

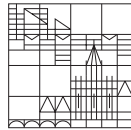
Introduction to the C Programming Language

Accompanying Tutorial to Operating Systems Course

Alexander Holupirek, Stefan Klinger

Database and Information Systems Group
Department of Computer & Information Science
University of Konstanz

Summer Term 2009



- ▶ Quick introduction
- ▶ Show essential elements of the language
- ▶ No details, rules, and exceptions
- ▶ Provide examples
- ▶ Show the basics, such as
 - ▶ variables and constants
 - ▶ arithmetic
 - ▶ control flow
 - ▶ functions
 - ▶ rudiments of input and output
- ▶ Some of the (important) differences to JAVA
 - ▶ memory layout and the consequences
 - ▶ memory allocation
 - ▶ pointers
 - ▶ structures

1

The First Program Is Always The Same

Print the words: "Hello, world"

Not that easy, because you have to:

- ▶ Create the program text
- ▶ Compile it successfully
- ▶ Run it
- ▶ Get the output

```
1 #include <stdio.h>
2
3 int
4 main(void)
5 {
6     printf("Hello ,_world\n");
7     return (0);
8 }
```

Compilation On A UNIX-like OS

```
$ cc -Wall hello.c
$ ls
hello.c a.out
$ ./a.out
Hello, world
$
```

engine	filename	description
	hello.c	source code
preprocessor	hello.i	source w/ preproc. directives expanded
compiler	hello.s	assembler code
assembler	hello.o	object code ready to be linked
linker	a.out	executable

3

2

4

From Source Code To Executable Code

Storage Classes

The C Preprocessor

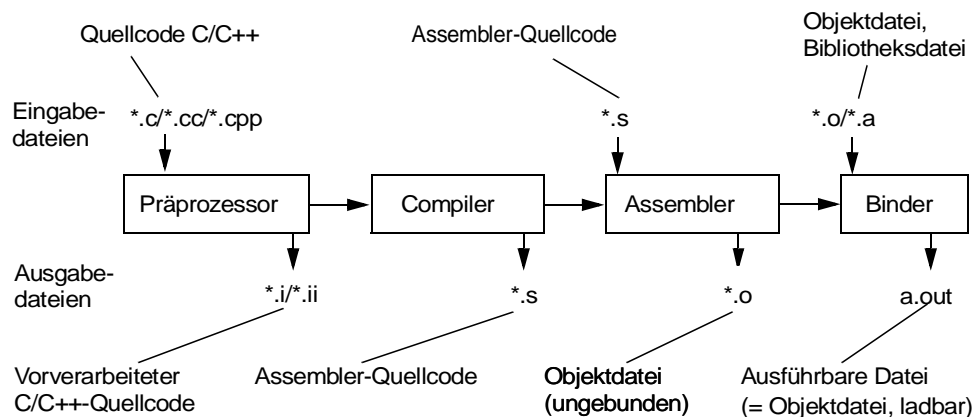
Variables and Arithmetic Expressions

Character Input and Output

Arrays

5

Translation Steps Using gcc(1)



7

From Source Code To Executable Code

Translation Steps (multi-phase compilation)

Compilation HLL source code to assembler source code

Assembly Assembler source code to object code

Linking Object code to executable code

Compilers and **assemblers** create object files containing the generated binary code and data for a source file. **Linkers** combine multiple object files into one, **loaders** take object files and load them into memory.

Goal: An executable binary file (a.out)

From high-level language (HLL) *source code* to *executable code*, i.e., concrete processor instructions in combination with data.

6

File Suffixes And Their Meaning

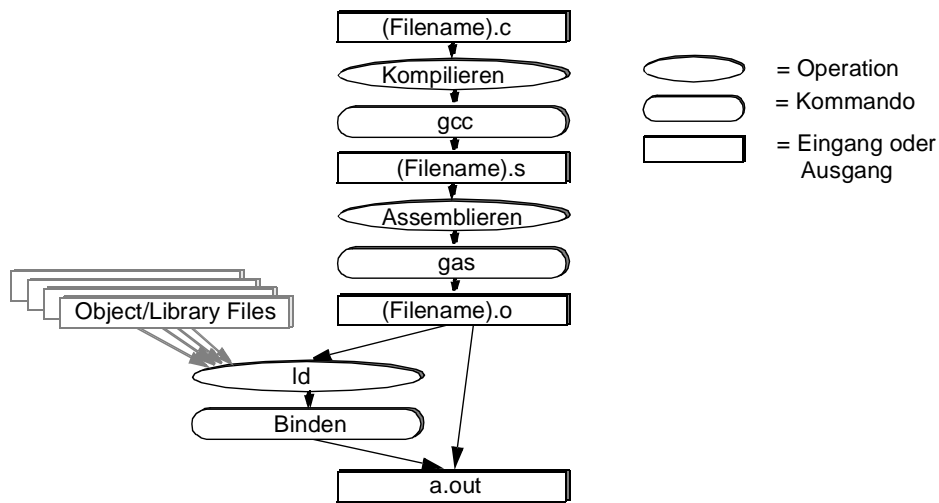
For any given input file, the file name suffix determines what kind of compilation is done (see gcc(1) for more details and suffixes):

suffix	compilation step
.c	C source code which must be preprocessed
.i	C source code which should not be preprocessed
.h	Header file to be turned into a precompiled header
.s	Assembler code
.o	An object file to be fed straight into linking

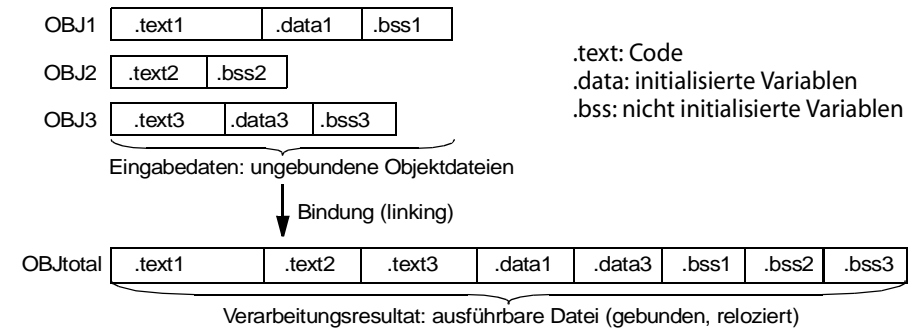
- ▶ Any of this steps can be done on its own.
- ▶ You may want to try and examine the outputs.

8

Creation Of An Executable File



Linking An Executable Binary



- ▶ Each object code (compiled separately) starts at address 0
- ▶ Linking them together involves
 - ▶ centralization of sections
 - ▶ relocation of addresses

9

10

Typical Program Organisation

A typical program divides naturally in sections

Code machine instructions, should be unmodifiable, size is known after compilation, does not change (.text)

- Data**
- ▶ constants
 - ▶ often placed in read-only .text
 - ▶ static data
 - ▶ initialized (.data) / uninitialized (.bss)
 - ▶ constant address in memory
 - ▶ permanent life time
 - ▶ dynamic data
 - ▶ stack or heap
 - ▶ storage space not known
 - ▶ volatile life time

Virtual Memory And Segments

Virtual Memory

- ▶ Whenever a process is created, the kernel provides a chunk of physical memory which can be located anywhere
- ▶ Through the magic of virtual memory (VM), the process believes it has all the memory on the computer

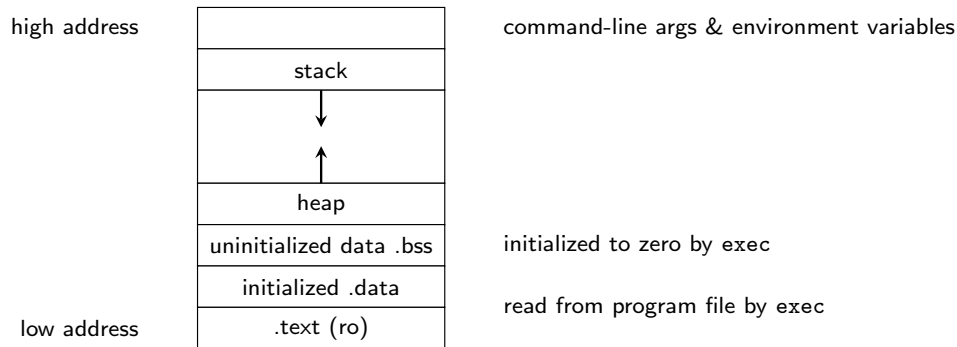
Typically the VM space is laid out in a similar manner:

- ▶ Text Segment (.text)
- ▶ Initialized Data Segment (.data)
- ▶ Uninitialized Data Segment (.bss)
- ▶ The Stack
- ▶ The Heap

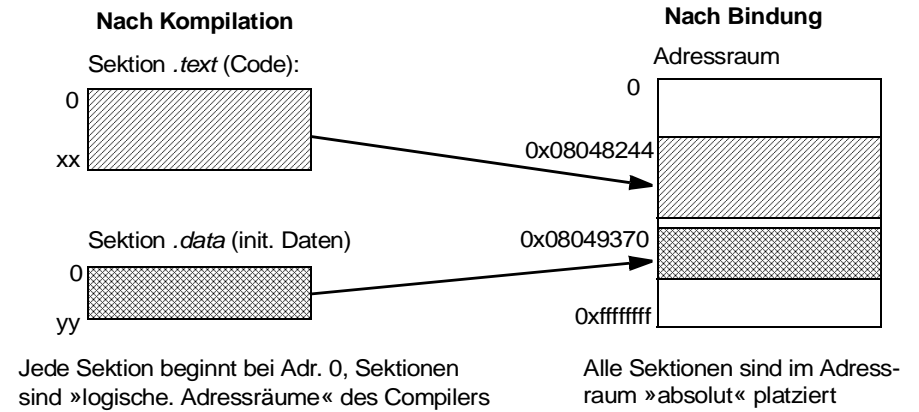
11

12

Memory Layout Running Process



Program Section In Virtual Memory



13

14

C Programs In (Address) Space And (Run-)time

Where is my data and why do I have to know?

- ▶ C is closely related to the machine. Before talking about pointers, storage allocation etc. some background knowledge about address space, (virtual) memory and its allocation during program execution comes in handy.
- ▶ Knowledge about the memory layout of a program is quite helpful when debugging.
- ▶ Knowledge about what is happening inside the machine on program execution is fundamental, to both, debugging programs and, in first place, writing clean code.

From Source Code To Executable Code

Storage Classes

The C Preprocessor

Variables and Arithmetic Expressions

Character Input and Output

Arrays

15

16

Storage Classes

Placement of data in memory depends on storage class

- ▶ An object, such as a variable, is a location in storage, and its interpretation depends on two main attributes: its *storage class* and its *type*.
- ▶ The *storage class* determines the lifetime of the storage associated with the identified object.
- ▶ The *types* determine the meaning of the values found in the identified object.
- ▶ In C we have two storage classes: *automatic* and *static*.
- ▶ Storage class specifiers (`auto`, `extern`, `register`, `static`) together with the context of an object's declaration, specify its storage class.

Static Storage Class

Static Objects

- ▶ May be local to a block or external to all blocks.
- ▶ In both cases, they retain their values across exit from and reentry to functions and blocks.
- ▶ Within a block, static objects are declared with `static`.
- ▶ Objects declared outside of all blocks (at the same level as function definitions) are always static.
- ▶ On the outer level, the keyword `static` makes them local to a particular translation unit (*internal linkage*).
- ▶ They are global to an entire program by omitting an explicit storage class, or by using `extern` (*external linkage*).

Automatic Storage Class

Automatic Objects

- ▶ `auto` and `register` give the declared objects automatic storage class, and may be used only within functions.
- ▶ They are local to a block¹, discarded on exit from the block.
- ▶ Declarations within a block create automatic objects if no storage class specification is mentioned or `auto` is used.
- ▶ Initialization of automatic objects is performed each time the block is entered at the top (if a jump into the block is executed the initializations are not performed).
- ▶ Objects declared `register` are automatic, and are (if possible) stored in fast registers of the machine
- ▶ For `register` the address operator `'&'` is not allowed.

¹aka "compound statement", such as the body of a function

Storage Class And Sections

Intermediate Summary

- ▶ A program executed does not only use storage for its instructions, but additionally needs space for, e.g., variables
- ▶ Variables may be temporary, dynamically allocated, or static (*i.e.*, permanent in terms of storage allocation), initialized or uninitialized, declared as constant (`const`) and thus read-only
- ▶ Placement of data in memory depends on its storage class.
- ▶ During the translation process the compiler uses *sections* to divide the address space into logical units.
- ▶ Details vary with operating systems and compiler used.

Variable Placement And Life Time (Code)

```

int a;
static int b;

void
func(void)
{
    char c;
    static int d;
}

int
main(void)
{
    int e;

    int *pi = (int*) malloc(sizeof(int));
    func();
    func();
    free(pi);
    return (0);
}

```

21

Variable Placement And Life Time (Code)

```

int a; /* Permanent life time */
static int b; /* dito, but reduced scope */

void
func(void)
{
    char c; /* only for the life time of func() */
            /* but 2x; visible only in func() */
    static int d; /* i'm unique, exist once at a stable */
                /* address, visible only in func() */
}

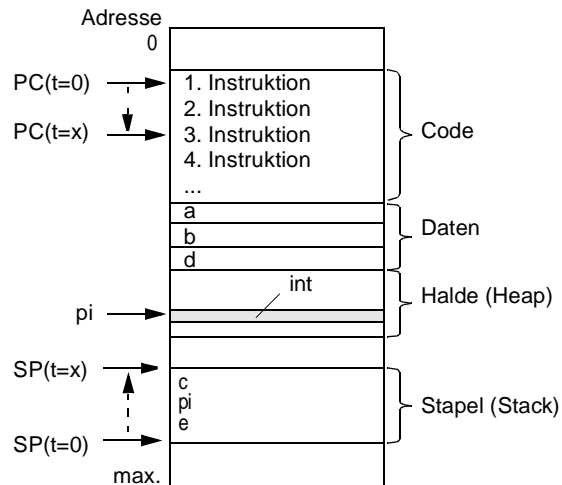
int
main(void)
{
    int e; /* life time of main() */

    int *pi = (int*) malloc(sizeof(int)); /* newborn */
    func();
    func();
    free(pi); /* RIP, pi points to an invalid address */
    return (0);
}

```

22

Variable Placement And Life Time (Diagram)



t=0: Programmausführung wird gestartet, d.h., Ausführungsumgebung ist bereits initialisiert

t=x: beliebiger Zeitpunkt während der Programmausführung

23

Variable Placement

Variables (outside a function) Globally declared variables go to the *Uninitialized Data Segment* if they are not initialized, to *Initialized Data Segment* otherwise. Necessary for the OS to decide if storage has to be *loaded* with initialization data from the executable binary.

Variables (inside a function) Implicit assumption of `auto`, go to *The Stack*. Declared as `static`, see above.

Constants (const) *Text Segment*

Function Parameters Are pushed on *The Stack* or stored in registers. If pointers are passed, data is elsewhere.

24

Memory Segments

Text Segment. The text segment contains the actual code (including constants) to be executed. It's usually sharable, so multiple instances of a program can share the text segment to lower memory requirements. This segment is usually marked read-only so a program can't modify its own instructions.

Initialized Data Segment. This segment contains global variables which are initialized by the programmer.

Uninitialized Data Segment. Also named `.bss` (block started by symbol) which was an operator used by an old assembler. This segment contains uninitialized global variables. All variables in this segment are initialized to 0 or NULL pointers before the program begins to execute.

Memory Segments (cont.)

The Stack The stack is a collection of stack frames which we will discuss later. When a new frame needs to be added (as a result of a newly called function), the stack grows downward.

The Heap Dynamic memory, where storage can be (de-)allocated via C's `free(3)/malloc(3)`. The C library also gets dynamic memory for its own personal workspace from the heap as well. As more memory is requested "on the fly", the heap grows upward.

25

26

From Source Code To Executable Code

Storage Classes

The C Preprocessor

Variables and Arithmetic Expressions

Character Input and Output

Arrays

The C Preprocessor

The C preprocessor performs ...

- ▶ Inclusion of named files
- ▶ Macro Substitution (→ consult your favourite C book)
- ▶ Conditional Compilation (→ consult your favourite C book)

27

28

File Inclusion

A control line of the form

```
#include filename
```

causes the replacement of that line by the entire contents of the file *filename*.

Note

The characters in the name *filename* must not include > or \n, and the effect is undefined if it contains any of ", ', \ , or /*.

Location

The named file is searched for in a sequence of implementation-dependent places (often starting in /usr/include).

Predefined Names

Several identifiers are predefined, and expand to produce special information. They, and also the preprocessor expression operator `defined`, may not be undefined or redefined.

<code>--LINE--</code>	A decimal constant containing the current source line number
<code>--FILE--</code>	A string literal containing the name of the file being compiled
<code>--DATE--</code>	A string literal containing the data of compilation 'Mmm dd yyyy'
<code>--TIME--</code>	A string literal containing the data of compilation 'hh:mm:ss'
<code>--STDC--</code>	The constant 1. It is intended that this identifier be defined to be 1 only in standard-conforming implementations

29

30

C Programs

Basic building blocks

- ▶ functions
- ▶ statements
- ▶ variables
- ▶ arguments

- ▶ *functions* contain *statements*
- ▶ *statements* specify computing operations to be done
- ▶ *variables* store values used during computation
- ▶ *arguments* (one way to) communicate data between functions

Building Blocks Of Our Example

- ▶ A function called `main`
- ▶ Liberty to name functions whatever you like, but ...
- ▶ `main` is special, as any program begins execution at the beginning of `main`
- ▶ Every program must have a `main` somewhere
- ▶ `main` will usually call other functions to help perform its job
 - ▶ Functions that you wrote
 - ▶ Functions that are provided for you, e.g. `printf`

31

32

Some Explanations About The Program Itself

```
1 #include <stdio.h>
2
3 int
4 main(void)
5 {
6     printf("Hello , world\n");
7     return (0);
8 }
```

- ▶ line 1: tell compiler to include information about the standard input/output library
- ▶ line 3/4: define a function named `main`, which receives no arg values. Parentheses after the function name surround the argument list. Returns an `int`.
- ▶ line 5/8: statements of `main` are enclosed in braces
- ▶ line 6: `main` calls library function `printf`, which prints this sequences of characters; `\n` represents the newline character.

Character String/String Constant

- ▶ A sequence of characters in double quotes is called a *character string* or *string constant*.
- ▶ Sequence `\n` stands for the *newline character*, which when printed advances the output to the left margin of the next line
- ▶ We have to use `\n` to include a newline character with `printf`

```
printf("Hello , world
");
```

```
$ cc hello.c
hello.c:6:16: missing terminating " character
hello.c:7:9: missing terminating " character
hello.c: In function 'main':
hello.c:8: error: syntax error before "return"
```

Line 6: Print A String

- ▶ A function is called by naming it, followed by a parenthesized list of arguments:
`printf("Hello world\n");`
calls the function `printf` with the argument
`"Hello world\n"`
- ▶ `printf` is a library function that prints output
(in this case the string of characters between the quotes)

33

Printing "Hello, world"

- ▶ `printf` never supplies a newline automatically
- ▶ So several calls can build up an output line in stages
- ▶ Our first program could just as well have been written like below to produce identical output:

```
#include <stdio.h>

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

35

34

36

Escape Sequences

- ▶ Notice that `\n` represents only a single character
- ▶ An *escape sequence* like `\n` provides a general and extensible mechanism for hard-to-type or invisible characters.

<code>\a</code>	alert (bell) character	<code>\\</code>	backslash
<code>\b</code>	backspace	<code>\?</code>	question mark
<code>\f</code>	formfeed	<code>\'</code>	single quote
<code>\n</code>	newline	<code>\"</code>	double quote
<code>\r</code>	carriage return	<code>\ooo</code>	octal number
<code>\t</code>	horizontal tab	<code>\xhh</code>	hexadecimal number
<code>\v</code>	vertical tab		

Table: The complete set of escape sequences

From Source Code To Executable Code

Storage Classes

The C Preprocessor

Variables and Arithmetic Expressions

Character Input and Output

Arrays

37

Fahrenheit-Celsius: $C = (5/9)(F - 32)$

```
1 #include <stdio.h>                                0      -17
2 /* print fahrenheit-celsius table                  20      -6
3    for fahrenheit = 0, 20, ..., 300 */             40      4
4 int                                                60      15
5 main(void)                                         80      26
6 {                                                  100     37
7     int fahr, celsius;                             120     48
8     int lower, upper, step;                        140     60
9                                                    160     71
10    lower = 0; /* lower limit */                    180     82
11    upper = 300; /* upper limit */                  200     93
12    step = 20; /* step size */                      220    104
13                                                    240    115
14    fahr = lower;                                   260    126
15    while (fahr <= upper) {                          280    137
16        celsius = 5 * (fahr - 32) / 9;              300    148
17        printf("%d\t%d\n", fahr, celsius);
18        fahr = fahr + step;
19    }
20    return (0);
21 }
```

38

Declarations And Assignment Statements

A *declaration* announces the properties of variables.
Consists of *type name* and a *list of variables*, such as:

```
7     int fahr, celsius;
8     int lower, upper, step;
```

Range/size of data types depends on machine

```
10    lower = 0; /* lower limit */
11    upper = 300; /* upper limit */
12    step = 20; /* step size */
```

Assignment statements set the variables to their initial values.

39

40

Basic Data Types

type	description
char	a single byte, capable of holding one character in the local character set
int	an integer, typically reflecting the natural size of integers on the host machine
float	single-precision floating point
double	double-precision floating point

short and long are qualifiers that can be applied to integers:

```
short int i;  
long int f;  
unsigned long d;
```

signed and unsigned can be applied to char and any integer.

The while Loop

Each line in the result table is computed the same way:

```
15     while (fahr <= upper) {  
16         celsius = 5 * (fahr - 32) / 9;  
17         printf("%d\t%d\n", fahr, celsius);  
18         fahr = fahr + step;  
19     }
```

Data Types And Sizes

Sizes are machine-dependent

- ▶ Each compiler is **free to choose appropriate sizes** for its own hardware. ISO C defines compile-time limits.
- ▶ short and int are *at least* 16 bit
- ▶ long is *at least* 32 bit
- ▶ short is no longer than int, int is no longer than long
- ▶ Numerical limits² are documented in <limits.h> and <stdint.h>³

²ISO C99 : 7.10/5.2.4.2 : Numerical limits

³ISO C99 : 7.18 : Integer Types

41

42

Integer Division

Why is $C = (5/9)(F - 32)$ computed as:

```
16     celsius = 5 * (fahr - 32) / 9;
```

As in many other languages, **integer division truncates**, i.e., any fractional part is discarded. Since 5 and 9 are integers, 5/9 would be truncated to zero and so all the Celsius temperature would be reported as zero.

43

44

printf(3) Revisited

```
#include <stdio.h>

int
printf(const char *format, ...);
```

printf(3) is a general-purpose output formatting function.⁴

- ▶ 1st argument is the string of characters to be printed.
 - ▶ Each % indicates **where** one of the other arguments
 - ▶ and **in what form** it is to be printed.
- ▶ Each % in the 1st arg is paired with the 2nd, 3rd arg etc.

```
17 printf("%d\t%d\n", fahr, celsius);
```

- ▶ %d, for instance, specifies an integer argument, so fahr and celsius are printed with a tab (\t) between them.

⁴Not part of the C language, but defined in ANSI X3.159-1989 ("ANSI C")

```
1 $ diff -up fahrenheit_v1.c fahrenheit_v2.c
2 --- fahrenheit_v1.c      Sat Apr 19 08:58:48 2008
3 +++ fahrenheit_v2.c      Sat Apr 19 08:58:05 2008
4 @@ -4,7 +4,7 @@
5  int
6  main(void)
7  {
8  -     int fahr, celsius;
9  +     float fahr, celsius;
10     int lower, upper, step;
11
12     lower = 0; /* lower limit */
13 @@ -13,8 +13,8 @@ main(void)
14
15     fahr = lower;
16     while (fahr <= upper) {
17 -         celsius = 5 * (fahr - 32) / 9;
18 -         printf("%d\t%d\n", fahr, celsius);
19 +         celsius = (5.0/9.0) * (fahr - 32.0);
20 +         printf("%3.0f\t%6.1f\n", fahr, celsius);
21         fahr = fahr + step;
22     }
23     return (0);
```

Fahrenheit-Celsius Converter Bug List

Fixing problems

- ▶ Pretty printing: Right-justified output
- ▶ Switch from integer to floating-point arithmetic

Construct a patch for the changes using diff(1)

```
NAME
    diff - compare files line by line

SYNOPSIS
    diff [OPTION]... FILES

DESCRIPTION
    Compare files line by line.
    -u  -U NUM  --unified[=NUM]
           Output NUM (default 3) lines of unified context.
    -p  --show-c-function
           Show which C function each change is in.
```

45

Patching The First Version

Applying the patch using patch(1)

```
$ ls
fahrenheit_v1.c  fahrenheit_v1_v2.diff

$ patch < fahrenheit_v1_v2.diff
Hmm... Looks like a unified diff to me...
The text leading up to this was:
-----
|$ diff -up fahrenheit_v1.c fahrenheit_v2.c
|--- fahrenheit_v1.c      Sat Apr 19 08:58:48 2008
|+++ fahrenheit_v2.c      Sat Apr 19 08:58:05 2008
-----
Patching file fahrenheit_v1.c using Plan A...
Hunk #1 succeeded at 4.
Hunk #2 succeeded at 13.
done

$ ls
fahrenheit_v1.c  fahrenheit_v1.c.orig  fahrenheit_v1_v2.diff
```

46

47

48

Fahrenheit-Celsius Converter v2

```

1 #include <stdio.h>
2 /* print fahrenheit-celsius table
3    for fahrenheit = 0, 20, ..., 300 */
4 int
5 main(void)
6 {
7     float fahr, celsius;
8     int lower, upper, step;
9
10    lower = 0; /* lower limit */
11    upper = 300; /* upper limit */
12    step = 20; /* step size */
13
14    fahr = lower;
15    while (fahr <= upper) {
16        celsius = (5.0/9.0) * (fahr - 32.0);
17        printf("%3.0f\t%6.1f\n", fahr, celsius);
18        fahr = fahr + step;
19    }
20    return (0);
21 }

```

0	-17.8
20	-6.7
40	4.4
60	15.6
80	26.7
100	37.8
120	48.9
140	60.0
160	71.1
180	82.2
200	93.3
220	104.4
240	115.6
...	...

Printing With printf(3)

specifier	print as ...
%d	decimal integer
%6d	decimal, at least 6 characters wide
%f	floating point
%6f	floating point, at least 6 characters wide
%.2f	floating point, 2 characters after decimal point
%6.2f	floating point, at least 6 wide and 2 after decimal point

- ▶ Further printf(3) recognizes %o for octal, %x for hexadecimal, %c for character, %s for string, %p for address (pointer)
- ▶ ISO C : 7.19.6 : Formatted input/output functions

49

50

The for Loop, Fahrenheit-Celsius v3

```

1 #include <stdio.h>
2 /* print fahrenheit-celsius table
3    for fahrenheit = 0, 20, ..., 300 */
4 int
5 main(void)
6 {
7     int fahr;
8
9     for (fahr = 0; fahr <= 300; fahr = fahr + 20)
10        printf("%3d\t%6.1f\n", fahr, (5.0/9.0)*(fahr-32));
11
12    return (0);
13 }

```

0	-17.8
20	-6.7
40	4.4
60	15.6
80	26.7
100	37.8
120	48.9
140	60.0
160	71.1
180	82.2
200	93.3
220	104.4
240	115.6
260	126.7
280	137.8
300	148.9

Symbolic Constants, Fahrenheit-Celsius Final

- ▶ Bad practice to bury “magic numbers” in a program
- ▶ Convey little information, hard to change in a systematic way
- ▶ A #define line defines a *symbolic name*

```

1 #include <stdio.h>
2
3 #define LOWER 0 /* lower limit of table */
4 #define UPPER 300 /* upper limit */
5 #define STEP 20 /* step size */
6
7 /* print fahrenheit-celsius table */
8 int
9 main(void)
10 {
11     int fahr;
12
13     for (fahr = LOWER; fahr <= UPPER; fahr += STEP)
14        printf("%3d\t%6.1f\n", fahr, (5.0/9.0)*(fahr-32));
15
16    return (0);
17 }

```

51

52

From Source Code To Executable Code

Storage Classes

The C Preprocessor

Variables and Arithmetic Expressions

Character Input and Output

Arrays

Processing character data

- ▶ Text I/O is dealt with as streams of characters
- ▶ A *text stream* is a sequence of characters divided into lines
- ▶ Each line consists of zero or more characters followed by a newline character (regardless of where the stream originates or where it goes to). The library makes each input or output stream conform to this model
- ▶ Standard library provides several functions for reading and writing one character at a time, of which `getchar(3)` and `putchar(3)` are the simplest.

53

getchar(3) and putchar(3)

```
#include <stdio.h>

int          int
getchar(void);  putchar(int c);
```

- ▶ `getchar(3)` reads the next input character from a text stream
- ▶ **Why does `getchar(3)` return an `int`?**
 - ▶ `getchar(3)` returns a distinctive value when there is no more input. A value, called EOF (end of file), that cannot be confused with any real data. EOF is defined in `<stdio.h>`
 - ▶ The return type must be big enough to hold EOF in addition to any possible char.
- ▶ `putchar(3)` prints a character each time it is called

55

File Copying

Given `getchar(3)` and `putchar(3)` ...

... we can write a surprising amount of useful code without knowing anything more about input and output

Copying input to output one character at a time

```
read a character
while (character is not end-of-file indicator)
    output the character just read
read a character
```

54

56

File Copying, v1

read a character
while (character is not end-of-file indicator)
output the character just read
read a character

```
1 #include <stdio.h>
2
3 /* copy input to output, v1 */
4 int
5 main(void)
6 {
7     int c;
8
9     c = getchar();
10    while (c != EOF) {
11        putchar(c);
12        c = getchar();
13    }
14
15    return (0);
16 }
```

57

Character Counting, v1

```
1 #include <stdio.h>
2
3 /* count characters in input, v1 */
4 int
5 main(void)
6 {
7     long nc;
8
9     nc = 0;
10    while (getchar() != EOF)
11        ++nc;
12    printf("%ld\n", nc);
13
14    return (0);
15 }
```

59

File Copying, v2

- ▶ An assignment, such as `c = getchar()` is an expression and has a value (value of the left hand side after the assignment)
- ▶ An assignment can appear as part of a larger expression

```
1 #include <stdio.h>
2
3 /* copy input to output, v2 */
4 int
5 main(void)
6 {
7     int c;
8
9     while ((c = getchar()) != EOF)
10        putchar(c);
11
12    return (0);
13 }
```

58

Character Counting, v2

```
1 #include <stdio.h>
2
3 /* count characters in input, v2 */
4 int
5 main(void)
6 {
7     double nc;
8
9     for (nc = 0; getchar() != EOF; ++nc)
10        ; /* nothing */
11    printf("%.0f\n", nc);
12
13    return (0);
14 }
```

60

Line Counting

- ▶ Standard library ensures that an input text stream appears as a **sequence of lines**, each terminated by a newline

```
1 #include <stdio.h>
2
3 /* count lines in input */
4 int
5 main(void)
6 {
7     int c, nl;
8
9     nl = 0;
10    while ((c = getchar()) != EOF)
11        if (c == '\n')
12            ++nl;
13    printf("%d\n", nl);
14
15    return (0);
16 }
```

```
1 #include <stdio.h>
2
3 #define IN 1 /* inside a word */
4 #define OUT 0 /* outside a word */
5
6 /* count lines, words and, characters in input */
7 int
8 main(void)
9 {
10    int c, nl, nw, nc, state;
11
12    state = OUT;
13    nl = nw = nc = 0;
14    while ((c = getchar()) != EOF) {
15        ++nc;
16        if (c == '\n')
17            ++nl;
18        if (c == '_' || c == '\n' || c == '\t')
19            state = OUT;
20        else if (state == OUT) {
21            state = IN;
22            ++nw;
23        }
24    }
25    printf("%d_%d_%d\n", nl, nw, nc);
26    return (0);
27 }
```

Word Counting

NAME

wc - word, line, and byte or character count

SYNOPSIS

wc [-c | -m] [-hlw] [file ...]

DESCRIPTION

The wc utility reads one or more input text files, and, by default, writes the number of lines, words, and bytes contained in each input file to the standard output

```
$ wc /etc/services
285   1398   9732 /etc/services
$ cc count_words.c
$ cat /etc/services | ./a.out
285 1398 9732
```

61

62

From Source Code To Executable Code

Storage Classes

The C Preprocessor

Variables and Arithmetic Expressions

Character Input and Output

Arrays

63

64

Counting Digits, White Spaces, And The Rest

Next is an artificial program, which counts the number of occurrences of each digit, of white space characters (blank, tab, newline), and all other characters.

It will help us to ...

- ▶ introduce arrays
- ▶ talk about initialization
- ▶ see that chars are, by definition, just small integers
- ▶ speak about coding conventions

The output of the program on itself is:

```
$ cat count_digits.c | ./a.out
digits = 10 3 0 0 0 0 0 0 0 1, white space=122, other=361
$ wc -m count_digits.c
497 count_digits.c
```

```
1 #include <stdio.h>
2
3 /* count digits, white space, others */
4 int
5 main(void)
6 {
7     int c, i, nwhite, nother;
8     int ndigit[10];
9
10    nwhite = nother = 0;
11    for (i = 0; i < 10; ++i)
12        ndigit[i] = 0;
13
14    while ((c = getchar()) != EOF)
15        if (c >= '0' && c <= '9')
16            ++ndigit[c-'0'];
17        else if (c == '_' || c == '\n' || c == '\t')
18            ++nwhite;
19        else
20            ++nother;
21
22    printf("digits_=");
23    for (i = 0; i < 10; ++i)
24        printf("_%d", ndigit[i]);
25    printf(",_white_space=%d,_other=%d\n", nwhite, nother);
26
27    return (0);
28 }
```