C PROGRAMMING

INTRODUCTION

This is not a full c-programming course. It is not even a full 'Java to c' programming course.

LITTERATURE

FOR C-PROGRAMMING

- The C Programming Language (Kernighan and Richie) - The bible, and the book that is referenced in this material.
- Essential C
- Online c-course
- Another online c-course
- Yet another online c-course

FOR VALGRIND

- Quickstart guide
- User manual

FOR MAKEFILE AND DDD

- Tutorial for makefile
- Manual for ddd debugger

GETTING STARTED

HELLO WORLD

hello_world.c

```c
#include<stdio.h>

int main(void) {
        printf("Hello world!\n");
        return 0;
}
```

Compile using:

```
gcc hello_world.c
```

which will generate **a.out**

Run using:

```
./a.out
```

COMPILER FLAGS

You can (and are recommended to) compile with several flags to the compiler:

- -g = debug information
- -Wall = All warnings are displayed
- -O9 = Optimization

Which makes it:

```
gcc -g -Wall hello_world.c
```

VARIABLE TYPES

```
char
int
float
double
```

short/long can specify size.

signed (use 1 bit for sign)/unsigned

`const int i = 42;` → i is not a variable.

STATIC VARIABLES

You can declare a variable static, and if done in global scope, use it inside functions etc.

```c
#include<stdio.h>

static int i = 0;

int main(void) {
    printf("%i\n", i);
    i = i+1;
    printf("%i\n", i);
    return 0;
}
```

GLOBAL VARIABLES

Prefixing variables with **extern** makes them globally visible from other files.

STRINGS AND ARRAYS

Strings are arrays of char

```
Hello"
```

is the same as

```
char str[] = {'H','e','l','l','o','\0'}
```

We will cover much more on strings when we get to pointers.

```c
#include<stdio.h>

int main(void) {
    char str[] = {'H','e','l','l','o','\0'};
    printf("%s - %s\n", str, "Hello");
    return 0;
}
```

ENUMS

```c
#include<stdio.h>

int main(void) {
    enum firstmonths{JAN,FEB,MAR,APR};
    enum lastmonths{NOV=10,DEC=11};
    printf("%i\n", JAN);
    printf("%i\n", NOV);
    return 0;
}
```

All enumerated values must be different, and will start with 0 if not explicitly defined.

OPERATORS

+ - * / %

COMPARISON:

<
<=
>
>=
==
!=

LOGIC

0 is false, everything else is true

! is used for negation

&& is boolean AND

| | is boolean OR

BITWISE OPERATORS

& is bitwise AND

| is bitwise OR

^ is bitwise XOR

<< is left shift

BITWISE OPERATORS

```c
#include<stdio.h>

static unsigned int binary0110 = 6;
static unsigned int binary0010 = 2;

int main(void) {
    printf("%i\n", binary0110 & binary0010 );
    printf("%i\n", binary0110 | binary0010 );
    printf("%i\n", binary0110 ^ binary0010 );
    printf("%i\n", binary0110 << 1 );
    printf("%i\n", binary0110 >> 1 );
    return 0;
}
```

PRINTF

Arguments

- `%i` int/decimal numner
- `%u` Unsigned number
- `%c` Single character
- %s Prints a string
- %p Pointer address

PRINTF

There are more arguments to specify adjustment and width, check the book/manual.

- If you create a segfault, and you do not use \n with your printf, you are not guaranteed that it is printed!

INCREMENT/DECREMENT

++ and - - can be used both before and after an operator.

```
i++
```

Increments i **after** its value is used

```
++i
```

Increments i **before** its value is used

INCREMENT/DECREMENT

```c
#include<stdio.h>

int main(void) {
    int i,j,before,after;

    i = 0;
    j = 0;

    before = ++i;
    after = j++;

    printf("B: %i A: %i i: %i, j: %i\n", before, after, i, j);
    return 0;
}
```

CONTROL FLOW

Looks like java/python etc.

LOOPS

```c
#include<stdio.h>

int main(void) {
    int i;

    for( i = 0; i < 10; i++) {
        printf("i: %i\n", i);
    }

    i = 0;
    while( i < 10 ) {
        printf("i: %i\n", i);
        i++;
}
```

IF/ELSE

```c
#include<stdio.h>

int main(void) {
    int i;

    i = 1;
    if( i ) {
        printf("if statement\n");
    } else {
        printf("else statement\n");
    }

    return 0;
}
```

SWITCH

```c
#include<stdio.h>

int main(void) {
    int i;
    switch( i ) {
        case 0:
            printf("i is 0\n");
            break;
        case 1:
            printf("i is 1\n");
            break;
        default:
            printf("i is not 0 or 1\n");
            break;
    }
    return 0;
}
```

BREAK

break can be used in both loops and switches to exit them.

continue can only be used in loops, and jumps to the beginning of the loop.

STYLE COMMENTS

A nice c-style guide can be found at
http://www.doc.ic.ac.uk/lab/cplus/cstyle.html

- Declare all variables of a function before anything else. Including before you assign them any value

FUNCTIONS, STRUCTS, ETC.

FUNCTIONS

Remember to forward declare functions. If you use a function before it is declared, it is an error.

```c
#include<stdio.h>

// The add method is forward declared, because it is used before it is defined.
int add(int a, int b);

int main(void) {
    int i,j;
    i = 2;
    j = 3;
    printf("i+j function style=%i\n", add(i,j) );
    return 0;
}
int add(int a, int b) {
    return a + b;
}
```

RECURSIVE FUNCTIONS

Notice again the forward declaration

```c
#include<stdio.h>

int factorial(int i);

int main(void) {
    printf("5!=%i\n", factorial(5) );
    return 0;
}

int factorial(int i) {
    if( i == 0) {
        return 1;
    }
    return i * factorial(i-1);
}
```

HEADER FILES/INCLUDING OTHER FILES

To divide code into logical parts, you can split it across multiple (pair of) files.

The function declarations, defined variables, structs etc. (think the public stuff), is declared in a header file, ending with .h.

The implementation is kept in a code file, ending with .c

INCLUDING HEADER FILES

Both user and system header files are included using the preprocessing directive `#include `. It has following two forms:

```
#include <file>
```

This form is used for system header files. It searches for a file named file in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code.

```
#include "file"
```

This form is used for header files of your own program. It searches for a file named file in the directory containing the current file. You can prepend directories to this list with the -I option while compiling your source code.

ONCE-ONLY HEADERS

If a header file happens to be included twice, the compiler will process its contents twice and will result an error. The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef HEADER_FILE
#define HEADER_FILE

the entire header file file

#endif
```

This construct is commonly known as a wrapper **#ifndef**.

When the header is included again, the conditional will be false, because `HEADER_FILE` is defined.

The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

STRUCTS

A Struct is a group of variables, that belong togeather in a logical way.

```c
#include<stdio.h>

struct Account {
    double amount;
    int account_id;
};

int main(void) {
    struct Account account;
    account.account_id = 19203827;
    account.amount = 1225.55;

    printf("Acc id %i amount: %g\n", account.account_id, account.amount);

    // Ugly shortcut - ordering matters.
    struct Account account2 = { 2541.75, 32568941 };
    printf("Acc. id %i amount: %g\n", account2.account_id, account2.amount);

    return 0;
}
```

TYPEDEF

To avoid typing struct a lot of times, it is common to use typedef

```c
#include<stdio.h>

typedef struct _account {
    double amount;
    int account_id;
} account;

int main(void) {
    account acc;
    acc.account_id = 19203827;
    acc.amount = 986.33;

    printf("Account id %i amount: %g\n", acc.account_id, acc.amount);
    return 0;
}
```

UNION

In datastructures etc. you sometimes need to use one or another set of variables. A union only uses space like the biggest of the sets

```c
#include<stdio.h>
union Card { /* Playing card */
  int value;
  char face[8]; // Try using 9 - explain
};
int main(void) {
  union Card card1; union Card card2;
  card1.value = 5;
  card2.face[0] = 'Q'; card2.face[1] = 'u'; card2.face[2] = 'e';
  card2.face[3] = 'e'; card2.face[4] = 'n'; card2.face[5] = '\0';
  printf("Card1: %i Card2 %s\n", card1.value, card2.face);
  printf( "Mem: %li = %li\n", sizeof(card1), sizeof(card2));
  printf("Card1: %s Card2 %i\n", card1.face, card2.value);
  return 0;
}
```

DEFINES

Defines are inserted before the code is compiled.

```c
#include<stdio.h>

#define VALUE 42
#define DIV(a,b) (a)/(b)

int main(void) {

    printf("Value: %i\n", VALUE);
    printf("DIV(6,3): %i\n", DIV(6,3));

    return 0;
}
```

POINTERS AND ARRAYS.

POINTERS

Every variable is located somewhere in memory, and thus have an adress where it is located.

A pointer is a variable, that is an adress of another variable.

A pointer is defined with a star in from of the variable name:

```
int *pointer;
```

You can declare more pointers on the same line, but the star must be with each variable:

```
int *pointer, *also_pointer;
```

To get the adress of a variable, you can use the & operator

POINTERS

The following example is commented for the operators used with pointers

```c
#include <stdio.h>

int main(void) {

    int x = 42;  /* A normal integer*/
    int *p;      /* A pointer to an integer ("*p" is integer,
                    so p must be a pointer to an integer) */
    p = &x;       /* Assign the address of x to p */

    printf( "x has value %i\n", x);
    printf( "Its address is %p\n", p);

    /* Note the use of the * to get the value */
    printf( "Using pointer i, the value of x: %i \n", *p );
    return 0;
}
```

POINTERS

In this example, the value of i is changed by **change()** even through it does not have i in its scope

```c
#include <stdio.h>

void change(int *pointer) {
    *pointer = 42;
}

int main(void) {
    int i = 2;
    change(&i);
    printf("i: %i\n", i);
    return 0;
}
```

POINTERS WITH STRUCTS

The arrow: → is a shorthand notation, for retrieving values from a struct pointer

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct _Account {
        double amount;
        int account_id;
} Account;
int main(void) {
        Account account;
        Account *accPointer;
        account.account_id = 19203827;
        account.amount = 1225.55;
        accPointer = &account;
        printf("amount: %g\n", (*accPointer).amount );
        printf("amount: %g\n", accPointer->amount );
        return EXIT_SUCCESS;
}
```

ARRAYS REVISITED

Arrays are just pointers to its first element!

The following is therefore equivalent

Table 1. Table title

Array access	Pointer access
arr[0]	*arr
arr[2]	*(arr + 2)
arr[n]	*(arr + n)

MEMORY HANDLING AND I/O.

MEMORY HANDLING

Local variables are placed in the Stack part of memory, and are only valid in the scope of the method. Once a method has returned, the memory is no longer available for that variable.

For more permanent location, memory must be allocated on the heap. For this you can use the `void *malloc(int num);` function. It allocates an array of num bytes and leave them initialized.

When the memory is no longer needed, you can use `free()` to hand it back to the operating system.

MEMORY HANDLING (1)

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

int main(void) {
    char string1[100];
    char *string2;

    (1)

    return EXIT_SUCCESS;
}
```

1 See next slide

MEMORY HANDLING (1)

```c
// Using a string copy function
strcpy(string1, "I'm an array, memory on stack");
/* allocate memory dynamically */
string2 = malloc( 200 * sizeof(char) );
// If malloc returns NULL, something went wrong (out of memory)
if( string2 == NULL ) {
  // fprintf: Output written to stream, here: std error
  fprintf(stderr, "Error: unable to allocate required memory\n");
} else {
  strcpy( string2, "I'm now storing data on the heap");
}
printf("string1 = %s\n", string1 );
printf("string2 = %s\n", string2 );
// I'm done using the memory - hand it back nicely
free(string2);
```

READING/WRITING FILES

C communicates with files using a new datatype called a file pointer.

This type is defined within **stdio.h**, and written as `FILE *`.

A file pointer called input_file is declared in a statement like

```
FILE *input_file;
```

READING/WRITING FILES

To open aa file, you use `fopen(filename, mode)`. The mode can be one of the following:

- `r` Open file for reading
- `w` Open file for writing
- `a` Open file for appending

You close a file using `fclose(file);`. There is a limit of open files on a system, and it is bad style to not close a file after use.

READING FILES

To read and write input, there are various methods, but an easy one for reading is:

```
char *fgets( char *buf, int n, FILE *fp );
```

The functions `fgets()` reads up to n-1 characters from the input stream referenced by fp. It copies the read string into the buffer buf, appending a null character to terminate the string.

If this function encounters a newline character '\n' or the end of the file EOF before they have read the maximum number of characters, then it returns only the characters read up to that point including new line character.

READING FILES

- Notice that stdin, stdout, stderr all are filepointers, but they are always open, and do not need to be opened with fopen.

WRITING FILES

For writing, the equivalent is:

```
int fputs( const char *s, FILE *fp );
```

The function `fputs()` writes the string s to the output stream referenced by fp. It returns a non-negative value on success, otherwise EOF is returned in case of any error.

WRITING FILES

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

int file_copy(char *filename_in, char *filename_out);
int main(void) {
    return file_copy("./test.txt", "./copied.txt");
}
int file_copy(char *filename_in, char *filename_out) {
        FILE *infile;
        FILE *outfile;
        char line[256];
    (1)
    (2)
    return EXIT_SUCCESS;
}
```

1 See next slide

2 See the following slide

WRITING FILES

```
infile = fopen(filename_in,"r");
if( !infile ) {
        fprintf(stderr, "Could not open input file");
    return EXIT_FAILURE;
}

outfile = fopen(filename_out,"w");
if( !outfile ) {
        fclose(infile);
        fprintf(stderr, "Could not open output file");
    return EXIT_FAILURE;
}
```

WRITING FILES

```
while( fgets(line, 256, infile)) {
        printf("Copying the line: '%s'", line);
        fputs(line,outfile);
}

fclose(infile);
fclose(outfile);
```

If you prefer to read one char at a time, `fgetc` and `fputc` can be used

SORTING AN ARRAY

The build in function `qsort` can be used for this.

```
#include <stdio.h>
#include <stdlib.h>
enum suits{ DIAMONDS, HEARTS, SPADES, CLUBS };
typedef struct _Card { /* Playing card */
   int value;
   int suit;
} Card;
char * getSuit(int suit) {
        switch( suit) {
                case DIAMONDS: return "Diamonds";
                case HEARTS: return "Hearts";
                case CLUBS: return "Clubs";
                case SPADES: return "Spades";
        }
        return NULL;
}
```

1 See following slides

SORTING AN ARRAY

Make a comparator function

```
int compareCards( Card** c1, Card** c2) {
        if( (*c1)->value == (*c2)->value ) {
                return (*c1)->suit - (*c2)->suit;
        }
        return (*c1)->value - (*c2)->value;
}
```

SORTING AN ARRAY

```
int main(void) {
    int i,j;
    Card *deck[52];

    // Fill array with deck of cards
    for( i = 0; i < 4; i++) {
            for( j = 0; j < 13; j++) {
                    deck[i*13+j] = malloc(sizeof(Card));
                    deck[i*13+j]->value = 1+j;
                    deck[i*13+j]->suit = i;
            }
    }
    (1)
    return 0;
}
```

SORTING AN ARRAY

```
// Print all cards
for( i = 0; i < 52; i++) {
    printf("Suit: %s, value: %i\n", getSuit(deck[i]->suit), deck[i]->value);
}

// Wish sorted by (value,deck)
printf("\nSorting!\n\n");
qsort(deck, 52, sizeof(Card *), (int (*)(const void *,const void *)) compareCards );

// Print all cards
for( i = 0; i < 52; i++) {
    printf("Suit: %s, value: %i\n", getSuit(deck[i]->suit), deck[i]->value);
}
```

MAKEFILE

MAKEFILE WHY ?

- Checks timestamps to see what has changed and rebuilds just what you need, without wasting time rebuilding other files.
- Manage Several source files and compile them quickly with a single Command Line.
- Make layers a rich collection of options that lets you manipulate multiple directories, build different versions of programs for different platforms, and customize your builds in other ways.

MAKE UTILITY

If you run

```
make
```

This will cause the make program to read the `Makefile` and build the first target it finds there.

- If you have several makefiles, then you can execute them with the command: `make -f mymakefile`

For more options use:

```
man make
```

HOW DOES IT WORK?

Typically the default goal in most makefiles is to build a program.

This usually involves many steps:

1. Generate the source code using some utilities like Flex & Bison.
2. Compile the source code into binary file (.o files c/c++/java etc.)
3. bound the Object files together using a linker(gcc/g++) to form an executable program

MAKEFILE STRUCTURE

Makefiles contain definitions and rules.

A definition has the form:

```
VAR=value
```

A rule has the form:

```
output files: input files
commands to turn inputs to outputs
```

All commands must be tab-indented.

To reference the variable VAR, surround it with $(VAR).

RULES

Makefile contains a set of rules used to build an application.

The first rule (default rule) seen by make consists of three parts:

1. The Target.
2. The Dependencies.
3. The Command to be performed.

HOW TO MAKE A SIMPLE MAKEFILE

Here's a simple example for a C program that defines foo() in foo.c, but uses it in bar.c.

```
CC=gcc

bar: bar.c foo.h foo.o
	$(CC) -o bar bar.c foo.o

foo.o: foo.c foo.h
	$(CC) -c foo.c

run:
	./bar

clean:
	rm -f bar foo.o
```

CONVENTIONS

It's common to include pseudo-targets in makefiles to automate other parts of the development process.

```
make clean
```

The rule for clean in the previous example doesn't build a file called clean. Rather, it deletes all of the build targets to allow a fresh build from scratch. Including a clean rule is a common convention for make.

```
make test
```

Running make test often runs the test suite for the software.

```
make install
```

Running make install typically installs the compiled binary in a PATH-accessible location.

For software using the autoconf toolchain, the installation directory is set by running ./configure command prior to using make.

```
make run
```

For simple programs, make run acts like an alias to the compiled binary.

VALGRIND

VALGRIND

The Valgrind tool suite provides a number of debugging and profiling tools that help you make your programs faster and more correct.

The most popular of these tools is called **Memcheck**. It can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behaviour.

PREPARTION

Compile your program with `-g` to include debugging information so that Memcheck's error messages include exact line numbers.

Using `-O0` is also a good idea

- With `-O1` line numbers in error messages can be inaccurate
- Use of `-O2` and above is not recommended as Memcheck occasionally reports uninitialised-value errors which don't really exist.

RUNNING MEMCHECK

If you normally run your program like this:

```
./myprog arg1 arg2
```

Use this command line:

```
valgrind --leak-check=yes ./myprog arg1 arg2
```

Memcheck is the default tool. The --leak-check option turns on the detailed memory leak detector.

CAVEATS

- Memcheck is not perfect; it occasionally produces false positives

DEBUGGING - DDD

DEBUGGING - DDD

DDD is a graphical front-end for GDB and other command-line debuggers.

DDD

Run from command line

```
ddd
```

Must compile programs with -g flag to get debug information included

USAGE

- Step through a program
- Use breakpoints
- See values of viariables

QUESTIONS