COMP327 Mobile Computing

Session: 2011-2012

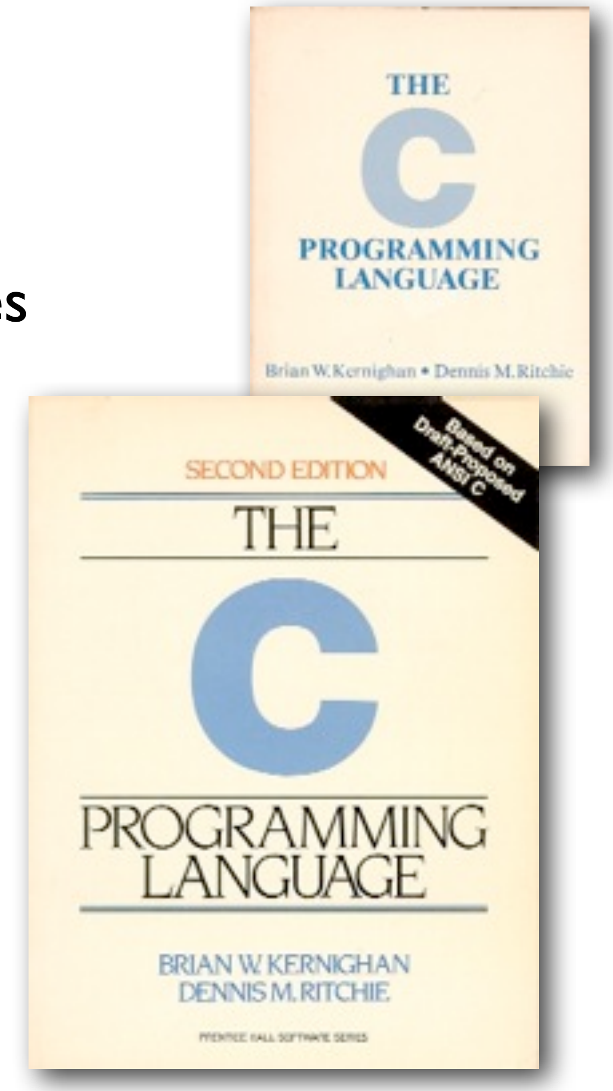
Lecture Set 2a - Introduction to C

In these Tutorial Slides...

- History of C
  - How it relates to OOP
- Introduction to C
  - K&R's "Hello World!"
  - Compiling C
  - Walkthrough of the K&R Chapter 1 Tutorial

Overview of these Tutorials

- Much of the material here has been sourced from a variety of places
  - Mainly from K&R's C (2nd Edition)
    - The "C Bible" from the guys who wrote the language
    - This is a *great* book from which to learn C
    - Revised from the 1st edition and based on the ANSI C standard
- Aim is to provide not only an understanding of the language, but to understand what is happening "under the hood"
  - Invaluable for:
    - Debugging code, and writing good, safe code
    - Managing and manipulating memory
    - Understanding why the language works as it does
      - (and hence the same for Objective-C, Java, etc)
- Aim is also to provide the basis for understanding Objective-C, so that you have the language skills to use the iPhone SDK
- **NOTE: The content in these slides can be dense, but they are meant also to act as a reference!**


THE C PROGRAMMING LANGUAGE
Brian W. Kernighan • Dennis M. Ritchie
SECOND EDITION
THE C PROGRAMMING LANGUAGE
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE
Based on Draft-Proposed ANSI C

Published in 1988, 10 years after the first edition

Brief History of C

The Language "C"

- **Developed** in the 1970s by Dennis Ritchie at Bell Laboratories
  - Language features derived from B (Ken Thompson, 1970), itself a stripped down version of BCPL.
  - Closely tied to the development of Unix (Ken Thompson, Dennis Ritchie, Brian Kernighan, Douglas McIlroy, Joe Ossanna & Bell Labs), which was developed in 1969 and re-written in 1972 in C.
- **Standardized** in 1989 by ANSI (American National Standards Institute) known as ANSI C
  - International standard (ISO) in1990 which was adopted by ANSI and is known as C89
  - The standard was also updated in 1995 (C95) and 1999 (C99)

Languages evolved from "C"

- **Extended** with Object Oriented Programming support
  - **C++** extends C to include support for OOP and other features that facilitate large software development projects
    - C is not strictly a subset of C++, but it is possible to write "Clean C" that conforms to both the C++ and C standards.
  - **Objective-C** is a strict superset of ANSI C that supports OOP
    - Very different (and somewhat simpler) to C++; it inherits many principles from Smalltalk

Why use C?

- C is an imperative language (procedural)
    - Designed with a straightforward syntax for simple compilation
        - structured programming
        - weakly typed with a static type system
        - support for lexical variable scope
    - Provides low-level access to memory
        - including addressable objects such as device control registers
    - Language constructs map efficiently to machine instructions
    - Requires minimal run-time support
- Widely used to code applications previously written in assembly language
    - Originally used extensively for system level programming
    - Subsequently employed for application programming
- Has provided the basis for more recent, higher level languages
    - C++, Objective-C, Java, C#, etc

C, Operating Systems and Embedded Systems

- When writing an OS kernel, efficiency is crucial
    - This requires low-level access to the underlying hardware:
        - e.g. programmer can leverage knowledge of how data is laid out in memory, to enable faster data access
    - Such requirements are still important for embedded systems
        - e.g. small devices where memory and processor speed are limited, or real-time and safety-critical systems
- Unix originally written in low-level assembly language – but there were problems:
    - No structured programming (e.g. encapsulating routines as "functions", "methods", etc.) – code hard to maintain
    - Code worked only for particular hardware – not portable
- Implementing Unix in C facilitated portability to other architectures
    - only a small amount of assembly code was therefore required per machine

C’s characteristics

- C is one of the most popular programming languages of all time
  - There are very few computer architectures for which a C compiler does not exist.
- C takes a middle path between low-level assembly language…
  - Direct access to memory layout through pointer manipulation
  - Concise syntax, small set of keywords
- … and a high-level programming language like Java:
  - Block structure
  - Some encapsulation of code, via functions
  - Type checking (pretty weak)
  - Library support for SDKs and 3rd party code

C's Dangers

- HOWEVER...
  - C is not object oriented!
    - Can't "hide" data as "private" or "protected" fields
    - You can follow standards to write C code that looks object-oriented, but you have to be disciplined – will others working on your code also be disciplined?
  - C has portability issues
    - Low-level "tricks" may make your C code run well on one platform – but the tricks might not work elsewhere
    - Compiled executables are architecture dependent, unlike Java's bytecode
  - The compiler and runtime system will rarely stop your C program from doing stupid/bad things
    - Compile-time type checking is weak
    - No run-time checks for array bounds errors, etc. like in Java

Java vs. C

- Many similarities with primitive data types
  - char, short, int, long, float, and double
    - size of these data types in C varies with destination architecture
  - Java also introduces boolean and byte (8-bit integers)
    - Java chars are 16-bit Unicode, whereas C chars are 8-bit
  - Operators and Statements are similar in both languages
    - if, else, switch, case, break, default, for, do, while, continue, and return.
- However:
  - Java checks for errors or enforces error checking
  - Java provides expandable arrays, container classes etc.
    - Java arrays are separate objects; C arrays are just pointers to memory
    - Java has string objects; C relies on C char array
  - Java has a huge number of classes libraries and packages
    - But C has better access to system calls and the OS
  - Java replaces pointers, pointer arithmetic & memory management with reference, subscription & garbage collection

Runtime / Performance

Java compiles to bytecode, which is then run within a Java virtual machine. The machine provides many services in the background, such as thread management and garbage collection.

The C environment is simpler and runs native executables. Therefore performance is typically faster - both due to the lack of other services, and because of the lack of interpretation.

Getting Started K&R's Tutorial Intro

- Every first program should be to print the words "hello, world"
  - Create a file that ends in ".c"

```
#include <stdio.h>
main() {
    printf("hello, world\n");
}
```

  - Compile it using cc or gcc
    - gcc hello.c
  - This generates an executable called a.out
    - which can then be run from the command line
    - the OS will then run this as native executable code

```
% ./a.out
hello, world
%
```

What is happening?

This tells the compiler to include info about the standard input/output library - necessary for the printf function.

A function definition consists of a name, a list of arguments and then commands enclosed in braces.

The topmost function is "main", which must always be defined. This definition assumes no return value and no arguments.

```
#include <stdio.h>
main() {
    printf("hello, world\n");
}
```

printf is a library function followed by a parenthesised list of arguments. It prints a string to standard output (without any inplicit terminating newline character).

A sequence of characters inside double quotes is a string constant. The \n is notation for a newline character.

Compilation and Development Environment

- A C development environment includes
  - **System libraries** and **headers**: a set of standard libraries and their header files.
    - For example see /usr/include and glibc.
  - **Application Source**: application source and header files
  - **C preprocessor (cpp):** used to insert common definitions into source files
  - **Compiler**: converts source to object code for a specific platform
  - **Linker**: resolves external references and produces the executable module

- Header files (*.h) export interface definitions
  - Function prototypes, data types, macros, inline functions and other common declarations
  - Do not place source code (i.e. definitions) in the header file with a few exceptions.
    - inline'd code macro definitions
    - class definitions
    - constant definitions (i.e. #defines)

Standard Header Files

- Several standard libraries available (though tiny compared to Java):
  - **stdio.h** – file and console (also a file) IO: *perror, printf, open, close, read, write, scanf*, etc.
  - **stdlib.h** - common utility functions: *malloc, calloc, strtol, atoi*, etc
  - **string.h** - string and byte manipulation: *strlen, strcpy, strcat, memcpy, memset*, etc.
  - **ctype.h** – character types: *isalnum, isprint, isupport, tolower*, etc.
  - **errno.h** – defines errno used for reporting system errors
  - **math.h** – math functions: *ceil, exp, floor, sqrt*, etc.
  - **signal.h** – signal handling facility: *raise, signal*, etc
  - **stdint.h** – standard integer: *intN_t, uintN_t*, etc
  - **time.h** – time related facility: *asctime, clock, time_t*, etc.
- More information available by using the Unix man command
  - e.g. man 3 printf, or view the files in /usr/include

Full list can be viewed at http://en.wikipedia.org/wiki/C_library

Compiling C code (from the Unix command line)

- To compile and link a C program that is contained entirely in one source file:
  - % cc program.c
  - The executable program is called a.out by default.

- If you don't like this name, choose another using the –o option:
  - % cc program.c –o exciting_executable

- To compile and link several C source files:
  - % cc main.c extra.c more.c
  - This will produce object (.o) files, that you can use in a later compilation:
    - % cc main.o extra.o moreV2.c
  - Here, only moreV2.c will be compiled – the main.o and extra.o files will be used for linking.

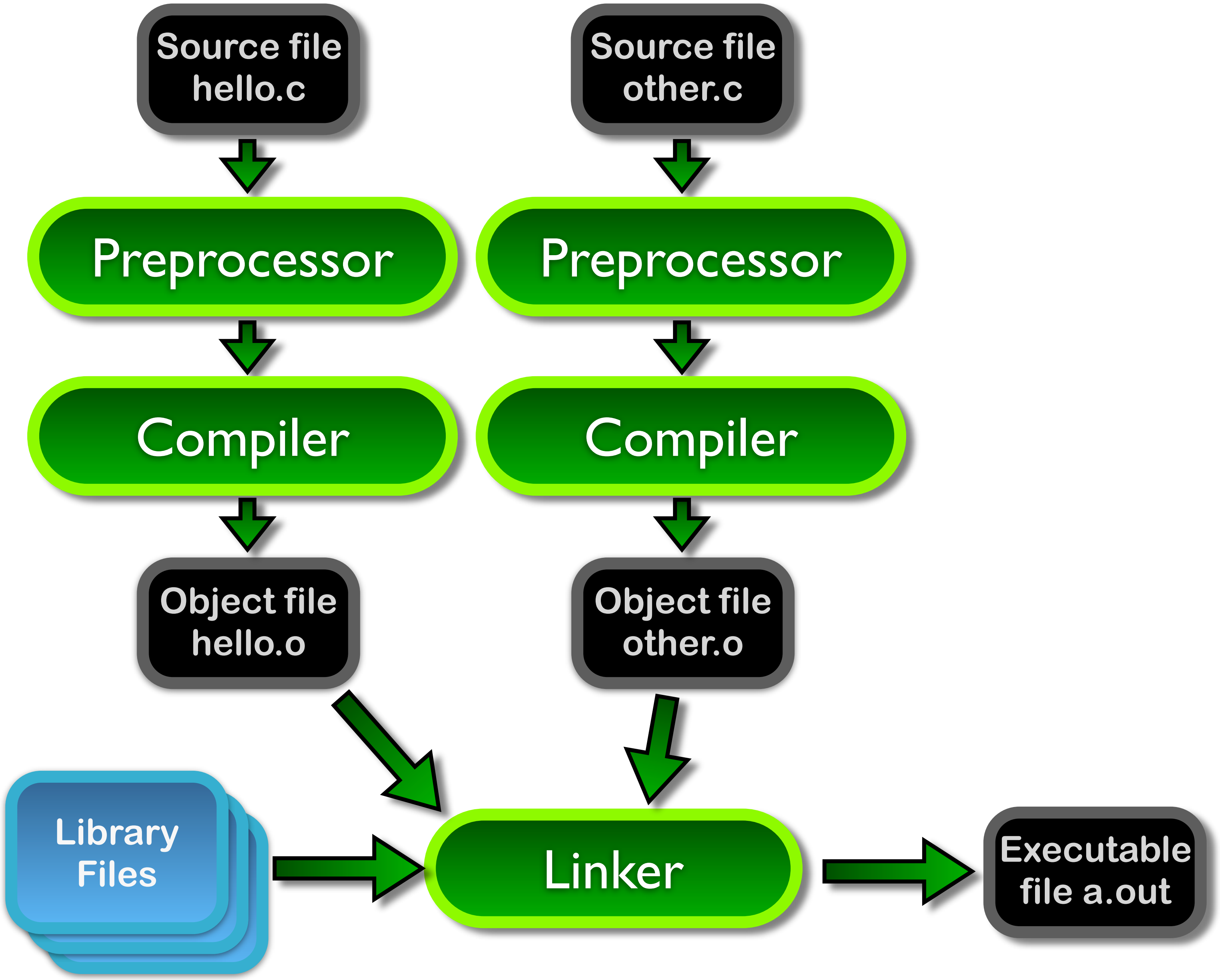
- To produce object files, without linking, use -c:
  - % cc –c main.c extra.c more.c

Compiling C code

- A C program consists of source code in one or more files

- Each source file is run through the preprocessor and compiler, resulting in a file containing object code

- Object files are tied together by the linker to form a single executable program

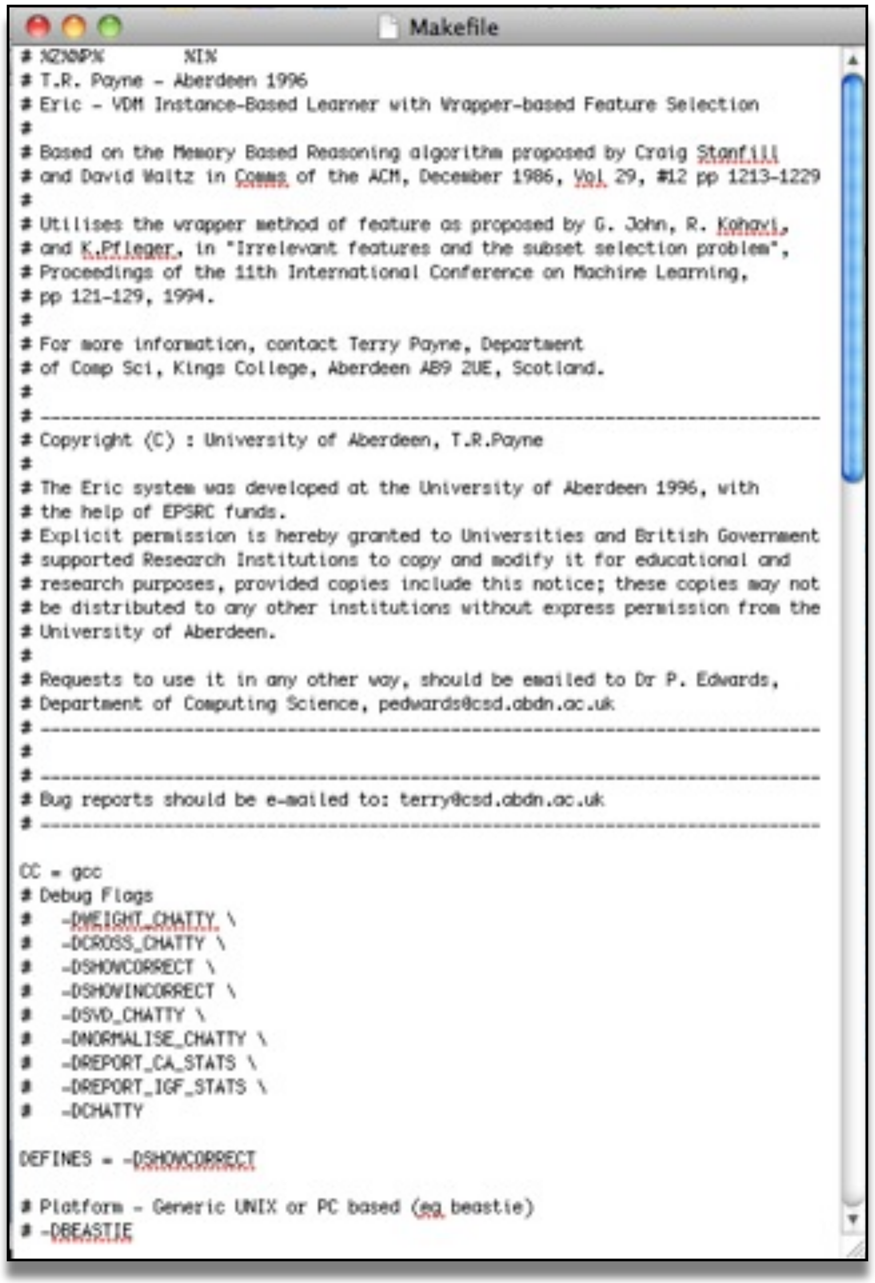


Separate Compilation

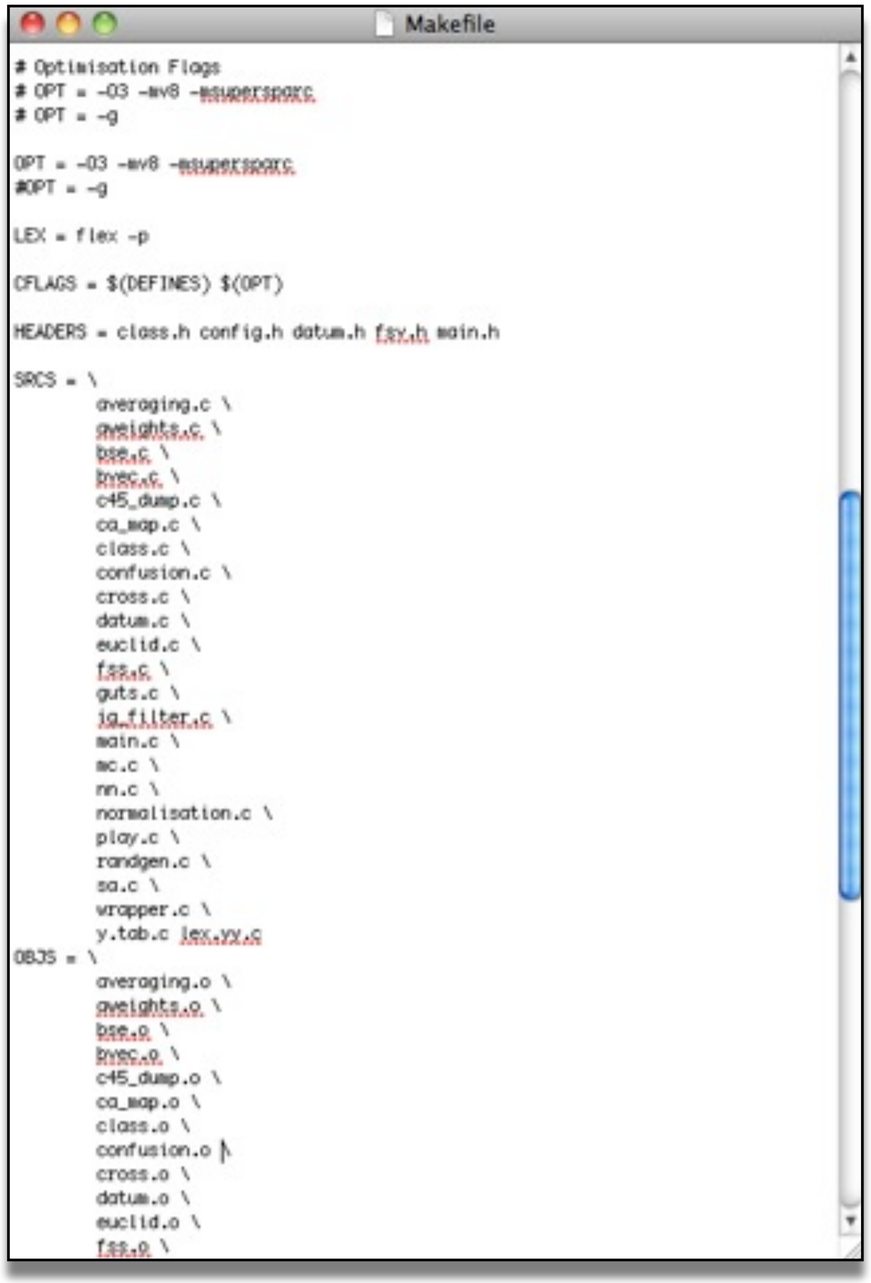
- Advantage: Quicker compilation
  - When modifying a program, a programmer typically edits only a few source code files at a time.
  - With separate compilation, only the files that have been edited since the last compilation need to be recompiled when re-building the program.
  - For very large programs, this can save a lot of time.

- Various tools can be used to manage object compilation
  - Makefile can be used to manage dependencies between object files, ensuring that only modified source files necessary for an application are recompiled.
    - Often used in larger projects, where other tools are used, such as lex and yacc for parsers
  - IDE's such as Apple's Xcode or Microsoft Visual Studio also do this...

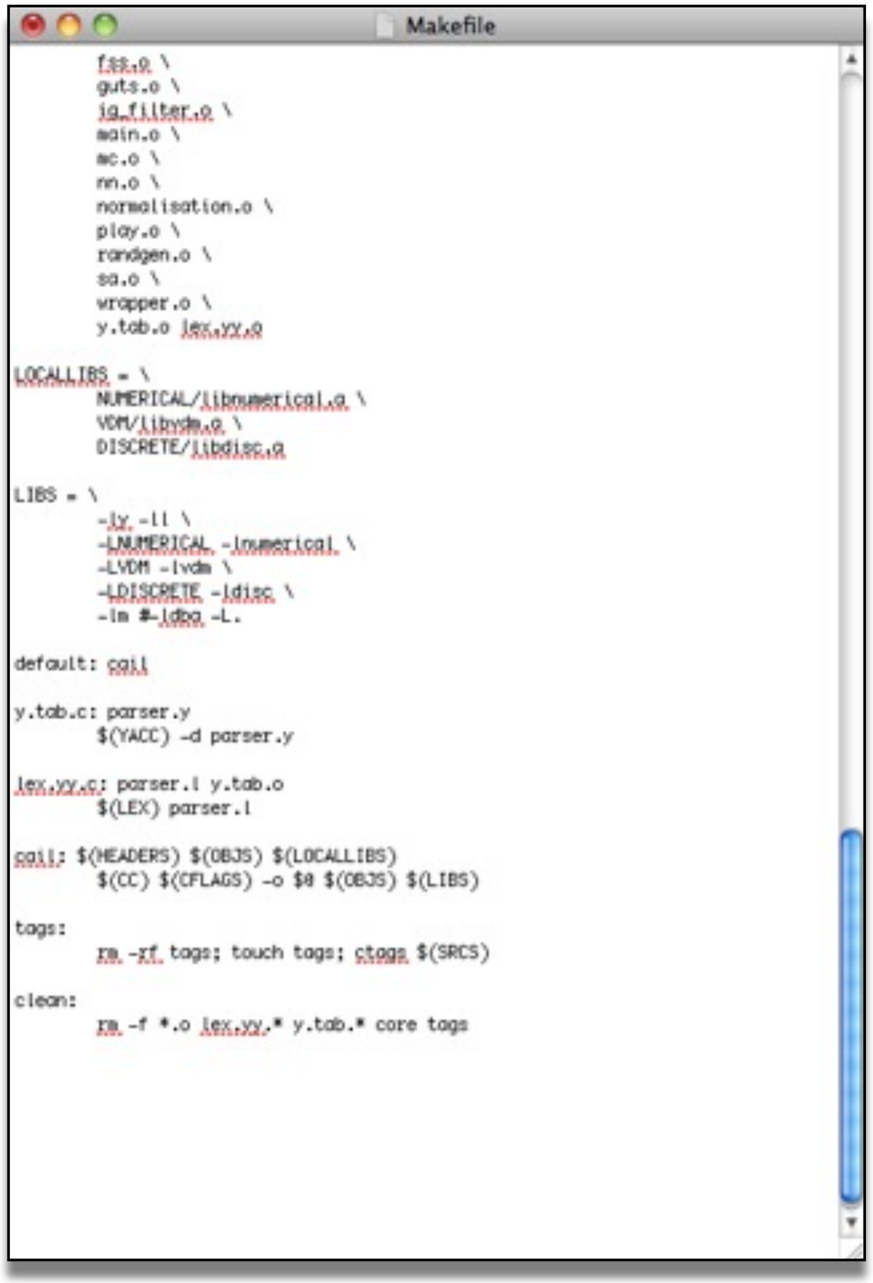
Example Makefile



```
# %Z%%P%        %I%
# T.R. Payne - Aberdeen 1996
# Eric - VDM Instance-Based Learner with Wrapper-based Feature Selection
#
# Based on the Memory Based Reasoning algorithm proposed by Craig Stanfill
# and David Waltz in Comms of the ACM, December 1986, Vol 29, #12 pp 1213-1229
#
# Utilises the wrapper method of feature as proposed by G. John, R. Kohavi,
# and K.Pfleger, in "Irrelevant features and the subset selection problem",
# Proceedings of the 11th International Conference on Machine Learning,
# pp 121-129, 1994.
#
# For more information, contact Terry Payne, Department
# of Comp Sci, Kings College, Aberdeen AB9 2UE, Scotland.
#
# ---------------------------------------------------------------------------
# Copyright (C) : University of Aberdeen, T.R.Payne
#
# The Eric system was developed at the University of Aberdeen 1996, with
# the help of EPSRC funds.
# Explicit permission is hereby granted to Universities and British Government
# supported Research Institutions to copy and modify it for educational and
# research purposes, provided copies include this notice; these copies may not
# be distributed to any other institutions without express permission from the
# University of Aberdeen.
#
# Requests to use it in any other way, should be emailed to Dr P. Edwards,
# Department of Computing Science, pedwards@csd.abdn.ac.uk
# ---------------------------------------------------------------------------
#
# ---------------------------------------------------------------------------
# Bug reports should be e-mailed to: terry@csd.abdn.ac.uk
# ---------------------------------------------------------------------------

CC = gcc
# Debug Flags
#   -DWEIGHT_CHATTY \
#   -DCROSS_CHATTY \
#   -DSHOWCORRECT \
#   -DSHOWINCORRECT \
#   -DSVD_CHATTY \
#   -DNORMALISE_CHATTY \
#   -DREPORT_CA_STATS \
#   -DREPORT_IGF_STATS \
#   -DCHATTY

DEFINES = -DSHOWCORRECT

# Platform - Generic UNIX or PC based (eg beastie)
# -DBEASTIE
```



```
# Optimisation Flags
# OPT = -O3 -mv8 -msupersparc
# OPT = -g

OPT = -O3 -mv8 -msupersparc
#OPT = -g

LEX = flex -p

CFLAGS = $(DEFINES) $(OPT)

HEADERS = class.h config.h datum.h fsy.h main.h

SRCS = \
        averaging.c \
        aweights.c \
        bse.c \
        bvec.c \
        c45_dump.c \
        ca_map.c \
        class.c \
        confusion.c \
        cross.c \
        datum.c \
        euclid.c \
        fss.c \
        guts.c \
        ig_filter.c \
        main.c \
        mc.c \
        nn.c \
        normalisation.c \
        play.c \
        randgen.c \
        sa.c \
        wrapper.c \
        y.tab.c lex.yy.c
OBJS = \
        averaging.o \
        aweights.o \
        bse.o \
        bvec.o \
        c45_dump.o \
        ca_map.o \
        class.o \
        confusion.o \
        cross.o \
        datum.o \
        euclid.o \
        fss.o \
```



```
        fss.o \
        guts.o \
        ig_filter.o \
        main.o \
        mc.o \
        nn.o \
        normalisation.o \
        play.o \
        randgen.o \
        sa.o \
        wrapper.o \
        y.tab.o lex.yy.o

LOCALLIBS = \
        NUMERICAL/libnumerical.a \
        VDM/libvdm.a \
        DISCRETE/libdisc.a

LIBS = \
        -ly -ll \
        -LNUMERICAL -lnumerical \
        -LVDM -lvdm \
        -LDISCRETE -ldisc \
        -lm #-ldbg -L.

default: cail

y.tab.c: parser.y
        $(YACC) -d parser.y

lex.yy.c: parser.l y.tab.o
        $(LEX) parser.l

cail: $(HEADERS) $(OBJS) $(LOCALLIBS)
        $(CC) $(CFLAGS) -o $@ $(OBJS) $(LIBS)

tags:
        rm -rf tags; touch tags; ctags $(SRCS)

clean:
        rm -f *.o lex.yy.* y.tab.* core tags
```

Fahrenheit-Celsius v1

Comments are any strings between /* and */ and can span multiple lines. NOTE: comments can’t be nested.

Variables are *declared* **before** they are used, normally before any executable statements. They declare the properties of variables (and consequently their memory requirements).

```
#include <stdio.h>

/* print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300 */

main() {
    int fahr, celsius;
    int lower, upper, step;

    lower = 0;    /* lower limit of temperature scale */
    upper = 300;  /* upper limit */
    step = 20;    /* step size */

    fahr = lower;
    while (fahr <= upper) {
        celsius = 5 * (fahr-32) / 9;
        printf(“%d\t%d\n”, fahr, celsius);
        fahr = fahr + step;
    }
}
```

Note that these variables are ints (i.e. no fractional part).

Computation starts with the *assignment* statements. Individual statements are terminated with a semi-colon.

The while loop is then responsible for the main calculation, and displaying the results.

Fahrenheit-Celsius v1

Check the condition in the parenthesis. If true (i.e. **`fahr`** is less than or equal to upper), then execute body. Body can be a single statement (terminated by a semicolon) or several statements enclosed in braces.

Celsius temperature is computed and assigned to the variable celsius.

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

Note that we multiply by 5 and then divide by 9, instead of multiplying by 5/9. This is because the numbers 5 and 9 are integers, as are **`celsius`** and **`fahr`**. Thus all calculations are done as integer calculations (and 5/9 would evaluate to 0)

```
%./a.out
0   -17
20  -6
40  4
60  15
80  26
100 37
120 48
140 60
160 71
180 82
200 93
220 104
240 115
260 126
280 137
300 148
%
```

%d	signed decimal int	%c	character
%f	float/double	%s	string (char array)
%e	double (with exponent)	%%	itself

printf takes as arguments a string of characters, and optional variables (depending on the formatting characters, denoted by %). Thus, this line prints an integer, a tab (\t), another integer, and the newline character (\n).

Fahrenheit-Celsius v2

```c
#include <stdio.h>

/* print Fahrenheit-Celsius table
   for fahr = 0, 20, ..., 300 */

main() {
    float fahr, celsius;
    float lower, upper, step;

    lower = 0;     /* lower limit of temperature scale */
    upper = 300;   /* upper limit */
    step = 20;     /* step size */

    fahr = lower;
    while (fahr <= upper) {
        celsius = (5.0/9.0) * (fahr-32.0);
        printf("%3.0f\t%6.1f\n", fahr, celsius);
        fahr = fahr + step;
    }
}
```

This version declares the variables as floats, and performs floating point arithmetic.

We calculate the fraction 5.0/9.0 before multiplying it with **`fahr`**-32.0. Note that the numbers have a decimal point to indicate that they are floating point operands. If we used 5/9, this would be evaluated as integer division (resulting in zero) before then being multiplied by the float.

```
%./a.out
  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
%
```

The printf statement displays values as floats. The first is at least 3 characters wide, but no characters after the decimal point. The second is at least 6 characters wide, with 1 decimal point.

Fahrenheit-Celsius v3

Now most of the variables have been eliminated, with the upper and lower limits appearing as **magic numbers** in the for loop, with the calculation appearing in the printf statement.

```
#include <stdio.h>

/* print Farenheit-Celsius table */

main() {
    int fahr;

    for (fahr = 0; fahr <= 300; fahr = fahr + 20)
        printf("%3d %6.1f\n", fahr, (5.0/9.0)*(fahr-32));
}
```

Note: as printf expects a float, any expression that generates a float can be used. If an expression produces another type, then a cast is required (see later).

To avoid magic numbers appearing, **symbolic constants** can be defined. Note that the convention is to use UPPER case. These symbols are replaced by their values by the **pre-processor**.

```
...
#define LOWER 0       /* lower limit of table */
#define UPPER 300     /* upper limit */
#define STEP 20       /* step size */
...
    for (fahr = LOWER; fahr <= UPPER; fahr = fahr + STEP)
...
```

Character I/O

```
#include <stdio.h>

/* copy input to output; 1st version */
main() {
    int c;
    c = getchar();
    while (c != EOF) {
        putchar(c);
        c = getchar();
    }
}
```

In this first version we read a character from the standard input (e.g. keyboard) and write it to standard output (e.g. screen).

EOF is a value returned when the end of file is reached. We therefore test for it to know when to end. As EOF is an integer defined in stdio.h, we use ints instead of chars.

```
#include <stdio.h>

/* copy input to output; 2nd version */
main() {
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

An assignment returns the value of itself. Thus, an assignment can also appear in an expression. This is useful, as the assignment from **`getchar()`** can appear within the boolean test for the while loop, allowing a more concise version of this code.

A note on precedence

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * (*type*) sizeof	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	right to left
= += -= *= /= %= &= ^= |= <<= >>=	right to left
,	left to right

Increasing precedence

With the previous example, parentheses were required as equality tests (== or !=) have a **higher precedence than** assignments or assignment operators.

```
c = getchar() != EOF
```

would be equivalent to

```
c = (getchar() != EOF)
```

which would return 1 or 0, and not the character!

Conditional expressions (?:) allow if-then tests to appear within assignments, in the form:

expr$_1$: expr$_2$? expr$_3$

For example:

```
z = (a > b) ? a : b;
```

is equivalent to

```
if (a > b)
    z = a;
else
    z = b;
```

Other examples:

```
printf("You have %d items%s.\n", n, n==1 ? "" : "s");
```

```
for (i = 0; i < n; i++)
    printf("%6d%c", a[i], (i%10==9 || i==n-1) ? '\n' : ' ');
```

Counting Characters

```
#include <stdio.h>

/* count characters in input; 1st version */
main() {
    long nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    printf("%ld\n", nc);
}
```

This version counts (in a long int) the number of characters found in the input. If a long were only 16 bits (on earlier architectures), then a maximum of 32767 chars could be read! The %ld in printf states the number is a long int.

```
#include <stdio.h>

/* count characters in input; 1st version */
main() {
    double nc;

    for (nc = 0; gechar() != EOF; ++nc)
        ;
    printf("%.0f\n", nc);
}
```

This variant uses a double float to count the characters, as doubles are bigger than longs.

All the work is done by the for statement, which has no body! The isolated semi-colon is a *null statement* (i.e. do nothing).

Note in both cases, with empty input (i.e. EOF is the first character,) then nothing is counted, as test are done before executing the **`for`** or **`while`** body.

Counting Lines

```
#include <stdio.h>

/* count lines in input */
main() {
    int c, nl;

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;

    printf("%d\n", nl);
}
```

Characters as ints

A character in single quotes (i.e. a *character constant*) represents an integer value equal to the numerical value of the character in the machine's character set. E.g. 'A' is ascii value 65. The newline character '\n' is ascii value 10.

NOTE that although there are two characters between the quotes, the backslash indicates that the sequence is an escaped character.

Equality Tests

The equality test == is used to check if the character is a newline \n. This is different to the = assignment operator.

Often, using the assignment operator instead of the equality test generates no error, but causes a bug. For example, the bug...

```
if (c=5)
```

...is valid, as the expression **`c=5`** also returns 5, which is non-zero, and thus true. To avoid this when testing a constant, put the constant first. E.g.

```
if (5 = c)
```

is invalid and will raise a compiler error, but

`if (5 == c)` is valid!

Counting Words

```
#include <stdio.h>

#define IN   1 /* inside a word */
#define OUT 0 /* outside a word */

/* count lines, words, and characters in input */
main() {
    int c, nl, nw, nc, state;

    state = OUT;
    nl = nw = nc = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (c == ' ' || c == '\n' || c = '\t')
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    printf("%d %d %d\n", nl, nw, nc);
}
```

A basic implementation of “wc”

The integer **`state`** determines whether we are in or out of a word. The constants **`IN`** and **`OUT`** make the code easier to read, than using 1 and 0.

This line assigns all the variables to 0. Because the assignment operator returns its assignment value, this is equivalent to writing:

`nl = (nw = (nc = 0));`

The if-then-else specifies a true and an (optional) false statement. The statement can be a single statement (ending in semicolon), or multiple statements within braces.

Counting Words

Testing for word boundaries

The code tests for words by looking for spaces (‘ ‘), newlines (‘\n’), or tab (‘\t’) characters.

```
        ...
        if (c == ' ' || c == '\n' || c = '\t')
            state = OUT;
        else if (state == OUT) {
            ...
```

Logical AND and Logical OR

Tests can be combined by using logical AND (&&) and logical OR (||). These expressions are evaluated from left to right. However, the evaluation stops as soon as the truth or falsehood of the result is known. Thus, not all the tests in a logical expression may be called.

This can be used to create conditional sub-expressions, by including tests before performing actions in an expression. If the test fails in an AND expression (or succeeds in an OR expression), then the remainder of the logical expression will not be evaluated.

The code fragment (below) is taken from a program to store characters (read from stdin) into an array s of size lim, where i characters have been counted.

```
for (i=0; i < lim-1 &&
          (c=getchar()) != '\n'
          && c != EOF; ++i)
    s[i] = c;
```

Before reading the new character, we should check there is space in the array. If the test **`i < lim-1`** fails, then we do not go on to read another character. Likewise, we only test that **`c != EOF`** after the character has been read.

Counting using Arrays

```
#include <stdio.h>

/* count digits, white space, others */
main() {
    int c, i, nwhite, nother;
    int ndigit[10];

    nwhite = nother;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c-'0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;

    printf("digits =");
    for (i = 0; i < 10; ++i)
        printf(" %d", ndigit[i]);
    printf(", white space = %d, other = %d\n",
         nwhite, nother);
}
```

A simple program to count the occurrence of digits, whitespace and other characters.

This declares an array of 10 digits (with an array subscript of 0..9). Array declarations only allocate static memory, so their values need to be initialised (this goes for all variables). **Note**, there is no bounds checking in C, so testing or setting values beyond the array will not generate a compiler error.

As chars are actually small (8-bit) ints, they can be used in integer arithmetic. Given that all character sets (ascii etc) store digits sequentially, testing chars to see if they are digits (and retrieving that number) is simple.

Character Arrays

```
#include <stdio.h>

#define MAXLINE 1000 /* maximum input line length */

int getline(char line[], int maxline);
void copy(char to[], char from[]);

/* print the longest input line */
main() {
    int len;                /* current line length */
    int max;                /* maximum length seen so far */
    char line[MAXLINE];     /* current input line */
    char longest[MAXLINE]; /* longest line saved here */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
        max = len;
        copy(longest, line);
    }
    if (max > 0)    /* there was a line */
        printf("%s", longest);
    return 0;
}
```

A simple program (split over two slides) to read text and print the longest input line.

The algorithm consists of **`main`**, a function to get a new line from the input, and a function to save the line if it is the longest encountered. These two functions are **declared** before being used.

`getline` returns the length of the current line. Blank lines consist of at least one character (newline); so a zero is returned to signify the end of the file.

Character Arrays

```
/* copy: copy 'from' into 'to';
 * assume 'to' is big enough
 */
 void copy(char to[], char from[]) {
     int i;

     i = 0;
     while ((to[i] = from[i]) != '\0')
         i++;
}
```

There is no “string” data type in C. Instead, strings are represented as char arrays, with a *null character* ‘**`\0`**’ (whose value is zero) at the end of the string in the array.

If the string constant “**`hello\n`**” appears, it is stored in a seven element char array (six elements for the characters, and one for the null character).

h	e	l	l	o	\n	\0

```
/* getline: read a line into s, and return length */
int getline(char s[], int lim) {
    int c, i;

    for (i=0; i < lim-1 && (c=getchar())!=EOF && c!= '\n'; ++i)
        s[i] = c;
    if (c == '\n')
        s[i++] = c; /* If c is a newline char, then add this */
    s[i] = '\0';
    return i;
}
```

As getline cannot know how big an input line can be, it checks for overflow (i.e. it checks for the size of the array, minus 1 char for the null character).

Passing Arrays to functions

Function declarations that take arrays should specify that the variable is an array of a given type. This is treated as a *pointer* (see later).

The array declaration states the elements' type (in this case, char), and the size of the array. All arrays defined in this way are static (there memory is allocated in the function definition), and are not resizable.

Call by Value

In C, all values are passed to functions *by value*. When a function is called, a **copy** of the values passed as arguments are used by the function. If the function then changes these values, the changes are only local to the function, and do not *side-affect* the calling function.

Sometimes, it is desirable for a function to modify the calling function's data. This is done by passing the address of the variable (i.e. a **pointer**), discussed later.

```
int getline(char line[], int maxline);
void copy(char to[], char from[]);
...
    char line[MAXLINE];    /* current input line */
    char longest[MAXLINE]; /* longest line saved here */
    ...
    while ((len = getline(line, MAXLINE)) > 0)
```

Arrays as a special case?
When the name of an array is used as an argument, the value passed to the function is the location or address of the beginning of the array - there is no copying of array elements.

When the name of the array is passed to getline, a copy of this location (i.e. its memory address) is made, just as passing any variable. However, the array values can be modified!

External Variables

This variant of the previous application uses **external** variables, rather than passing local variables.

```
#include <stdio.h>

#define MAXLINE 1000 /* maximum input line length */

int max;                 /* maximum length seen so far */
char line[MAXLINE];      /* current input line */
char longest[MAXLINE]; /* longest line saved here */

int getline(void);
void copy(void);

/* print the longest input line */
main() {
    int len;             /* current line length */
    extern int max;
    extern char longest[];

    max = 0;
    while ((len = getline()) > 0)
        ....
```

An external variable must be ***defined*** only once (outside a function) so that memory can be allocated to it.

A external variable must be ***declared*** (as extern) in each function that wants to access it. This ensures that the compiler can check the type, but prevents any additional memory from being allocated.

Extern declarations (and function prototypes) are not always necessary - e.g. if their definitions appear before their use.

Avoid using external (global) variables whenever possible

To Summarise

- We've now covered the conventional core of the language.
    - Basic structure of main and functions
    - The notion of libraries, header files, declaration of extern variables and symbolic constants
    - Basic variable types, their declaration and definition
    - The main syntax of the language
    - Character Arrays (for managing strings), and exploiting char arithmetic
    - Expressions, precedence, logical operators, conditionals, loops
    - Passing values by value, and arrays as memory references
- We've also looked at a brief history of C, and why it is still significant as a programming language,
- The next slide set will explore other language features in detail

Simple Exercises

- Exercise 1-9. Write a program to copy its input to its output, replacing each string of one or more spaces by a single space.
- Exercise 1-13. Write a program to print a histogram of the lengths of words in its input. It is easy to draw the histogram with the bars horizontal; a vertical orientation is more challenging.
- Exercise 1-14. Write a program to print a histogram of the frequencies of different characters in its input.
- Exercise 1.15. Rewrite the Fahrenheit-Celsius v3 temperature conversion program to use a function for conversion.
- Exercise 1-18. Write a program to remove trailing spaces and tabs from each line of input, and to delete entirely blank lines.
- Exercise 1-19. Write a function reverse(s) that reverses the character string s. Use it to write a program that reverses its input a line at a time.

Challenging Exercises

- Exercise 1-20. Write a program detab that replaces tabs in the input with the proper number of spaces to space-out to the next tab stop. Assume a fixed set of tab stops, say every n columns. Should n be a variable or a symbolic parameter?
- Exercise 1-21. Write a program entab that replaces strings of spaces by the minimum number of tabs and spaces to achieve the same spacing. Use the same tab stops as for detab. When either a tab or a single space would suffice to reach a tab stop, which should be given preference?
- Exercise 1-22. Write a program to “fold” long input lines into two or more shorter lines after the last non-blank character that occurs before the n-th column of input. Make sure your program does something intelligent with very long lines, and if there are no spaces or tabs before the specified column.
- Exercise 1-23. Write a program to remove all comments from a C program. Don't forget to handle quoted strings and character constants properly. C comments don't nest.