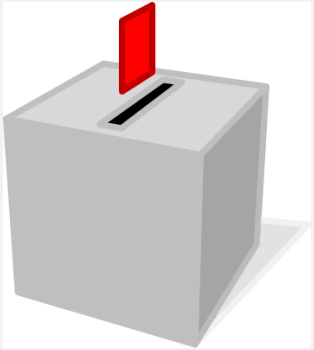
MPCS 51040 – C Programming

Lecture 2

Dries Kimpe

April 3, 2017

Poll Results

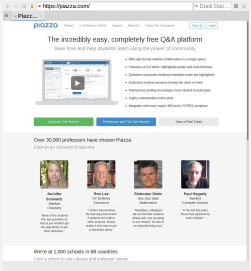


Preliminary Results ($\approx$ 50% response)

- Most people are new to C
- Preference for more assignments ($>$ 90%)
  - Including final project ($>$ 90%)
  - including optional assignments for extra credit
  - Note: total quiz weight remains the same!
- Topics:
  - Memory allocation
  - Operating Systems
- Majority is interested in a career as software developer

Still possible to provide feedback:
`https://goo.gl/forms/15yZxdUxJwjJdlIg2`

Piazza & GitLab



Piazza

- Everybody should have access to piazza
  (if not, contact me (via email) immediately)
- Collaboration encouraged
  (other than for homework etc.)
  - Don't be too eager to mark posts private in piazza – might get help sooner if public!

GitLab

You should have access to two repositories on (mit.cs.uchicago.edu)

Class `https://mit.cs.uchicago.edu/mpcs51040-spr-17/mpcs51040-spr-17/`

Personal `https://mit.cs.uchicago.edu/mpcs51040-spr-17/cnetid`

CSIL Mini-courses

CSIL (Computer Science Instructional Laboratory)

- Organizes mini-courses several times a year
- Highly recommended, in particular the courses about unix and git
- Notes available:

  git `https://csil.cs.uchicago.edu/minicourses/Git.html`

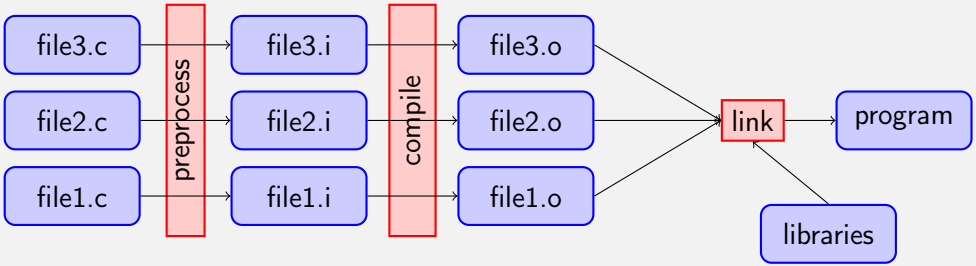
  unix `https://csil.cs.uchicago.edu/minicourses/linux.html`

`https://csil.cs.uchicago.edu/minicourses.html`

Unfortunately not scheduled this quarter...

From code to executable...

A processor can only execute a fixed set of binary instructions. How do we get from a set of files containing C code to an executable?



Every file is preprocessed and compiled *separately*

Preprocessor

Recap

Preprocessor?

- Completely different "language"
- Independent from C
- Its output is fed into compiler
- *text* manipulation language

What is it used for?

- Implement "module" concept
- Reduce repetition
- Conditional compilation
- Adapt to build time environment

- The preprocessor *does not understand C!*
- Avoid side-effects (unpure functions)
- Precedence
- Preprocessor (and C) is *case-sensitive*
- ...

Invoke using cpp or `gcc -E` (latter only for `.c` extension).
Use `-std=c11 -pedantic` for strict compliance.

Preprocessor

Emulating "modules"

Consider your program `mine.c`:

```
1 #include <stdlib.h>
2 ...
3 puts("Hello World!");
4 return EXIT_SUCCESS;
```

Output of `gcc -E`:

```
1 ... // thousands of lines of code
2 extern int puts (const char *__s);
3 ...// thousands of lines of code
4 puts("Hello World!");
5 return 0;
```

- Including the header (module?) `stdlib.h` makes extra functionality available, such as the predefined macro `EXIT_SUCCESS`
- This is similar to other languages: `import` (python/java/go)
- Unlike other languages, C does not have native support for the concept.
  - Instead it is 'emulated' by the preprocessor
  - The compiler can't tell the difference between code in `stdio.h` and code in your file (`mine.c`).

Examine the preprocessed source for your `helloworld.c`

Preprocessor

Avoiding repetition

```
1  int intmax(int a, int b)
2  { return (a > b ? a : b); }
3
4  double doublemax(double a, double b)
5  { return (a > b ? a : b); }
6
7  // ...
8
9  // scripts/kconfig/lxdialog/dialog.h:
10 #define MAX(x,y) (x > y ? x : y)
```

There are many sources of code repetition:

- language itself: strict/strong typing

Preprocessor

Avoiding repetition

From `net/bluetooth/rfcomm/core.c`:

```
1 #define __get_rpn_parity(line) \
2         (((line) >> 3) & 0x7)
```

There are many sources of code repetition:

- inherent to the problem being solved

Preprocessor

Avoiding repetition

(drivers/gpu/drm/amd/include/asic_reg/bif/bif_5_1_sh_mask.h)

```
1  ...
2  #define BUS_CNTL__BIOS_ROM_WRT_EN__SHIFT 0x0
3  #define BUS_CNTL__BIOS_ROM_DIS_MASK 0x2
4  #define BUS_CNTL__BIOS_ROM_DIS__SHIFT 0x1
5  #define BUS_CNTL__PMI_IO_DIS_MASK 0x4
6  #define BUS_CNTL__PMI_IO_DIS__SHIFT 0x2
7  #define BUS_CNTL__PMI_MEM_DIS_MASK 0x8
8  #define BUS_CNTL__PMI_MEM_DIS__SHIFT 0x3
9  #define BUS_CNTL__PMI_BM_DIS_MASK 0x10
10 #define BUS_CNTL__PMI_BM_DIS__SHIFT 0x4
11 #define BUS_CNTL__PMI_INT_DIS_MASK 0x20
12 #define BUS_CNTL__PMI_INT_DIS__SHIFT 0x5
13 #define MM_INDEX__MM_OFFSET_MASK 0x7fffffff
14 #define MM_INDEX__MM_OFFSET__SHIFT 0x0
15 #define MM_INDEX__MM_APER_MASK 0x80000000
16 #define MM_INDEX__MM_APER__SHIFT 0x1f
17 #define MM_DATA__MM_DATA_MASK 0xffffffff
18 #define MM_DATA__MM_DATA__SHIFT 0x0
19 ...
```

There are many sources of code repetition:

- Constants

Preprocessor

Conditional Compilation

Example from `kernel/sched/core.c`:

```
1  static void ttwu_do_activate(struct rq *rq,
2           struct task_struct *p,
3           int wake_flags,
4           struct pin_cookie cookie)
5  {
6           int en_flags = ENQUEUE_WAKEUP;
7
8           lockdep_assert_held(&rq->lock);
9
10 #ifdef CONFIG_SMP
11          if (p->sched_contributes_to_load)
12                  rq->nr_uninterruptible--;
13
14          if (wake_flags & WF_MIGRATED)
15                  en_flags |= ENQUEUE_MIGRATED;
16 #endif
17
18          ttwu_activate(rq, p, en_flags);
19          ttwu_do_wakeup(rq, p, wake_flags, cookie);
20 }
```

The example on the left shows a fragment of linux scheduler code:

- It is highly performance critical and configurable
  (from embedded systems with one CPU and little memory to systems with hundreds of hyperhtreaded-cores)
- Why not a simple if test?
  - Run time overhead (computation **and** memory)
- Preprocessor heavily relied on in linux kernel:
  - Approximately 2,000,000 preprocessor directives (starting with #) in source (for 16,363,350 lines of code; $\approx 12\%$)

Examine

Preprocessor

Adjust to build environment

Example from `kernel/sched/core.c`:

```
1  /*
2   * See if our compiler is known to
3   * support flexible array members.
4   */
5  #if defined(__STDC_VERSION__) && \
6          (__STDC_VERSION__ >= 199901L)
7  # define FLEX_ARRAY /* empty */
8  #elif defined(__GNUC__)
9  # if (__GNUC__ >= 3)
10 #  define FLEX_ARRAY /* empty */
11 # else /* older GNU extension */
12 #  define FLEX_ARRAY 0
13 # endif
14 #endif
15
16 /*
17  * Otherwise, default to safer but a bit wasteful traditional style
18  */
19 #ifndef FLEX_ARRAY
20 # define FLEX_ARRAY 1
21 #endif
```

The example on the left shows a fragment of `tools/perf/util/util.h` (again linux kernel)

- The code tries to determine if a certain non-standard C extension is available

Examine example file

Preprocessor overview

Processing a document: step 1

```
1  // Continuation: \ at end of
2  // line (no space!)
3  te\
4  st // replaced by test
5
6  // single line comment
7  // Comment: replaced by space
8  There will be a space/*h*/here
9
10 /* multi-line comment
11 can span multiple
12 lines
13 */
14
15 // Prove the claim below
16 "/*not a comment*/"
```

- Read input stream (file)
- Perform some simple transformations
  - split into lines
  - merge continuation lines
  - replace comments by single space
  - substitute trigraphs
    (see next slide)

preprocessor −step1.c

Trigraphs & digraphs : example

```
1  ??=include <stdio.h>
2  ??=include <stdlib.h>
3  int main (int argc, char ** args)
4  ??<
5      puts("Hello World!");
6      return EXIT_SUCCESS;
7  ??>
```

trigraphs	replacement
??(	[
??)	]
??<	{
??>	}
??=	#
??/	\
??'	^
??!	|
??-	~

See trigraphs.c

Trigraphs are disabled in some compilers, unless strict compliance is requested!
For gcc/cpp, use `-trigraphs` or `-std=c11` (recommended)

Preprocessor

Step 2: Tokenization

```
1  A+B     // 3 tokens: A, +, B
2  "test test2" // one token
3
4  // Continuation: \ at end of
5  // line (no space!)
6  te\
7  st // one token: test
8
9  23++x19 // 3 tokens: 23 ++ x19
10 <::> // 2 tokens (2x digraph)
```

digraph	equivalent
<%	{
%>	}
<:	[
:>	]
%:	#
%:%:	##

- Tokenization is the splitting of the input stream into *tokens*; A token is a sequence of characters.
- Certain characters/sequences separate tokens from each other.
  - White space (space, newline, ...)
    Whitespace itself does not result in a token
  - Characters such as '+', '-', '*', ...
    These mostly correspond to C operators
- Different kinds of tokens:

identifier	sequence of letters, digits, underscore which starts with a letter or underscore.
number	23 23.1202 .99e+10 1.23E-100
string literal	`"bla"`, `'c'`, <stdio.h> (only for **#include**)
punctuators	++, −, [, ... including *digraphs*
other	Any other character

See https://gcc.gnu.org/onlinedocs/cpp/Tokenization.html

Preprocessor

String Literals: Character Escape

Escape sequence	Description	Representation
\'	single quote	byte 0x27 in ASCII encoding
\"	double quote	byte 0x22 in ASCII encoding
\?	question mark	byte 0x3f in ASCII encoding
\\	backslash	byte 0x5c in ASCII encoding
\a	audible bell	byte 0x07 in ASCII encoding
\b	backspace	byte 0x08 in ASCII encoding
\f	form feed - new page	byte 0x0c in ASCII encoding
\n	line feed - new line	byte 0x0a in ASCII encoding
\r	carriage return	byte 0x0d in ASCII encoding
\t	horizontal tab	byte 0x09 in ASCII encoding
\v	vertical tab	byte 0x0b in ASCII encoding
\nnn	arbitrary octal value	byte nnn
\xnn	arbitrary hexadecimal value	byte nn
\unnnn	universal character name (arbitrary Unicode value); may result in several characters	code point U+nnnn
\Unnnnnnnn	universal character name (arbitrary Unicode value); may result in several characters	code point U+nnnnnnnn

Character escape

- Goal: Allow special characters within string
- A ‘\’ before a character (within a string) gives or prevents special meaning of following character.

Example

```
1  puts("a quote \" in a string");
2  puts("a newline(return)"
3       "here ->\n in a string");
```

‘\0’ and ‘\n’ most common...

Preprocessor

Step 3: Preprocessor directives and macros

```
1  // Inclusion of files
2  #include <stdio.h>
   // system headers
3  #include "myheader.h" // program headers
4
5  // Macro substitution
6  #define identifier replacement tokens
7
8  #define identifier(param1,param2) \
9    replacement tokens param1 param2
10
11 // Conditional compilation
12 // (other conditionals exist)
13 #ifdef identifier
14  ...
15 #else
16  ...
17 #endif
```

Inclusion

- Difference in *search path*
- Argument treated as string (no macros or comments) but also no escape (\\)

Macros

- Parameters separated by ‘,’
  - Parenthesis (only), i.e. () must balance
- Arguments expanded first
- parameters substituted in output
- . . . followed by rescanning for macros

In-depth information (recommended): https://gcc.gnu.org/onlinedocs/cpp/

`#define, #ifdef, #include, max problem (l2_cpp_1.c,l2_cpp_2.c, l2_cpp_3.c)`

Use of the preprocessor in C

Using the preprocessor for headers

Code reuse (modules/library concept) in C is implemented by the preprocessor. This means the C language itself has no such concept!

By *convention*, code fragments that can be 'shared' (reused) are placed in a *header file*. **But, beware of duplicate definitions!**

```
1 #ifndef MYHEADER_H
2 #define MYHEADER_H
3
4 // header content
5
6 #endif
```

Issues

- Problem with unique names
- Preprocessor still opening all files (can be slow)

duplicate problem, solution.
(consider `stdio.h` as an example)

#pragma once

For completeness...

A.12.8 Pragmas

A control line of the form "`# pragma token-sequence opt`" causes the preprocessor to perform an implementation-dependent action. An unrecognized pragma is ignored.

```
1 // Header myheader.h
2 #pragma once
3
4 // normal header content
```

(any method is fine for homework)

Not officially part of C11 – compiler dependent Combination of techniques is possible.

Demonstrate that `#pragma once` is or is not working for your compiler (and corresponding preprocessor)

See header_protect.c (don't need to understand all code at this point)

Compilation

Transform C code into *object files* (machinecode + information for the linker).

Many different compilers exist for the same language.

- Different target systems
- Each typically has its own *dialect*
- Each supports some set of *standards* in varying degrees
- Warnings are not part of the standard and will differ between compilers.

We will be using gcc. You can use any C11 compatible compiler, however grading happens using the compiler installed on linux.cs.uchicago.edu.

- For this class, **WARNINGS ARE CONSIDERED ERRORS**.
- Make sure to select the correct standard and warning level! (`-Wall -Werror -pedantic -std=c11` for gcc)

```
gcc -std=c11 -Wall -Werror -pedantic
```

(if you don't want to link, but just preprocess and compile, using `-S`)

- Last public draft: `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`

Linking

The linker takes a set of object files (and libraries or archives) and combines them into a single executable program.

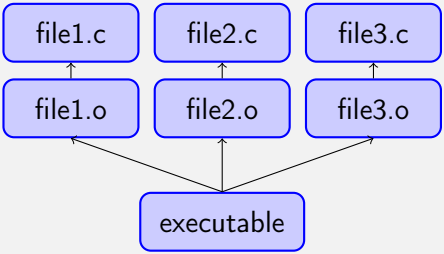
- Normally, as is the case with the preprocessor, the C compiler takes care of invoking the linker
  (adds system dependent runtime libraries)
- Most standard required C functions are provided by a library which is automatically linked in by the compiler.
  (For others, example the math functions, you need to explicitly link the library.)

For now, don't worry about the details.

Useful tools: `objdump`, `nm`. See manpages!

- Compile to object files `-c`
- Examine object files using `nm` or `objdump`

Makefile



Makefile.example

- Automates steps to build executable
- Set of recipes to create(build) files
- Considers create/last modification time of *dependencies*
- Builds *dependency tree*
- "making" a node is done by recursively making the node's dependencies
  - execute rule if *target* (i.e. the node) is *older* than any of its dependencies

- Simple make tutorial: `http://mrbook.org/blog/tutorials/make/`
- GNU make manual: `http://www.gnu.org/software/make/manual/make.html`

Expected to be able to write and understand simple makefiles!

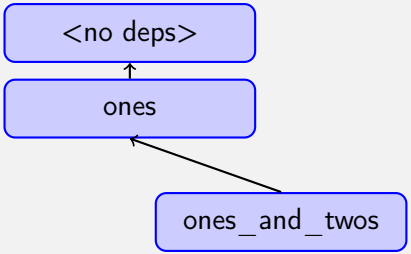
Single-slide introduction to make

```
# example makefile: execute with
# make -f example.make targetname

ones:
     echo 11111 > ones

# first target, so default
ones_and_twos: ones
    cat ones > ones_and_twos
    echo Hello!
    echo 22222 > ones_and_twos
```

Make is a tool to create and update files. . .
(Note: does not mention C, programming, compiling, . . .



Single-slide introduction to make

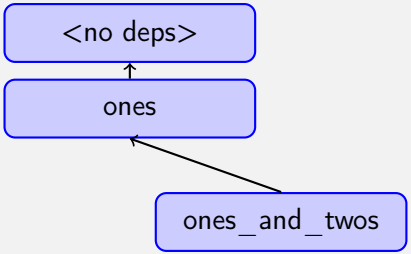
```
# example makefile: execute with
# make -f example.make targetname

ones:
	echo 11111 > ones

# first target, so default
ones_and_twos: ones
	cat ones > ones_and_twos
	echo Hello!
	echo 22222 > ones_and_twos
```

Target:

- The name of the file to create or update
- The target(s) are specified on the make command line
- The default target (if none is specified) is the *first* target encountered in the makefile.

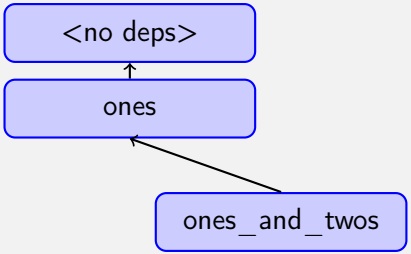


Single-slide introduction to make

```
# example makefile: execute with
# make -f example.make targetname

ones:
     echo 11111 > ones

# first target, so default
ones_and_twos: ones
    cat ones > ones_and_twos
    echo Hello!
    echo 22222 > ones_and_twos
```



Dependencies (or prerequisites):

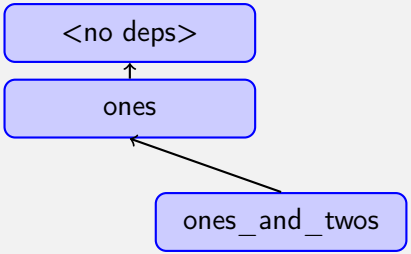
- names of *files* that are needed to 'make' (create) the target
- The dependencies also help make to determine when a file needs to be updated. Rebuild or update if any of the following is true:
  - the *target* doesn't exist
  - the *target* is *older* (considering last modified file time) than any of its dependencies
- After rebuilding a target, the dependencies are re-examined.

Single-slide introduction to make

```
# example makefile: execute with
# make -f example.make targetname

ones:
	echo 11111 > ones

# first target, so default
ones_and_twos: ones
	cat ones > ones_and_twos
	echo Hello!
	echo 22222 > ones_and_twos
```



Recipe: *how* to (re)build a target

- Every line is executed by the shell
- A recipe line is recognized by a leading <TAB>
  - Spaces are not accepted!

numberfiles.make

Manpages

Most unix/linux systems come with built-in documentation for shell commands as well as most C library functions.

- Question: what about name conflicts?
  (find the answer in `man man`)
- Pay attention to the *restrictions/conditions* and standard information (CONFORMING TO)

manpage for `ssh`, `printf`, `strdup` and `sin`, `stdio.h`, `stdlib.h`, `string.h`.

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.

```
1 struct tagname
2 {
3     members
4 }
```

Union members *share* the same memory space.

```
1 union tagname
2 {
3     members
4 }
```

Example

```
1 struct coordinate
2 {
3     int x;
4     int y;
5 } coordinate_var;
6 struct coordinate var2;
```

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

```
1 int x[10]; // Array of 10 ints
2 unsigned int size=2;
3 int y[size]; // VLA
4 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

```
1  if ((23 + 4) > 10) printf ("OK");
2
3  if (a)
4    if (b)
5      callfunc();
6    else
7      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

```
1  switch(a)
2  {
3      case 0:
4          printf("Zero");
5          break;
6      case 1:
7          printf("One");
8          break;
9      default:
10         printf("Many");
11 }
```

- 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) ;

```
1  while (x<10)
2  { ++x; }
3
4  do { ++x; } while (x<10);
5
6  while (x<10)
7  {
8      if (++x==3) continue;
9      printf("num");
10 }
```

- 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

```
1  for (int i=0; i<100; ++i)
2    callfunc();
3
4  for ( ; ; )
5  { 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

```
1  // THis is a single line comment
2
3  /*
4 This is a multi-line comment.
5 It ends with: */
```

Multi-line comments do not nest

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

Assertions

Example

```
1 #include <assert.h>
2
3 int divide (int x, int y)
4 {
5     assert (0!=y);
6 }
```

Example

```
1 #include <assert.h>
2 static_assert(
3     sizeof(long long)>=8,
4     "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

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main (int argc, char * args[])
4  {
5      printf ("Invoked with \%i arguments\n", argc);
6      return EXIT_SUCCESS;
7  }
```

- 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

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  #define D(a) a*2 // BAD!
5
6  int file_scope_variable;
7
8  // Function declaration
9  int myfunc (int i);
10
11 // Function definition
12 int myfunc (int i) { return 0; }
13
14 // There should be a single main
15 // function in your program
16 // (Can be in any .c file)
17 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.

Next week

- To commit **before class**: Homework 2
- Reading assignment:
  - Chapter 4
  - pointers & Arrays (Ch 5; section 1.6)

Homework

Homework 1 review

`https://mit.cs.uchicago.edu/mpcs51040-spr-17/mpcs51040-spr-17/raw/master/homework/hw1/hw1.pdf`

- Any issues?

Homework 2

`https://mit.cs.uchicago.edu/mpcs51040-spr-17/mpcs51040-spr-17/raw/master/homework/hw2/hw2.pdf`

- Actual C programming...
- Makefile !
- Due next week *before* class...