```c
#include <stdio.h>
int main(void)
{
    int count;
    for (count = 1; count <= 500; count++)
        printf("I will not throw paper airplanes in class.\n");
    return 0;
}
```
Topics

1) Background of C.
2) Eclipse 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

  - 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 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.
Eclipse Demo

• Demo Eclipse for Cross-Development
  – Project creation.
  – Build with makefile; edit makefile.
    • Window → Show View → Other, Make → Make Target
  – IDE Features:
    • Tab-complete, mouse-over.
    • Project navigation:
      – Ctrl+Click function name: browse into.
      – Ctrl+Alt+h: where is function called
    • Refactor: rename (Alt+Shift+R).

• Debugging with Eclipse will be demonstrated later.

See Debugging Guide for install and use directions.
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

- Either inline (C99 //...) or block (/* ... */).
- Comment “locals” only when needed.

- Pick a consistent indentation style and stick with it.
  - Suggestion: Try the Linux Kernel style.

```c
#include <stdio.h>

#define KEYPAD_NUM_KEYS 10

_Bool Keypad_isSomeButtonDown();
```

```c
#define DEFAULT_BUTTON 0
static int buttonState = 0;
static void initButtons() {...}
```

See Style-guide on website.
Linkage

- Function or global variable accessible in...

  ```c
  printer.c
  int badGlobal = 1;
  _Bool Printer_hasPaper() { ... }
  ```

- Function or global variable accessible in...

  ```c
  printer.c
  static int numPagesInPrinter = 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)
  ```c
  int Printer_pageCount = 0;
  ```
- Better (in printer.c)
  ```c
  static int pageCount = 0;
  int Printer_getPageCount() {...}
  void Printer_updatePageCount() {...}
  ```
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.
// deck.c continued...

void Deck_initialize(void)
{
    populateCards();
    shuffleCards();
    initialized = true;
}

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

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_AGCPGAGainAGC_ModeDeltaPGAAGC_MagRSL_MagPGAGAIN0dac_gainAGC_SignalAGC_OnOld_PGAGAIN0Old_dac_gainRSL_Cal;
// 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():

```c
#include <stdlib.h>
#define NUM_TREES 5

void foo() {
    float *pHeights;
    pHeights = malloc(sizeof(*pHeights) * NUM_TREES);
    if (!pHeights)
        exit(EXIT_FAILURE);

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

- Free memory using free():

```c
free(pHeights);
pHeights = NULL;
```

For safety.

\[ 2^{nd} \text{ free does nothing. (no dangling pointer)} \]
<table>
<thead>
<tr>
<th>C Code</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>printf(&quot;char %c\n&quot;, 'c');</code></td>
<td>char c</td>
</tr>
<tr>
<td><code>printf(&quot;decimal %d\n&quot;, 100);</code></td>
<td>decimal 100</td>
</tr>
<tr>
<td><code>printf(&quot;string %s\n&quot;, &quot;Hello&quot;);</code></td>
<td>string Hello</td>
</tr>
<tr>
<td><code>printf(&quot;float %f\n&quot;, 3.14);</code></td>
<td>float 3.140000</td>
</tr>
<tr>
<td><code>printf(&quot;hex %x\n&quot;, 0xDEADC0DE);</code></td>
<td>hex deadc0de</td>
</tr>
<tr>
<td><code>printf(&quot;unsigned %u\n&quot;, 4000000000U);</code></td>
<td>unsigned 4000000000</td>
</tr>
<tr>
<td><code>printf(&quot;Cash $%05.2f\n&quot;, 0.1);</code></td>
<td>Cash $00.10</td>
</tr>
</tbody>
</table>
// 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 are null-terminated char arrays:

```c
#define LEN 1024
char msg[LEN];
sprintf(msg, "Answer is: %d", 42);
printf("1. %s\n", msg);

printf("2. %d\n",
       strncmp(msg, "Answer", 6));

#define SLEN 4
char target[SLEN];
strncpy(target, msg, SLEN);
target[SLEN-1] = 0;
printf("3. %s\n", target);

strncat(msg, " world!", LEN);
printf("4. %s\n", msg);

printf("5. %d\n", strlen("Hello World!"));
printf("6. %d\n", atoi("-987654321"));
```

1. Answer is: 42
2. 0
3. Ans
4. Answer is: 42 world!
5. 12
6. -987654321
Macros

Put brackets around parameters:
#define WTOD(w) ( (w) * 7)
#define BAD_WTOD(w) ( w * 7)
...
int days = BAD_WTOD (1+2);

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

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

Multi-line and statements:
#define WAIT_LONG() do {
    sleep();
    sleep();
    sleep();
} while(0)

// Error during compilation.
#error "Die here!"
C++ vs C while(1)

- C++ allows infinite loops to be optimized out by the compiler.
  - This supports compiler optimizations for merging loops.

Solution

```
volatile int myVolatile = 0;
while (1) {
    // Prevent optimizing out
    myVolatile++;
}
```

C handles it differently:

```
while (1) // Undefined
    // behaviour!
printf("done!");
```

"An iteration statement whose controlling expression is not a constant expression, that performs no input/output operations, does not access volatile objects, and performs no synchronization or atomic operations in its body ..., may be assumed by the implementation to terminate."

C11 spec, 6.8.5
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

```c
#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.
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.