# MPCS 51040 – C Programming

## Lecture 3 – Beyond Hello World

Dries Kimpe

April 11, 2017

# Homework 2

Review example solution, Makefile.

## Some remarks

- Do not commit binaries (program, object files)
  (or in general anything which is ‘derived’)
- Careful with whitespace in the makefile (TAB vs space)
  - Think about your dependencies: histogram-in.txt?
  - A few ’for show’ Makefiles...
- Traditional characters are stored in ASCII compatible encoding in C (extensions in C11) and can be treated as numeric values in the 0-255 range.
- Make sure to check return codes of important functions (fopen for example)
  - Be careful with `TRUE` and `FALSE`!!
  - Check the manpage for the exact meaning of the return value of a function

# Quiz

- Quiz during lecture 5
- Everything up to the point, including homework, piazza questions, ...
  - Including HW4 topics
- Will include ‘paper’ programming...

# Git Problems


Some common git issues/questions:

- I can't push.
  Example: `http://stackoverflow.com/questions/10298291/cannot-push-to-github-keeps-saying-need-merge`
  - Likely the issue is that your local commits are cannot go on top of the commits that are in the remote repository.
  - Solution is to merge in remote changes or rebase your local changes on the remote.
- Should I force push? NO (this can destroy data)

# Variables

## Definition

Every variable has a **fixed type known at compile time** (statically typed language!) as well as an associated *scope*, *lifetime* and *storage*.
Variables have a (runtime) *value* and *storage address*.

This means that:

- The compiler knows the exact type (and thus size) of each variable
  (The address is usually only known at runtime)
- The scope is also known at compile time

size, address, read and write (vars.c)

# Types

## Definition

The type determines the meaning of the value stored in a variable or returned by a function. It also determines the set of possible values the variable can have, as well as how these values are represented in memory. There are *object* (such as a number) and *function* (describing a function that can be called) types. Every type has a compile-time, fixed size (use **sizeof** to get that size).

- _Bool, **char**
- signed char, short int, int, long int, long long int.
  (Also **unsigned** versions)
- Real floating types:
  **float** , **double**, **long double**.
- **void**: empty set of values, i.e. no value.

Types can be *qualified* by adding qualifiers such as **const** (different qualifiers make it a different type).
Types can be derived (*derived types*) from other types: for example, a structure, union, array, pointer or function.
**typedef** introduces a *type alias*, not a new type.

typedemo.c: Show type, qualified types, derived types, sizeof.

# Derived Types

## **struct** and **union**

In addition to the *basic* types that are part of the language, new types can be created (as well as aliases (synonyms) for existing types).

Struct members are sequential in memory.

```
struct tagname
{
    members
}
```

Union members *share* the same memory space.

```
union tagname
{
    members
}
```

### Example

```
struct coordinate
{
    int x;
    int y;
} coordinate_var;
struct coordinate var2;
```

NOTE

tag *space* is different from the typedef name *space*. **typedef** can be used to link them. Structs and unions without a tag are *anonymous*.

**struct**–**union**.c: Tag space, typedef, member access.

# Arrays

```c
int x[10]; // Array of 10 ints
unsigned int size=2;
int y[size]; // VLA
char str[10]="test";
```

- Element access is via operator [] (which takes an integer expression).
- There is no array bounds checking; Array indices start at 0.
- C11 arrays can be variable sized (see example).
- Arrays will *decay* into pointer. More on this later.

## C strings

Character arrays are used to store strings.
In C, **strings are terminated by a 0-character**. Each character of the string (stored in a **char**) is represented by a numeric value (typically from the ASCII set).

The size of the array is not the size/length of the string!

String initialization, access, strlen (see `string_init.c`)

# Scope & Lifetime

## Scope

The *scope* of a variable (or identifier in general) is the region where it is accessible, i.e. where the identifier or name is connected to the actual object (address).
Note that

## Lifetime

The *lifetime* of a variable is the period of time during which memory space is allocated to the variable (i.e. during which it can hold a value).

```
1 if (1)
2 {
3     int i=6;
4     for (int i=0; i<20; ++i)
5     { use_i(i); }
6 }
```

The scope will be a subset of the lifetime.

Variable `i` on line 4 ‘shadows’ the variable from line 3. Note that it retains it value (it’s still alive) – the value will be 6 on line 6 – but we can’t access it by name until the variable from line 4 goes out of scope.

# Some terminology

## Statement

- Statements are generally actions to be performed.
- Statements are executed in sequence.
- Statements can be grouped in blocks.
- Example: **if** (a) ;

Grammar: Expression-statement: expression ;
printf ("OK"); is a statement.

## Expressions

- Expressions have a type and can be evaluated.
- In C, expressions can have side-effects (for example: a=10).
- Example: 10 or printf ("OK")

Grammar: A.2.3 from `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`

# Some terminology

## Continued...

### Declaration

- Tells the compiler what *kind* an identifier is (and some of its properties).

### Definition

- Provides full information about an object. For functions, what the function does.

### Example

- The body of a function is made up out of statements and declarations.
- A translation unit (C file) consists out of declarations and function-definitions.

# Control Flow

## if

- **if** ( expression ) statement
- **if** ( expression ) statement **else** statement

### Example

```c
if ((23 + 4) > 10) printf ("OK");

if (a)
   if (b)
      callfunc();
   else
      callfunc2();
```

- Expression must have a scalar type.
- Will be compared to ‘0’ (equal for else statement).
- **else** associates with lexically nearest preceding if that allows it.
- Question: why no semi column after statement?

You can compare against 0, you generally can’t compare against 1!

# Control Flow

## switch/case

- **switch** ( expression )
- **case** expression :
- **default** :

### Example

```c
switch(a)
{
    case 0:
        printf("Zero");
        break;
    case 1:
        printf("One");
        break;
    default:
        printf("Many");
}
```

- Expression of switch must have integer type.
- Expression of each case must be integer constant
  (i.e. computable at compile time).
- Unique values for all cases in each **switch**.
- `switch` causes jump to statement following matching **case** (or **default**).
  If no match, nothing within **switch** is executed.
- Remember to **break** or execution will continue with the next case label!

# Iteration

## `while` and **do**

- **while** ( expression ) statement
- **do** statement **while** ( expression ) ;

```
while (x<10)
{ ++x; }

do { ++x; } while (x<10);

while (x<10)
{
    if (++x==3) continue;
    printf("num");
}
```

- Expression must have scalar type.
- Iterates until expression is equal to 0.
- **while** performs test *before* executing loop body, **do** *after*.
- **break** stops iteration and exits the loop body right away.
- **continue** skips the rest of the loop body and possible starts the next iteration.
- **continue** and **break** apply to the inner loop.

# Iteration

## for

### Syntax

- **for** ( expression1 ; expression2 ; expression3 ) statement
- **for** ( declaration ; expression2 ; expression3 ) statement

```
for (int i=0; i<100; ++i)
   callfunc();

for ( ; ; )
{ callfunc();  }
```

- Evaluate expression1 (or execute declaration) a single time.
- if expression2 is non-zero execute loop body.
- Evaluate expression3 before going back to the previous step.
- For the declaration, the scope is up to the end of the for body, including expression2 and expression3.
- If expression2 is omitted, it is replaced by a non-zero constant.

iteration .c:

- Demonstrate **break**, **continue**
- Modifying loop variables

# Comments

## Example

```
 // THis is a single line comment

 /*
This is a multi-line comment.
It ends with: */
```

Multi-line comments do not nest

- Can comments appear anywhere? Why (not)?
- When to use which style?

# Assertions

**Example**

```c
#include <assert.h>

int divide (int x, int y)
{
    assert (0!=y);
}
```

**Example**

```c
#include <assert.h>
static_assert(
    sizeof(long long)>=8,
    "need 64 bits or more");
```


- `assert` is disabled by defining NDEBUG.
- Be careful for side-effects!
- `static_assert` is executed at *compile time*.
  - Can only use information known at compile time

- How would you show that `assert` is indeed implemented as preprocessor macro?

# Program start & Command line arguments

## Program Startup

The program starts with the `main` function:

Two forms:

- **int** main ();
- **int** main (**int** argc, **char** * args []);

## Example

```
#include <stdio.h>
#include <stdlib.h>
int main (int argc, char * args[])
{
    printf ("Invoked with \%i arguments\n", argc);
    return EXIT_SUCCESS;
}
```

NOTE

- The second form receives a *pointer to an array of strings*.
- The number of strings in the array is indicated by the first parameter (integer) passed to the function.
- The first string is the *program name*.
- Returning from `main` ends the program, the return code is passed to the shell. exit (retcode) is the same as returning from main.

args.c: Demonstrate program name & args.

# Program Structure

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

#define D(a) a*2 // BAD!

int file_scope_variable;

// Function declaration
int myfunc (int i);

// Function definition
int myfunc (int i) { return 0; }

// There should be a single main
// function in your program
// (Can be in any .c file)
int main () { printf("%i",D(2+3));}
```

A program exists out of one or more `.c` files (and possibly libraries). Each `.c` file generally will have the following parts:

- `#include` statements
- Preprocessor macro definitions.
- *file scope* (global) variables
- Function declarations and definitions.
- Comments

Good code has:

- Descriptive variable names
- Relatively short functions
- No code duplication
- Useful comments

# Basic I/O

```c
#include <stdio.h>
int main ()
{
    FILE * f =
      fopen("filename.txt", "r");
    char c;
    while ((c=fgetc(f))!=EOF)
    {
        // Writes to stdout
        // same as: putc(c,stdout);
        putchar(c);
    }
    fclose (f);
}
```

- See manpage for stdio.h
- File stream concept; Predefined streams: stdin, stdout, stderr.
- Never use gets!
    - why not?
- Do not forget to close the file

Read a file character by character.

# Pointers

## Pointer

A pointer is any variable which holds (as value) a memory address. Like all variables, pointer variables have a static compile-time data type and a runtime value.

```
// Value = 0
// Type = integer
int myvariable = 0;

// Value = address of a
// Type = pointer to int
int * myvariable = &a;

// Change value of myvariable
*myvariable=10;
```

- Like all variables, pointer variables have a value (an address) and a type.
- Addresses are always the same size (for a given platform and compiler), so all pointer variables (*even of different type*) are the same size.
- Pointer variables with different types are compatible (can be assigned to one another)
  - However, this is dangerous and should be avoided!

`pointer-type.c`: address-of, dereference, pointers and struct/union, dangling pointer, nullpointer.

# Pointers

## Pointer

A pointer is any variable which holds (as value) a memory address. Like all variables, pointer variables have a static compile-time data type and a runtime value.

```
// Value = 0
// Type = integer
int myvariable = 0;

// Value = address of a
// Type = pointer to int
int * myvariable = &a;

// Change value of myvariable
*myvariable=10;
```

- operator & returns the address of a variable (and the type of the expression will be pointer-to-type-of-variable)
- operator * (dereference) takes a pointer-to-sometype and returns an *lvalue* of type ‘sometype’.
- operator −> is shorthand for * followed by . (member selector).

`pointer-type.c`: address-of, dereference, pointers and struct/union, dangling pointer, nullpointer.

# Pointer Arithmetic

## Operations on the pointer *datatype*

Pointers form a family of datatypes (i.e. **int** $*$, **int** $**$, **char** $*$, ...)
All members of this family support a common set of operations (just like all integer-types support + and −.

```
// a now points to someint
int * a = &someint;

// b now points to the
// address FOLLOWING
// someint (no matter if there
// is an integer
// there or not)
int * b = a+1;
```

- Pointers support addition, subtraction, increment, decrement.
- Pointer arithmetic is in multiples of the type pointed to.
  - ++a increments the value (address) by **sizeof**(*a), not by 1!
  - a−b (with a and b pointers) return the difference between the addresses of a and b in units of sizeof(*a)

Demonstrate pointer + pointer, pointer - pointer, pointer + int, pointer - int, pointer  int, ...

# **void** pointers

```
int b;
void * a = &b;
*a = 10; // illegal
int * c = a;
*c = 10; // OK
*(int *)a = 10; // OK
```

- **void** pointers cannot be dereferenced (**void** is 'no type')
- To dereference them, provide type information by
  - Assigning them to non-void pointer.
  - *casting* (i.e. override deduced type) to a typed pointer.

# Pointers and Arrays

```c
char test[] = "test";
somefunctions(test); // decay
test[0]='a';         // decay

// no decay
somefunc(sizeof(test));
```

- Arrays are not really a first-class data type in C
- Arrays almost always 'decay' into pointers; Exceptions:
  - Argument of operator &
  - Argument of operator **sizeof**
  - Argument of operator `alignof` (C11)
  - Use as string literal

NOTE

Even the subscript operator ( [] ) is not what it seems:

a[2] => *(a+2)

# Function: overview

```c
// function prototype(declaration)
int testfunc ();

// function definition
int testfunc ()
{
    // declarations and
    // statements
}

// Function taking 1 argument
// returning nothing
void returnnothing (int arg);
```

- A function starts a new scope: variables and declarations are local to the function.
- The scope of a variable in a function is until the end of the function. Same for function parameters.
- The (default) lifetime of a variable in a function starts from the moment it is defined until the function returns (same for function parameters).
- Each function invocation has its own set of local variables (allows recursion).
- All functions share the same namespace (can't have two functions with the same name *in the same program*)

## Pass-by-value or pass-by-reference?

pass-by-value The value of the variable is copied to the function; The function operates on a *copy* of the variable.

pass-by-reference The variable in the function is an *alias* for the variable passed to the function. The function directly operates on the passed variable.

pass-by-object-reference (java, python) Pass-by-value but objects are references (which are passed by value).

In C, function calls are **pass-by-value**.

# Returning Values

```c
// Grammar:
// return <expression> ;
//
int myfunc ()
{
    // return takes int
    return 10;
}

void myfunc2 ()
{
    return;
}
```

- Functions can return any datatype (including structs) but not arrays.
- The return value is *copied*; Be careful with returning large structures.
- **return** optionally returns a value and always returns to the caller immediately (further statements are skipped).
- **return** is optional in functions returning void.

# Emulating Pass-By-Reference

```c
// function
void example(int * p)
{
    *p=0;
}

int a = 1;
example(&a);
// prints 0
printf("%i\n", a);
```

## Question

Since function parameters are always pass-by-value, how do we *modify* a variable from within the function?

Solution: pass a *pointer* to the variable.

- The pointer is passed by value (which is OK)
- The function can use dereference the pointer to modify the variable.

# Pointers to functions

```c
// function prototype:
// int myfunc();

// & is optional
// (but good practice)
int (*funcptr)() = &myfunc;
```

- Pointers can point to functions as well.
- Dereferencing is optional for function pointers.
- Pointer arithmetic accepted but probably wrong.

qsort.c: qsort() function and example.

# Decoding Declarations

Study the signal function.

# Valgrind

## Valgrind

*Valgrind is an instrumentation framework for building dynamic analysis tools. There are Valgrind tools that can automatically detect many memory management and threading bugs, and profile your programs in detail. You can also use Valgrind to build new tools.*

- Valgrind can help debug pointer issues
  (null pointer usage, accessing memory outside of allocated memory region, using memory that has been freed, memory leaks, ...)
- Homework 3 will use valgrind to validate your code.
- Valgrind is available on linux.cs.uchicago.edu.
- **Make sure to compile your code with debug information enabled! (`-g` option for gcc)**

demo memory leak, out-of-bounds, use-after-free

- General documentation: `http://valgrind.org/`
- Memory-debugging: `http://valgrind.org/docs/manual/quick-start.html#quick-start.mcrun`

# Valgrind Example

## Example

```
LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 0 bytes in 0 blocks
        suppressed: 88 bytes in 1 blocks
```

- Useful options for memory debugging:
  - `--show-reachable=yes`
  - `--leak-check=full`
- Valgrind cannot detect all errors or mistakes!
- Normally you can ignore 'suppressed' errors or leaks.

See `incorrect_program.c` in lecture materials

# GDB

## GDB

GDB is a command-line debugger (what's a debugger?)

- GDB is installed on linux.cs.uchicago.edu.
- GDB allows us to run the program step-by-step and to inspect the value of variables.
- GDB can do post-mortem analysis (i.e. on a coredump file).

NOTE

Add −ggdb to your compile commandline to instruct the compiler to include more information. Adding −O0 is also recommended.

setting breakpoints, inspecting variables, use a coredump.

- https://www.gnu.org/software/gdb/documentation

# GDB

## Core dumps

```
dries@linux2:~$ ulimit -c
0
dries@linux2:~$ ./incorrect_program
Segmentation fault
dries@linux2:~$ ulimit -c unlimited
dries@linux2:~$ ./incorrect_program
Segmentation fault (core dumped)
dries@linux2:~$ ls -l core
-rw------- 1 dries dries 253952 ...
dries@linux2:~$
```

- A core dump is a snapshot of the memory state of the program at the moment an unrecoverable error occurs
- It can be used afterwards to observe the state of the program (variables, which function, etc.)
- Normally, code dumps are disabled. Enable them using the `ulimit` command.

check & change `ulimit`, cause core, open gdb with core (`incorrect_program.c`)

## Homework

**Reminder**

From this point on, points will be subtracted for *memory leaks* and *warnings* in your code!

See `https://mit.cs.uchicago.edu/mpcs51040-spr-17/mpcs51040-spr-17/raw/master/homework/hw3/hw3.pdf`

## Reading Assignment

- The C Programming Language: Ch 5
  (At this point, every chapter (except for CH8) covers some or all of what we've discussed in class so far)