > 3
    printf("Button read: \n", daButton);
#endif
```


Strings

- C Strings
 - ..
- Example

```
char buff[100];  
snprintf(buff, 100, "Hi");
```
- Forgetting null is a **common bug!**
 - Use string functions with 'n' in name: they are passed the size of the buffer
 - Avoids buffer overflow



Strings: snprintf, strlen, strncmp

```
#define BIG 100
#define DAYS_PER_YEAR 365
void demoBasics()
{
    char buff[BIG];
    int numYears = 4;

    sprintf(buff, "that is %d days",
            numYears * DAYS_PER_YEAR);

    printf("%s\n", buff);

    printf("%zu\n", strlen(buff, BIG));

    printf("%d\n", strncmp(buff, "that", BIG));
}
```

that is 1460 days

17

-1

- Spot the bug
 - What if: #define BIG 13?

Strings: Copy

```
#define BIG 100
#define SMALL 5
```

```
void demoCopyToBuff()
{
```

```
    char buff[BIG];
    char smallBuff[SMALL];
```

```
    snprintf(buff, BIG, "Hello world (%d)!", 433);
```

```
    strncpy(smallBuff, buff, SMALL);
    smallBuff[SMALL-1] = 0;
```

\$ man strncpy

... Warning: If there is no null byte among the first n bytes of src, the string placed in dest will not be null-terminated.

```
    printf("Source: %s\n", buff);
    printf("Target: %s\n", smallBuff);
```

```
}
```

Source: Hello world (433)!

Target: Hell

Strings: Append

```
#define BIG 100
void demoAppend()
{
    char buff[BIG];
    snprintf(buff, BIG, "Hello CMPT %d", 433);

    // More safely append to end.
    int freeSpace = BIG - strlen(buff, BIG) - 1;
    strncat(buff, " world!", freeSpace);

    printf("%s\n", buff);
}
```

\$ man strncat

... As with strcat(), the resulting string in dest is always null-terminated.

Hello CMPT 433 world!

Strings: Append

```
void demoAppendAbuse()  
{  
    const size_t MAX_LEN = 16;  
    char buff[MAX_LEN];  
    snprintf(buff, MAX_LEN, "Your grade is ");  
  
    // Append (without overflow)  
    const char* copyMe = "AN F!";  
    size_t freeSpace = MAX_LEN - strlen(buff, MAX_LEN) - 1;  
  
    // Checking for truncation  
    if (strlen(copyMe) > freeSpace) {  
        err(EXIT_FAILURE, "Err: String would be truncated");  
    }  
    strncat(buff, copyMe, freeSpace);  
    printf("%s\n", buff);  
}
```

Your grade is A

Err: String would be truncated

Functions to Avoid (“banned”)

- String functions without size (“n”):
 - strcat()
 - strtok(), strtok_r()
 - sprintf(), vsprintf()
 - gets()
- May not null-terminate strings
 - strcpy(), strncpy()
- Problematic
 - strncat()

Must calculate how many characters can be added

Will truncate what’s being copied without warning
(truncated string could be problematic!)

Strings: Number from String

```
void demoToInt()
{
    long fromStr = atol("-987654321");
    printf("%ld\n", fromStr);

    #define BASE 16
    long fromHexStr = strtol("0xDEADCODE", NULL, BASE);
    printf("%ld\n", fromHexStr);
}
```

-987654321

912092

Macros

Put brackets around parameters:

```
#define WTOD(w) ( (w) * 7)
#define BAD_WTOD(w) ( w * 7)
...
int days = BAD_WTOD (1+2);
```

Multi-line and statements:

```
#define WAIT_LONG() do {\
    sleep();\
    sleep();\
    sleep();\
} while(0)
```

```
#define MIN(x, y) ( (x) < (y)? (x) : (y))
```

// Avoid side effects:

```
int a = 1, b = 10;
int c = MIN(a++, b++);
// becomes:
```

```
// Error during compilation.
#error "Die here!"
```

Loop variable and Struct

- Only in C99 can declare variables in for loop initializer:

- C99:

```
for (int i=0; i<10; i++) {  
    ...  
}
```
- ANSI (old-school):

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

- Struct

```
#define MAX_LEN 200  
struct student_t {  
    char name[MAX_LEN];  
    int age;  
    float height;  
};  
  
struct student_t s1;
```

-

Example

- Class Exercise

- Design interface for joystick module

- initialize, cleanup,
 - check if joystick pressed in a specific direction
 - get the name (string) for a joystick direction.

- Show Implementation

- Use an array of `structs` inside the module to store information about the directions.

Error Handling

Return Values

- C Functions often..
- **Returning Success / Fail**
 - Some functions return 0 or 1 to indicate success; -1, or 0 to indicate failure
 - **Ex: `fclose()`**: 0 for success, EOF (-1) for failure.
`if (fclose(my_file) == EOF) {`
 `perror("Unable to close file.");`
 `exit(EXIT_FAILURE);`
`}`

errno

- **errno: Error code**
 - ..
to track which error was encountered
 - C library functions and system calls can set **errno** to indicate which error occurred (if any)
- **exit(): terminates program**
 - `exit(EXIT_SUCCESS);`
 - `exit(EXIT_FAILURE);`
- **perror() prints a message based on errno:**

```
char ch;  
if (fscanf(myFile, "%c", &ch) == EOF) {  
    perror("fscanf error");  
    exit(EXIT_FAILURE);  
}
```

```
fscanf error: Bad file descriptor
```

In-Band Error Indicators

- In-band Error Indicators

- Many functions return useful information

Ex: `int ch = getchar();`

- These functions may indicate an error by returning a value not otherwise possible.

- `getchar()`'s Error Reporting

- Returns a character (0 to 255?) on success, or EOF (-1, likely) on failure
- Common usage:

```
int ch;  
while ( (ch = getchar()) != EOF) {  
    do_something(ch);  
}
```


Some C Standard Library Functions

C Library Function	Return Value on Success	Return Value on Failure
<code>fclose()</code>	0	EOF (negative)
<code>fgetc()</code>	Character read	EOF
<code>fgets()</code>	Pointer to string	NULL
<code>fopen()</code>	Pointer to stream	NULL
<code>scanf()</code>	Number of conversions (nonnegative)	EOF (negative)
<code>sprintf()</code>	Number of non-null characters written	Negative
<code>strtol()</code>	Converted value	LONG_MAX or LONG_MIN, <code>errno == ERANGE</code>
<code>time()</code>	Calendar time	<code>(time_t)(-1)</code>

Summary

- **Version of C:** K&R, Ansi C/C90, C99, C11
- Use a powerful IDE for cross development.
- Use **modular design** & **naming convention**.
- **Details:**
 - **malloc()** & **free()**
 - **printf()** types: **%c**, **%d**, **%s**, **%f**, **%x**, **%u**
 - **#define**, **#ifdef**
 - String functions: **sprintf()**, **strncmp()**, **strncpy()**, **strlen()**...
 - Macros: Put parameters in brackets.
- Carefully check for errors; use **errno**, **perror()**