An introduction to program design and implementation using C99

S.P. Platt
www.spplatt.co.uk

Revised August 2005

© S.P. Platt 2001-2005

Contents

1 Introduction ... 5
2 Getting started ... 8
2.1 A first program ... 8
2.2 Design notation ... 10
2.3 Program errors ... 10
2.3.1 Syntax errors ... 10
2.3.2 Run time errors ... 11
3 Using data – characters and strings ... 12
3.1 Variables ... 12
3.2 Using characters ... 13
3.3 The character set ... 15
3.4 Character strings ... 15
4 Using numerical data ... 18
4.1 Using integers ... 18
4.2 Using floating-point numbers ... 20
4.3 Integer and floating-point arithmetic ... 20
4.4 Single and double-precision floating-point numbers ... 21
5 Formatted Input/Output (IO) ... 23
5.1 Formatted input ... 23
5.2 Formatted output ... 25
5.3 Characters are really numbers ... 26
5.4 Robust IO ... 28
6 Selection between two alternatives – if and if-else ... 29
6.1 If ... 29
6.2 If-else ... 30
6.3 Boolean values ... 31
6.4 Making comparisons ... 33
6.5 Boolean operations ... 33
7 Selecting among several options – else-if and switch ... 34
7.1 Else-if ... 34
7.2 Switch ... 36
7.2.1 Breaking out and falling through ... 39
7.3 Switch versus if-else and else-if ... 42
8 Programming style ... 43
8.1 Code Layout ... 44
8.2 Naming ... 45
8.3 Comments ... 45
9 Iteration ... 46
9.1 Do-while loop ... 46
9.2 While loop ... 48
9.3 For loop ... 49
9.4 What sort of loop? ... 51
10 Arrays ... 52
10.1 One-dimensional arrays ... 52
10.1.1 Character array (string) example ... 52
10.1.2 Numerical array example ... 54
10.2 Multi-dimensional arrays ... 57

© S.P. Platt 2001-2005

11 Data types and addresses 59
11.1 Integral types 59
11.2 Floating-point types 59
11.3 Type conversions 59
11.4 Number bases 61
11.5 Addresses 61
11.6 Data initialisation 62
12 Introduction to program design 64
12.1 Structured programming 64
12.2 Functional Decomposition 65
12.2.1 Functional decomposition example 66
12.3 Writing pseudocode 70
13 The standard library 71
13.1 Standard functions 71
13.2 Function interface 72
13.3 Simple examples 73
13.4 Functions with no arguments 75
13.5 Functions with no return value 77
13.6 Functions with a variable argument list 79
13.7 Using arguments for output – "pass by reference" 79
14 File input and output 80
14.1 A simple example 80
14.2 Text and binary files 81
14.3 Standard IO streams 82
15 Modular program design 84
15.1 Modular design example 84
15.2 Choosing modules 84
15.3 Module hierarchy 86
15.4 Module specification 86
16 Introduction to function implementation 87
16.1 A simple example 87
16.2 A more detailed example 89
16.2.1 Function declaration 89
16.2.2 Function definition 89
16.2.3 Function call 90
16.2.4 Result 91
17 Calling functions 93
18 Function arguments and return values 99
18.1 Processing arguments 99
18.2 Local variables 101
18.3 Return values 101
19 Data scope and storage class 103
19.1 Local and global data 103
19.2 Automatic and static data 104
20 Pass by value and pass by reference 106
20.1 Formal and actual arguments 106
20.2 Pass by value 106
20.3 Pass by reference 110
21 Pass by reference 111
21.1 Array input to functions 111

© S.P. Platt 2001-2005

21.2 Array output from functions 114
21.3 Passing constant data by reference 116
21.4 Multiple scalar output data from functions 117
22 Testing and debugging programs 119
22.1 Testing 119
22.1.1 A simple example 119
22.1.2 Choosing a test set 120
22.1.3 Test programs 122
22.2 Debugging 126
23 Splitting programs among several source code files 128
23.1 A familiar example 128
23.2 What goes where? 131
23.3 Compiling and linking code in separate files 131
23.4 Naming conventions 133
23.5 Reusing object code 133
23.6 What’s the point? 134
24 Robust IO 136
24.1 Robust string IO example 136
24.2 Robust numerical input 140
25 Appendices 144
25.1 C99 reserved words 144
25.2 Principal C data types 144
25.3 Principal formatted IO conversion specifications 144
25.4 Principal C operations 145
25.5 Boolean operation truth tables 145
25.6 Principal format modifiers 146
25.7 Principal special characters (“escape sequences”) 146
25.8 ASCII character set 147
25.9 DOS character set 147
25.10 Selected ANSI C standard functions 148
25.11 Pass by value / pass by reference comparison 151
25.12 Principal file IO modes 151
25.13 Common C programming errors 152
25.14 Function definition checklist 152
25.15 Code style guidelines 153
25.16 Pseudocode guidelines 155
25.17 Selected DOS/Windows console commands 156
25.18 Selected glossary 157
25.19 Bibliography 157

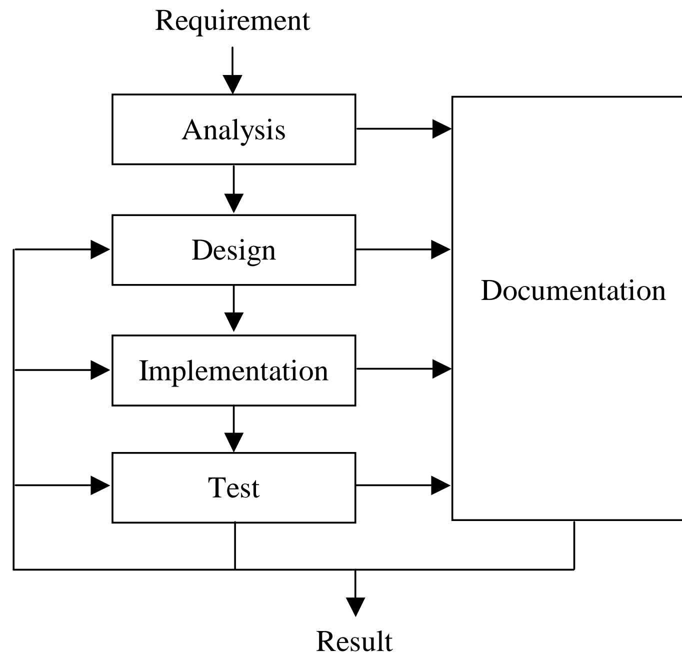
© S.P. Platt 2001-2005

1 Introduction

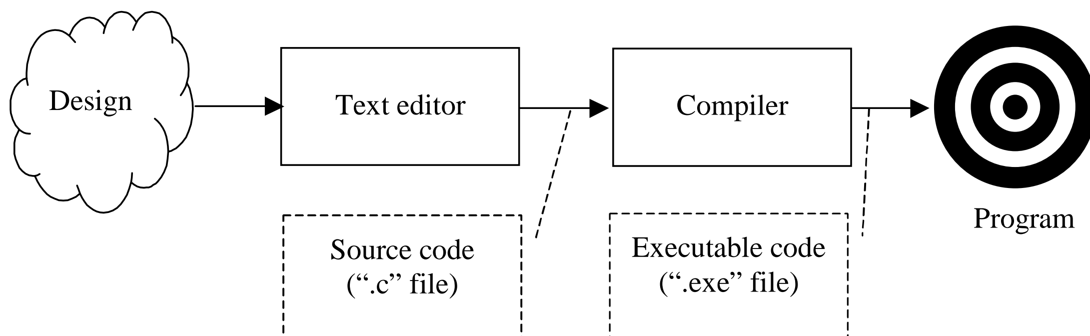
- A computer program is a set of *instructions* given to a computer.
- All programs process *data*, e.g.
    - temperatures
    - names and addresses
    - prices
    - voltages and currents
- There are many programming applications, e.g.:
    - web servers, browsers, scripts
    - databases
    - signal processing
    - engineering simulation
    - machine control
- **Program design** is the process of ensuring that your instructions to the computer are:
    - correct
    - reliable
    - efficient
    - intelligible
- This means
    - Choosing the right data
    - Choosing the right operations
    - Choosing the right sequence of events
- **Program implementation** is the process of writing instructions that can be understood by a computer using a *programming language*.
- A programming language is a precisely defined set of instructions that can be
    - *Understood* (and therefore written) by people ("high-level")
    - *Translated* into a form that can be used to control the operation of a computer ("low-level").
- There are many high-level computer programming languages, e.g.:
    - Pascal
    - FORTRAN
    - Ada
    - C
- We shall use C
    - C is very widespread in industry in a range of applications
    - "ANSI C" defines a specific standard version of C
        - Last revised in 1999, hence "C99"
    - Can be used to learn almost all programming techniques.
    - Basis of many other, more complex or more specialised, languages.

© S.P. Platt 2001-2005

- The PDI process:



  - **Analysis**: specify *what* the program must do
  - **Design**: define *how* it will be done
  - **Implementation**: write the program in "source code" and translate into "executable code"
  - **Test**: confirm that the program meets its requirement
  - **Documentation**: explain how it works
- You need to use two programming tools:
  - A **text editor** (e.g. TextPad): to create a *source code* (".c") *file*.
  - A **compiler** (e.g. bcc32): to create an *executable* (".exe") *file*.



© S.P. Platt 2001-2005

- What ANSI C source code looks like:

```
/*
 * helloworld.c
 * A minimal C program in the traditional style.
 * S.P.Platt Last updated 27/6/2003
 */
#include <stdio.h>

int main(void)
{
      puts("Hello, World!");

      return 0;
}
```

- What executable code looks like when you try to read it into a word processor:


```
MZP ┐   ┘ ☼ ÿÿ  ¸        @ →                            ┐  º┼ ´
      Í!¸┌LÍ!  This program must be run under Win32
 $7




                    PE  L┌◘ ôlü>        à └
┌│      0       ┼   ┼        @  ┼   ┐  ┘       ┘
┌  ─       └      ┼
   ┼  ┼      ┼    ð  n    à  D┘    ┌  ┐                 ┼┌ œ•
                Ð  ↑                                                 .tex
t        ┼   z  ─
  `.data    0       $                  @  À.tls     ┼   À   ┐   ¤
    @  À.rdata   ┼   Ð   ┐   ¦              @  P.idata   ┼   à   ─   ¨
         @  @.edata   ┼   ð   ┐   ®              @  @.rsrc    ┼    ┌  ┐   º
              @  @.reloc   ┼   ┼┌  ◘   ²              @  P





                                                                   ë┼fb:C++HOO
K é '@ ¡☼'@ Áà┐£‼'@ Rj èqw  <ÐèŠ┼  Zè
┘  èƒ┼  j è"   Yh¸ @ j èKw  £┤'@ j ékY  éÂ   3À ┌'@ Ã¡┤'@ Ã`» Pº¼Sh-
  Ã¹œ
ÉtMƒ=☼'@  s ¸þ   è×ÿÿÿ¹œ   Qj◘è◘w  Pè,w
Àu ¸ý   è¶ÿÿÿPPÿ5☼'@ èÍZ  ÿ5☼'@ èÖZ  _Ã¹œ
Ét├èŠZ  £☼'@ ƒø s'¸ü   èzÿÿÿÃƒ=☼'@  r(ÿ5☼'@ èyZ
Àt├Pj◘è—
v  PèÁv  ÿ5☼'@ è^Z  ÃÃƒ=☼'@  r┼è¿ÿÿÿÿ5☼'@ è1Z  Ã¡☼'@ dg<┬, <┘,Ã U<ìh('@ è
'  Y3À]Ã  U<ìSV<u◘☼¯uVèî┌  Y<Ø…ÀtVj Sèª☼  ƒÄ<Ã^[]Ã U<ì<E◘PèØ┌  Y]Ã U<ì<E◘Pè
äÿÿÿY]Ã SVW<ù<òƒ=8'@  u&<┴Ü»@ ƒú┐t│ƒú┌uÇ│8'@   ┼ ë Ç│8'@   @ <┴8'@ <Ű└ØKJ÷Õ
#Űj┌h    Sj è v  ‰¯…Àu┘3Àë•‰-
¸┌   _^[Ã  SVW<ú<Øj┘h ┼  W<└Pèïu  <ð‰3…ö☼•Àƒà┌_^[Ã SVW<ò<Ø…ö<ût5j┘h ┼  h ┼
 Sè¿u  …Àu☼<Ó<Ç+×è-   3Àë┴ Ã ┼   î ┼  …öuË¸┌   _^[Ã h @  RPèŒu  Ht└3ÀÃ¸┌
Ãh    j Pèsu  Ht└3ÀÃ¸┌   Ã   h<'@ è↕u  ;│ ´@ t
hE'@ èXI  YÃ  U<ìSVW<}<u◘…öt …ÿt−ƒ}┼ u┘3ÀëYhu'@ èŒt  <Ø…Ûu┘3ÀëEh~'@ Sèƒt  ‰
−h—'@ Sèvt  ‰•h²'@ Sèit  <U┼‰┐ƒ> t
ƒ? t◘<M┼ƒ9 u┘3Àë│¸┌   _^[]Ã3ÀÃ U<ì<E◘Pÿ┴è'@ Y]ÃU<ì<E◘Pÿ┴ä'@ Y]ÃU<ì<EP<U◘Rÿ┴
ì'@ ƒÄ◘]Ã  U<ì輿ÿÿ…Àu2Ç│ä'@ <→@ Ç│è'@ ¬←@ Ç│ì'@ ┘
@ Ç│ð'@ Ä @ Ç│à'@ ┌   <E◘£Ü'@ ]ÃU<ìj┌è®ÿÿÿY<E◘Pÿ┴ä'@ Y]ÃU<ìj┌è-
ÿÿÿY<E◘Pÿ┴è'@ Y]ÃU<ìj┌è~ÿÿÿY<EP<U◘Rÿ┴ì'@ ƒÄ◘]Ã  Ã   Ç│Ô'@ ┌   Ã ƒ=Ô'@  t−ÿ┴
ð'@ ÃÃ   º   ë┬<Â└À└│↑'@ ƒÀô‰@┘‰@◘ƒÂ┘;┴┘'@ │â¡┘'@ 3Ò└À└│↑'@ ƒÀô3É‰P┘‰H◘ÃU<ì
SVW<}<U◘<Â3ö ÿ ┼  s◘ƒÈÿéÚ   Ç@◘┌   ‰P‰8<U┼‰P┘3Ò
```

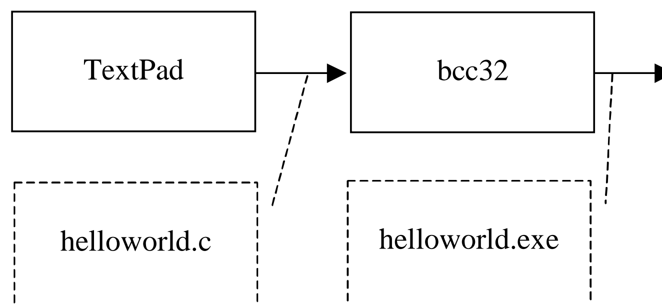
2 Getting started

2.1 A first program

- helloworld source code

```
/*
 * helloworld.c
 * A minimal C program in the traditional style.
 * S.P.Platt Last updated 27/6/2003
 */
#include <stdio.h>

int main(void)
{
      puts("Hello, World!");

      return 0;
}
```

- "Comments"
    - Text enclosed between `/*` and `*/`
    - Text following `//`
    - Not translated to executable code – give extra information to reader
- "Main function"
    - All C programs have one
    - `int main(void)` starts it
    - What the program actually does goes between { ("begin") and } ("end").
    - `return 0;` goes last
    - This program will **put** the *string* (`puts`) "Hello, World!" on the computer screen.
- "#include"
    - `puts` is a function from the standard IO (input/output) library.
    - stdio.h is the "standard IO header file"
    - `#include <stdio.h>` makes standard IO functions available
    - Nearly all programs need this at the top
- Source code is helloworld.c:
    - Written in a *text editor* – e.g. TextPad
- Executable code is helloworld.exe
    - Translated from helloworld.c by a *compiler* – e.g. bcc32

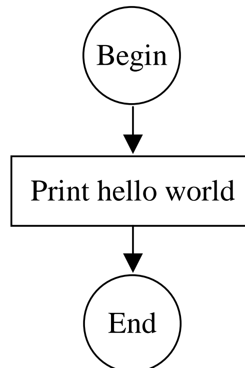


- Compilation process (simplified):
  - Checks *source code* for syntax (language) errors.
  - If there are none, translates source code to *machine instructions*.
  - Machine instructions finally stored in *executable file*.
- At the command prompt:
  - Enter "`type helloworld.c`" to list source code file
  - Enter "`bcc32 helloworld.c`" to run the compiler (bcc32), checking helloworld.c and translating to executable file, helloworld.exe
  - Enter "`helloworld`" to run helloworld.exe.

```
prompt> type helloworld.c
/*
 * helloworld.c
 * A minimal C program in the traditional style.
 * S.P.Platt Last updated 27/6/2003
 */
#include <stdio.h>

int main(void)
{
      puts("Hello, World!");

      return 0;
}
prompt> bcc32 helloworld.c
Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland
helloworld.c:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

prompt> helloworld
Hello, World!

prompt>
```

2.2 Design notation

- Programs are written ("coded") very *formally* in C
- Pseudocode ("pretend code") is an *informal* description in *natural language*
- Pseudocode for helloworld:
  Print hello world.
- Flowcharts provide *pictures* of program operation
- Flowchart for helloworld:



- Pseudocode and flow charts are used to:
  - Make design *process* easier
  - *Document* a design
- Pseudocode is nearly always better than flow charts.

2.3 Program errors

- Two sorts:
  - Syntax errors – errors of language, detected by the compiler
    - Prevent creation of executable file
  - Run-time errors – logical errors or "bugs"
    - Cause program to crash
    - Cause program to operate incorrectly

2.3.1 Syntax errors

- C has strict language rules, e.g.
  - All "statements" (nearly all lines) must end in "`;`"
  - All brackets `()`, `[]`, or `{}`, must come in matching pairs, as must comment delimiters, `/* */`

© S.P. Platt 2001-2005

- Spot the syntax error:

```
prompt> type helloworld.c
/*
 * helloworld.c
 * A minimal C program in the traditional style.
 * S.P.Platt Last updated 27/6/2003
 */
#include <stdio.h>

int main(void)
{
      puts("Hello, World!")

      return 0;
}
prompt> bcc32 helloworld.c
Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland
helloworld.c:
Error E2379 helloworld.c 12: Statement missing ; in function main
*** 1 errors in Compile ***

prompt>
```

2.3.2 Run time errors

- It's easy to write a valid program that doesn't do what you wanted:

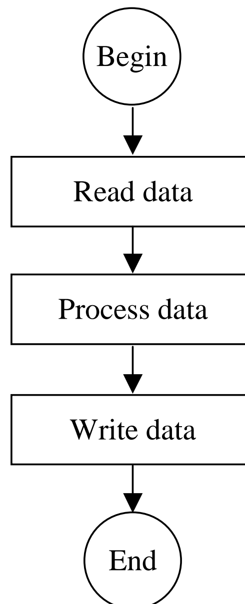
```
prompt> type helloworld.c
/*
 * helloworld.c
 * A minimal C program in the traditional style.
 * S.P.Platt Last updated 27/6/2003
 */
#include <stdio.h>

int main(void)
{
      puts("Goodbye, World!");

      return 0;
}
prompt> bcc32 helloworld.c
Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland
helloworld.c:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

prompt> helloworld
Goodbye, World!
prompt>
```

References (see p. 157)

- Dawson:
  - A1
- Kernighan and Ritchie:
  - 1.1

3 Using data – characters and strings

- All useful programs must process some data
- We do three sorts of things with data:
  - Read it in (from the keyboard or a file or other source)
  - Process it (e.g. do arithmetic)
  - Write it out (to the screen or a file or other location)
- Programs often follow the input/process/output pattern:



- There are three basic data types
  - Integers – numbers with no fractional part (0, 1, 2, -45, 999 etc.).
    - Used mainly for counting
  - "Floating point" numbers – those with fractional parts (1.234, -3.56, 10.0, etc.)
    - Generally used for calculation
  - Characters – character symbols ('0', 'a', 'Z', '!' etc.).

3.1 Variables

- A *variable* is an item of data whose value may change during program execution
- Variable *definition* or *declaration*:
  - Gives a variable a name
  - Gives it a type (integer etc.)
  - Reserves space for it in memory
- Example declarations
  - `int count;` declares an integer called `count`
  - `float length;` declares a floating-point number called `length`
  - `char symbol;` declares a character called `symbol`

© S.P. Platt 2001-2005

- Choosing variable names:
    - Use only *letters*, *digits* and '_' ("underscore"): `len2`✓, `len_2`✓, `len 2`✗
    - Avoid *reserved words* (see p. 144): `ch`✓, `char`✗
    - Start with a *letter*: `firstgo`✓, `1stgo`✗
    - Make them *meaningful*: `TCel`✓, `C`✗
    - Keep them *short*: `TCel`✓, `the_temperature_in_degrees_Celsius`✗
    - Names are *case sensitive*: `TCel` and `tcel` are different

3.2 Using characters

- Character *constants* are written in single quotes
    - 'a', 'Z', '!', etc

```
/*
  * charop.c
  * Illustrates character output using character constants
  * S.P. Platt Last modified 30/6/2003
  */
 #include <stdio.h>

 int main(void)
 {
     putchar('a');
     putchar('Z');
     putchar('!');

     return 0;
 }
```

- charop analysis:

  `putchar('a');`

    - displays ("puts") the symbol 'a' on the screen using the `putchar` function.
- charop results:

```
prompt>charop
aZ!
prompt>
```

- Character *variables* are declared with type char

```
/*
 * chario.c
 * Illustrates character input/output
 * S.P. Platt Last modified 27/08/2003
 */
#include <stdio.h>

int main(void)
{
    char ch;

    puts("Enter any character: ");
    ch=getchar();

    puts("You entered ");
    putchar(ch);

    return 0;
}
```

© S.P. Platt 2001-2005

- Pseudocode for chario

  ```
  Print prompt
  Get character
  Print character
  ```

- chario analysis:

  `char ch;`

  - defines a character variable called `ch`

  `ch=getchar();`

  - reads ("gets") the symbol for `ch` from the keyboard using the `getchar` function.

  `putchar(ch);`

  - displays ("puts") the symbol for `ch` on the screen using the `putchar` function.

- chario results:

```
prompt>chario
Enter any character:
q
You entered
q
prompt>chario
Enter any character:
qwerty
You entered
q
prompt>
```

- Alternatively:

```
/*
 * chario.c
 * Illustrates character input/output
 * S.P. Platt Last modified 27/08/2003
 */
#include <stdio.h>

int main(void)
{
    char ch;

    printf("Enter any character: ");
    scanf("%c",&ch);

    printf("You entered %c",ch);

    return 0;
}
```

- This version of chario uses functions printf and scanf
  - `printf` ("**print f**ormatted") is a general-purpose output function
  - `scanf` ("**scan f**ormatted") is a general-purpose input function
- Analysis:

  `printf("Enter any character: ");`

  - Displays the string "Enter any character: " on the screen

  `scanf("%c",&ch);`

  - `%c` tells `scanf` to read a character symbol from the keyboard
  - `&ch` tells `scanf` to store what it has read in variable `ch`.

  - The ampersand (`&`) is very important in `scanf` – your program won't work without it.

  `printf("You entered %c",ch);`

  - Displays the string "You entered " on the screen, followed by the value of variable `ch`
  - `%c` tells `printf` to print a character symbol
  - `%c` is replaced by the symbol for character `ch`
  - `%c` is a *placeholder* for a character symbol
  - Do not use an ampersand (`&`) in `printf` – or your program will fail.

- chario results:

```
prompt>chario
Enter any character:
q
You entered q
prompt>chario
Enter any character:
qwerty
You entered q
prompt>
```

3.3 The character set

- The set of symbols available to be processed on a computer is known as the *character set*
- Standard characters (chars) can take any one of 256 values (symbols)
  - It is also possible to use extended character sets to cope with mathematical symbols, non-European languages, etc.
- Most operating systems use a superset of the ASCII character set (p. 147)
  - The Windows XP command interpreter uses a character set known as the DOS character set (p. 147)
- Character constants are shown as the conventional symbol inside single quotes:
  - 'a', 'Z', '!'
- *Escape sequences* are used to represent some special characters (p. 146), e.g.
  - '\'', '\n', '\0', '\\'

3.4 Character strings

- Individual characters have limited usefulness
- Words and sentences require character *strings*
  - "Several characters strung one after the other"
- Constant strings ("string literals") are typed in double quotes
  - e.g. "Hello, world"
- Variable strings are stored in character *arrays*.
- Remember:
  - *Single* quotes for a single *character*
  - Plural (*double*) quotes for plural characters (a *string*)
- Escape sequences can be embedded in string literals, e.g.

```
printf("This statement prints a new line HERE\nand at the end.\n");
printf("This statement shows how to print double quotes (\").");
```

- Example:

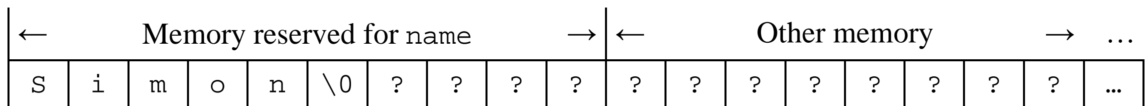
```
/*
 * greetings.c
 * Prints a friendly greeting.
 * S.P. Platt Last edited 27/6/2003
 */
#include <stdio.h>

int main(void)
{
    char name[10];

    printf("What is your name? ");
    scanf("%s",&name);
    printf("Hello, %s.",name);

    return 0;
}
```

- greetings processes a character *string*
  `char name[10];`
  - declares an *array* of 10 characters called `name`
  - `name` can store only 9 (10-1) true characters safely
  - An extra character (`'\0'`) is needed to indicate the end of the string
- greetings reads a character string from the keyboard with `scanf`
  `scanf("%s",&name);`
  - `"%s"` specifies that a string is to be read, stopping when a space or the end of a line is encountered
  - The characters are stored in the variable called `name`
  - The terminating character, '\0', is added to the end of the string
- greetings displays a character string on the screen with `printf`
  `printf("Hello, %s.",name);`
  - `"%s"` is a placeholder for the string
  - The contents of `name` are written instead of `%s`
- greetings uses three strings:
  - `"What is your name? "` – constant
  - `name` – variable
  - `"Hello, %s."` – constant
- greetings results:

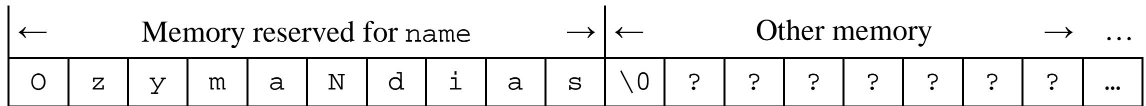
```
prompt> greetings
What is your name? Simon
Hello, Simon.
prompt> greetings
What is your name? Simon Platt
Hello, Simon.
prompt> greetings
What is your name? Ozymandias
Hello, Ozymandias.
prompt> greetings
What is your name? Slartybartfast
Hello, Slartybartfast.
```

© S.P. Platt 2001-2005

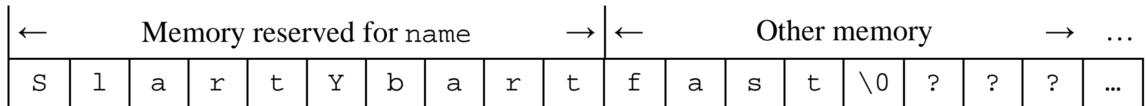
- When "Simon" is entered:



- When "Ozymandias" is entered:



- When "Slartybartfast" is entered:



- Unpredictable things happen when too many characters are read in to a string
  - Programs nearly always fail
  - Programs often crash

References

- Dawson:
  - A2.1-2.3, 2.6, 2.7, 4.1-4.3
- Kernighan and Ritchie:
  - 1.5, 2.1-2.4

© S.P. Platt 2001-2005

4 Using numerical data

- There are two categories of numbers used in C programs:
  - Integers (whole numbers)
  - Floating-point numbers (those with fractional parts)

4.1 Using integers

- Example: converting Fahrenheit to Celsius (FtoC)
- FtoC Pseudocode

```
Get Fahrenheit temperature
Calculate Celsius temperature
Print Celsius temperature
```

- FtoC processes two items of data
  - Fahrenheit temperature
  - Celsius temperature
- FtoC source code listing

```
/*
 * FtoC.c
 * Converts temperature from Fahrenheit to Celsius.
 * S.P. Platt Last modified 27/8/2003
 */
#include <stdio.h>

int main(void)
{
    int TCel,TFar;       //Fahrenheit and Celsius temperatures

    printf("Program FtoC converts Fahrenheit to Celsius.\n\n");

    printf("Enter a Fahrenheit temperature: ");
    scanf("%d",&TFar);

    TCel=(TFar-32)*5/9;

    printf("%d degrees F = %d degrees C",TFar,TCel);

    return 0;
}
```

- FtoC.c analysis:

  `int TCel,TFar;       //Fahrenheit and Celsius temperatures`
  - declares two integers, `TCel` and `TFar`

  `printf("Program FtoC converts Fahrenheit to Celsius.\n\n");`
  - `printf` writes *string*
  - `\n` starts a new line (see also p. 146)

  `scanf("%d",&TFar);`
  - `scanf` reads data from keyboard
  - `%d` specifies integer input (see p. 144)
  - `&TFar` specifies that the value will be stored in variable `TFar`
  - The ampersand (`&`) is necessary

`TCel=(TFar-32)*5/9;`

- Calculates Celsius temperature and stores it in `TCel`
    - `TFar-32` – subtraction
    - `*5` – multiplication
    - `/9` – division
    - `=` – assignment
    - `(`whatever's in the brackets is done first`)`

`printf("%d degrees F = %d degrees C",TFar,TCel);`

- `printf` writes *formatted string*
- `printf` writes data to screen
- `%d` is a *conversion specification* (placeholder for formatted output)
- `printf` writes values of `TFar` and `TCel` instead of `%d`s.

- FtoC results:

```
prompt> FtoC
Program FtoC converts  Fahrenheit to Celsius.

Enter a Fahrenheit temperature: 32
32 degrees F = 0 degrees C
prompt> FtoC
Program FtoC converts  Fahrenheit to Celsius.

Enter a Fahrenheit temperature: 212
212 degrees F = 100 degrees C
prompt> FtoC
Program FtoC converts  Fahrenheit to Celsius.

Enter a Fahrenheit temperature: 100
100 degrees F = 37 degrees C
prompt>
```

- Checking FtoC results:
    - 32°F correctly converted to 0°C✓
    - 212°F correctly converted to 100°C✓
    - 100°F incorrectly converted to 37°C✗ – should be 37.8°C
    - floats would be more suitable than ints for `TCel` and `TFar`.

© S.P. Platt 2001-2005

4.2 Using floating-point numbers

- Improved FtoC source code listing:

```
/*
 * FtoC.c
 * Converts temperature from Fahrenheit to Celsius.
 * S.P. Platt Last edited 27/8/2003
 */
#include <stdio.h>

int main(void)
{
    float TCel,TFar;       //Fahrenheit and Celsius temperatures*/

    printf("Program FtoC converts Fahrenheit to Celsius.\n\n");
    printf("Enter a Fahrenheit temperature: ");
    scanf("%f",&TFar);

    TCel=(TFar-32)*5/9;

    printf("%f degrees F = %f degrees C",TFar,TCel);

    return 0;
}
```

- Analysis:

  `float TCel,TFar;       //Fahrenheit and Celsius temperatures`
  - defines `TCel` and `TFar` as floating point numbers

  `scanf("%f",&TFar);`
  - reads value of `TFar` data from keyboard
  - `"%f"` specifies floating-point input (see p. 144)

  `TCel=(TFar-32)*5/9;`
  - Calculates value of `TCel`. Fractions are not lost.

  `printf("%f degrees F = %f degrees C",TFar,TCel);`
  - Displays values of `TFar` and `TCel`: `%f` is a placeholder for a floating-point number.
- Results:

```
prompt> ftoc
Program FtoC converts Fahrenheit to Celsius.

Enter a Fahrenheit temperature:
100
100.000000 degrees F = 37.777779 degrees C
prompt>
```

4.3 Integer and floating-point arithmetic

- The basic arithmetic operations are:
  - division /
  - multiplication *
  - addition +
  - subtraction -
- There is one other – the modulus operator, %

- For integers, division *throws away* any remainder:
  - The result of 5/2 is 2 (5÷2 = 2 ~~remainder 1~~)
  - The result of 5/9 is 0 (5÷9 = 0 ~~remainder 5~~)
- The remainder can be found with the *modulus* operator:
  - The result of 5%2 is 1 (5÷2 = ~~2~~ remainder 1)
- For floating point numbers, division *keeps* any remainder
  - The result of 5.0/2 is 2.5
  - The result of 5.0/9 is 0.555556
- The modulus operator is not used with floating-point arithmetic
- Floating-point calculations are more precise than integer calculations
  - Or are they?
- Floating point calculations are often *not* precise
  - They suffer from *roundoff error*
  - 5÷9 ≠ 0.555556
  - 5÷9 = 0.555555555555555555555555555555555555555555555555555555555555…

4.4 Single and double-precision floating-point numbers

- Single-precision floating point numbers (floats) are accurate to about 6 significant decimal digits
  - 5/9 becomes 0.555556
- Double-precision floating point numbers (doubles) are accurate to about 15 significant decimal digits
  - 5/9 becomes 0.555555555555556
  - The roundoff error is much reduced in double-precision arithmetic
- Double-precision example:

```
/*
 * circle.c
 * Calculates circumference and area of a circle.
 * S.P. Platt Last edited 27/6/2003
 */
#include <stdio.h>

int main(void)
{
    double radius, circumf, area;

    printf("Enter a radius: ");
    scanf("%lf",&radius);

    circumf = 2*3.14159265358979*radius;
    area = 3.14159265358979*radius*radius;
    printf("Radius %f, circumference %f, area %f",radius, circumf, area);

    return 0;
}
```

- Analysis:

  `double radius, circumf, area;`

  - Three double precision variables are declared: `radius`, `circumf` and `area`.

```
scanf("%lf",&radius);
```

- `radius` is read using `scanf`.
- `%lf` is used to read a double, *not* `%f`

```
circumf = 2*3.14159265358979*radius;
area = 3.14159265358979*radius*radius;
```

- The values of `circumf` and `area` are calculated

```
printf("Radius is %f, circumference is %f, area is %f",
        radius, circumf, area);
```

- The values of `radius`, `circumf` and `area` are displayed using `printf`.
- **`%f`** is used to display a double

- Results:

```
prompt> circle
Program circle calculates circumference and area of a circle.

Enter a radius: 1
Radius is 1.000000, circumference is 6.283185, area is 3.141593
prompt> circle
Program circle calculates circumference and area of a circle.

Enter a radius: 5
Radius is 5.000000, circumference is 31.415927, area is 78.539816
prompt>
```

- Note that the full precision of the results is not displayed.
  - It is almost never necessary for *final* results to be displayed to full precision.
- Many scientific and engineering applications are very sensitive to small errors in *intermediate* calculations
  - It is important to make intermediate calculations much more precise that the final results.
- doubles are often used in C programs in preference to floats, to *minimise errors*.
  - This is particularly important when large numbers of intermediate calculations are needed
- floats are sometimes used to *save memory*
  - A double normally takes twice as much memory to store as a float.

References

- Dawson:
  - A2.1-2.3, 3.1-3.4
- Kernighan and Ritchie:
  - 1.2, 2.1-2.5

5 Formatted Input/Output (IO)

- We have already seen several ways of doing IO
  - `getchar` is a standard function to **get** a single **char**acter from the keyboard
  - `putchar` is a standard function to **put** a single **char**acter on the screen
  - `puts` is a standard function to **put** a character **s**tring on the screen
    - There is also a function called "`gets`" – it has a bug, so don't use it.
  - `scanf` is a standard function to **scan f**ormatted data from the keyboard
    - "Formatted data" means that you specify how input is to be interpreted, e.g. whether '1' typed at the keyboard means the character '1' or the numeral 1.
  - `printf` is a standard function to **print f**ormatted data on the screen
    - "Formatted data" means that you specify how output is to be presented, e.g. whether the number 1.0 should de displayed as "1", "1.0", or "1.000000".
- `%c`, `%d`, `%f` etc. (pp. 144, 146) are the format specifiers used by printf and scanf.
- Put `#include <stdio.h>` at the top of your program to make IO possible

5.1 Formatted input

- Use standard function `scanf` to read data into a program from the keyboard.
- The operation of `scanf` is controlled by *format strings*:
  - **`%d`** reads an int: `scanf("%d",&TFar);   //assumes TFar an int`
  - **`%f`** reads a float: `scanf("%f",&TFar);   //assumes TFar a float`
  - **`%lf`** reads a double: `scanf("%lf",&TFar);  //assumes TFar a double`
  - **`%c`** reads a char: `scanf("%c",&ch);`
  - **`%s`** reads a string: `scanf("%s",&name);`
- The ampersand is required in front of the variable name in `scanf`
  - It's not *absolutely* necessary for a string, but it does no harm to leave it in
  - It *is* absolutely necessary for all other types of input – your program will fail without it

© S.P. Platt 2001-2005

- It is easy to read several things at once with `scanf`:
  - `scanf("%f%f",&length,&width);` reads length and width in one go
- Example:

```
/*
File:         rectangle.c
Author:       S.P. Platt
Last edited: 27/08/2003
Description: Calculates the area of a rectangle
*/

#include <stdio.h>

int main(void)
{

   float length,width,area;

   printf("Program rectangle calculates the area of a rectangle.\n");

   printf("Enter length and width: ");
   scanf("%f%f",&length,&width);
   area = length*width;

   printf("The area is %.2f",area);

   return 0;
}
```

- Results:

```
prompt> rectangle
Program rectangle calculates the area of a rectangle.
Enter length and width: 10 5
The area is 50.00
prompt>
```

- To control string input better, specify the maximum number of characters read with a *field width*:
  - `scanf("%9s",&name);` reads 9 characters at most
- Example:

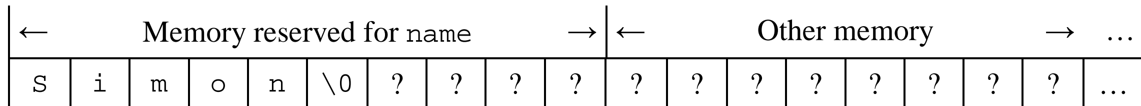
```
/*
 * greetings.c
 * Prints a friendly greeting.
 * S.P. Platt Last edited 27/8/2003
 */
#include <stdio.h>

int main(void)
{
    char name[10];

    printf("What is your name? ");
    scanf("%9s",&name);
    printf("Hello, %s.",name);

    return 0;
}
```

© S.P. Platt 2001-2005

- Results:

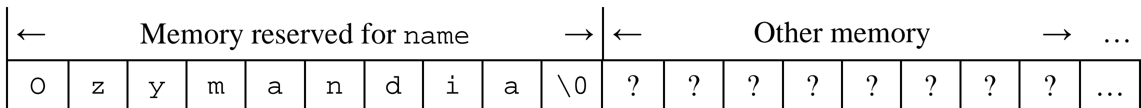
```
prompt> greetings
What is your name? Simon
Hello, Simon.
prompt> greetings
What is your name? Ozymandias
Hello, Ozymandia.
prompt> greetings
What is your name? Slartybartfast
Hello, Slartybar.
prompt>
```

- When “Simon” is entered:



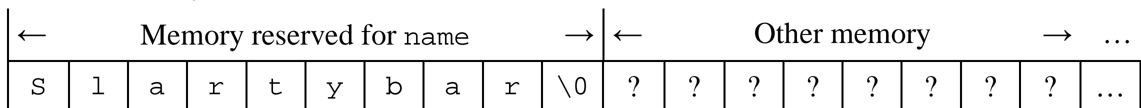
← Memory reserved for `name` →										← Other memory →								…
S	i	m	o	n	\0	?	?	?	?	?	?	?	?	?	?	?	?	…

- When “Ozymandias” is entered:



← Memory reserved for `name` →										← Other memory →								…
O	z	y	m	a	n	d	i	a	\0	?	?	?	?	?	?	?	?	…

- When “Slartybartfast” is entered:



← Memory reserved for `name` →										← Other memory →								…
S	l	a	r	t	y	b	a	r	\0	?	?	?	?	?	?	?	?	…

- It is very important to control the number of characters read into a string:
  - If you try to read more characters than the number of spaces available (e.g. 9, in this example), you will overwrite some other memory
  - Your program might crash if you try to write to some memory you shouldn’t.
  - Even if it doesn’t crash, you will lose whatever data was previously stored on the other memory – and your program is likely to go wrong in some less obvious way.
- `scanf` is a very literal-minded function. Be careful not to put unwanted characters into its format string, by mistake:
  - `scanf("%d,%d",&length,&width);` expects to see a comma like this:

    ```
    Enter length and width: 10,5
    ```

    and will fail if it doesn’t.

5.2 Formatted output

- Use standard function `printf` to write data from a program to the screen.
- The operation of `printf` is controlled by the same format strings `%d`, `%f` etc.
- To print ‘%’, use `%%` in `printf`:
  - `printf("The percentage is %d%%.",result);`
- To control output better, specify *field widths* and *precision*:
  - `printf("%4d",NTemp);` prints value of int `NTemp` in a field 4 characters wide
  - `printf("%.2f",TFar);` prints value of float (or double) `TFar` to a precision of two decimal places
  - `printf("%6.1f",TFar);` prints value of float (or double) `TFar` to one decimal place in a field 6 characters wide

- Example:

```
/*
 * FtoC.c
 * Converts temperature from Fahrenheit to Celsius.
 * S.P. Platt Last edited 27/8/2003
 */
#include <stdio.h>

int main(void)
{
      float TCel,TFar;

      printf("Program FtoC converts Fahrenheit to Celsius.\n\n");

      printf("Enter a Fahrenheit temperature: ");
      scanf("%f",&TFar);

      TCel=(TFar-32)*5/9;

      printf("%.1f degrees F = %.1f degrees C",TFar,TCel);

      return 0;
}
```

- Results:

```
prompt> ftoc
Program FtoC converts Fahrenheit to Celsius.

Enter a Fahrenheit temperature: 100
100.0 degrees F = 37.8 degrees C
prompt>
```

5.3 Characters are really numbers

- The character set relates character symbols to numbers
  - Character number 65 has symbol ‘A’ (in most character sets)
  - Character number 94 has symbol ‘^’
- All characters can be presented as symbols or numbers.
- charindex example

```
/*
 * charindex.c
 * Shows the integer equivalent of a character symbol
 * S.P. Platt Last edited 27/8/2003
 */
#include <stdio.h>

int main(void)
{
    char ch;

    printf("Program charindex compares character and\n");
    printf("integer representations of the same data.\n\n");

    printf("Enter any character ");
    scanf("%c",&ch);
    printf("The integer representation of '%c' is %d",ch,ch);

    return 0;
}
```

© S.P. Platt 2001-2005

- Pseudocode for charindex

```
Print title
Get character
Print character and integer representations
```

- charindex uses character data

```
char ch;
```

- Characters are *stored* as numbers
    - ASCII character set used on PCs (see p. 147)
- Characters can be read as
    - Symbols (`%c` in `scanf`)

```
scanf("%c",&ch);
```

    - Numbers (`%d` in `scanf`)
- Characters can be printed as
    - Symbols (`%c` in `printf`)
    - Numbers (`%d` in `printf`)

```
printf("The integer representation of '%c' is %d",ch,ch);
```

- Charindex results:

```
prompt> charindex
Program charindex compares character and
integer representations of the same data.

Enter any character A
The integer representation of 'A' is 65
prompt> charindex
Program charindex compares character and
integer representations of the same data.

Enter any character z
The integer representation of 'z' is 122
prompt> charindex
Program charindex compares character and
integer representations of the same data.

Enter any character !
The integer representation of '!' is 33
prompt> charindex
Program charindex compares character and
integer representations of the same data.

Enter any character 1
The integer representation of '1' is 49
prompt>
```

- Checking charindex results (see p. 147):
    - Number 65 corresponds to symbol ‘A’
    - Number 122 corresponds to symbol ‘z’
    - Number 33 corresponds to symbol ‘!’
    - *Number* 49 corresponds to *symbol* ‘1’

© S.P. Platt 2001-2005

5.4 Robust IO

- IO very often goes wrong, because the actions of things outside the program (e.g., the user at the keyboard) are outside the programmer's control (e.g. users often make typing mistakes).
- There are techniques for robust IO – checking input (sometimes also output) actions and doing something sensible if something goes wrong
  - e.g. discarding invalid data and trying again
- We shall cover some of these techniques later on (p. 136).

References

- Dawson:
  - A4.4-4.6
- Kernighan and Ritchie:
  - 7.2, 7.4

© S.P. Platt 2001-2005

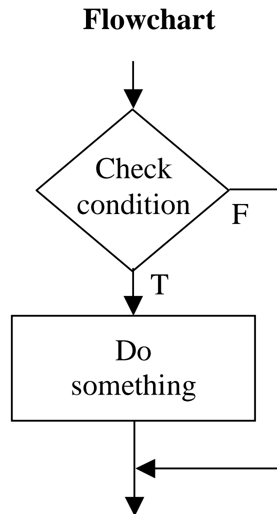
6 Selection between two alternatives – if and if-else

- *Conditional statements* execute only if some condition is true
- *Boolean operations* determine whether a condition is true (T) or false (F)

6.1 If

- If is used to do something *conditionally*.

Flowchart



Pseudocode

```
if condition
      do something
endif
```

C syntax

```
if (condition)
{
      do_something;
}
```

N.B.

- `condition` goes in brackets
- No semicolon after `(condition)`
- Braces ({ }) mark beginning and end

- Example pseudocode:

```
get mark
if mark>=40
      print pass message
end if
```

- Example source code:

```
/*
 * passfail.c
 * Determines pass/failure of a module
 * S.P. Platt Last modified 27/08/03
 */

#include <stdio.h>

int main(void)
{
    int mark;

    printf("Program passfail determines whether ");
    printf("a module has been passed.\n\n");

    printf("Enter the mark (0-100%%): ");
    scanf("%d",&mark);

    if (mark>=40)
    {
        printf("Student has passed (%d%%).\n",mark);
    }

    return 0;
}
```

© S.P. Platt 2001-2005

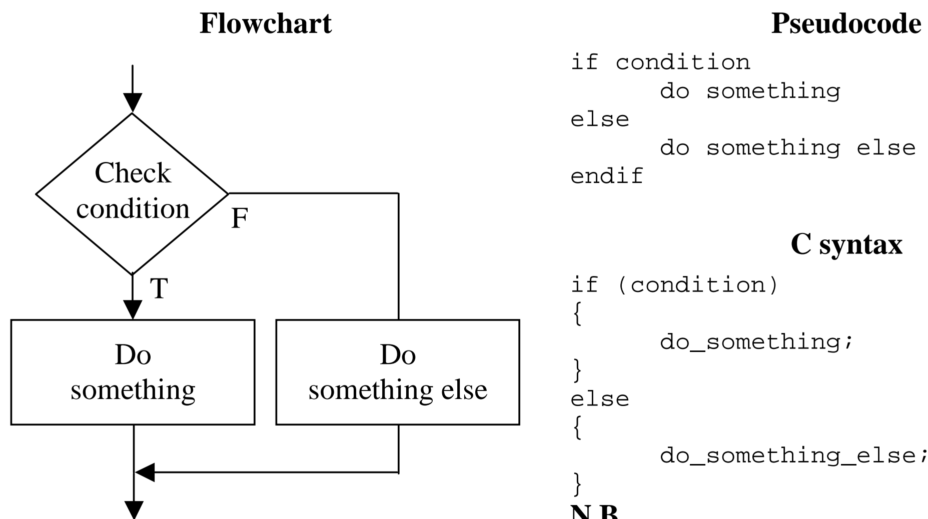
- Example results:

```
prompt>passfail
Program passfail determines whether a module has been passed.

Enter the mark (0-100%): 39

prompt>passfail
Program passfail determines whether a module has been passed.

Enter the mark (0-100%): 40
Student has passed (40%).

prompt>
```

6.2 If-else

- If-else is used to select between alternative operations.

Flowchart



Pseudocode

```
if condition
      do something
else
      do something else
endif
```

C syntax

```
if (condition)
{
      do_something;
}
else
{
      do_something_else;
}
```

N.B.

- No semicolon after `else`
- Use braces ("curly brackets")

- Example pseudocode

```
get mark
if mark>=40
      print pass message
else
      print fail message
end if
```

- Example source code

```
/*
 * passfail.c
 * Determines pass/failure of a module
 * S.P. Platt Last modified 27/08/03
 */

#include <stdio.h>

int main(void)
{
    int mark;

    printf("Program passfail determines whether ");
    printf("a module has been passed.\n\n");

    printf("Enter the mark (0-100%%): ");
    scanf("%d",&mark);

    if (mark>=40)
    {
        printf("Student has passed (%d%%).\n",mark);
    }
    else
    {
        printf("Student has failed (%d%%).\n",mark);
    }

    return 0;
}
```

- Example results:

```
prompt>passfail
Program passfail determines whether a module has been passed.

Enter the mark (0-100%): 39
Student has failed (39%).

prompt>passfail
Program passfail determines whether a module has been passed.

Enter the mark (0-100%): 40
Student has passed (40%).

prompt>
```

6.3 Boolean values

- Boolean operations are about applying *logic* in a computer program.
- The result of a boolean operation is either “true” or “false”.
  `mark>=40`
  - “evaluates to true” (1) if `mark` is greater than or equal to 40
  - “evaluates to false” (0) otherwise.
- Result can be stored in an integer variable
  - False is represented by 0
  - True is represented by 1 (actually, any number except 0, but 1 is conventional)
- This is especially useful if the result of the same test is used several times.

© S.P. Platt 2001-2005

- Example:

```
/*
 * passfail.c
 * Determines pass/failure of a module
 * S.P. Platt Last modified 27/08/03
 */
#include <stdio.h>

int main(void)
{
    int mark;
    int pass;     //boolean variable

    printf("Program passfail determines whether ");
    printf("a module has been passed.\n\n");

    printf("Enter the mark (0-100%%): ");
    scanf("%d",&mark);

    pass = (mark>=40);
    printf("The value of pass is %d\n",pass);
    if (pass)
    {
        printf("Student has passed (%d%%).\n",mark);
    }
    else
    {
        printf("Student has failed (%d%%).\n",mark);
    }

    return 0;
}
```

- Results:

```
prompt>passfail
Program passfail determines whether a module has been passed.

Enter the mark (0-100%): 39
The value of pass is 0
Student has failed (39%).

prompt>passfail
Program passfail determines whether a module has been passed.

Enter the mark (0-100%): 40
The value of pass is 1
Student has passed (40%).

prompt>
```

© S.P. Platt 2001-2005

6.4 Making comparisons

- The following are true (evaluate to 1)
    - `4.3 > -23`
    - `2 != 5`
    - `my_age >= 21`
- The following are false (evaluate to 0)
    - `2 <= 0`
    - `10 == 9`
    - `100 < 100`

6.5 Boolean operations

- AND
  `toocold && heater_off`
- OR
  `toocold || toohot`
- NOT
  `!heater_on`
- Each side of each operation must be a complete expression:
  `check = a == b || a ==c || a==d;` ✓
  `check = a == b || c || d;` ✗
- Use brackets and spaces for clarity:
  `check = ( (a==b) || (a==c) || (a==d) );` ✓✓

References

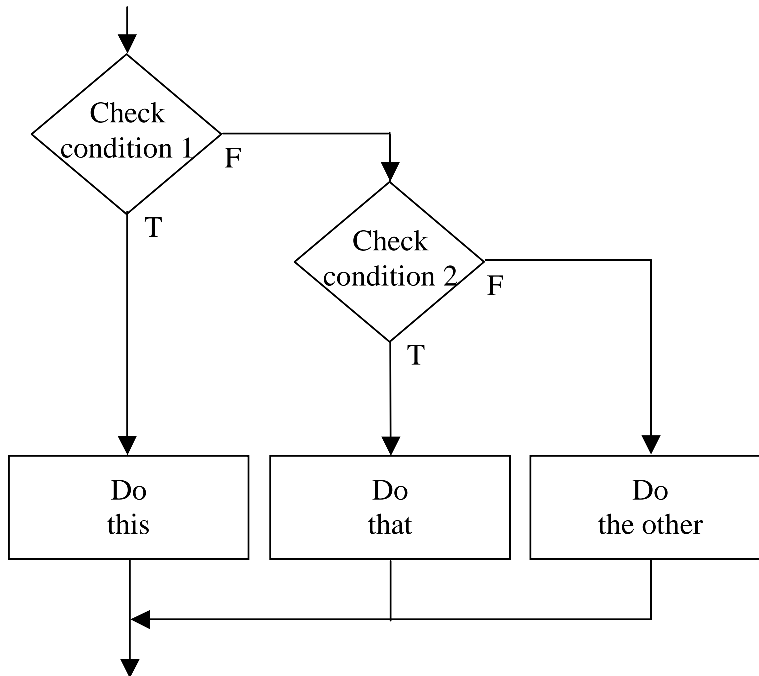
- Dawson:
    - A6.1-6.12
- Kernighan and Ritchie:
    - 3.1,3.2

7 Selecting among several options – else-if and switch

7.1 Else-if

- Else-if can be used to select among several options.

Flowchart



Pseudocode

```
if condition 1
      do this
else if condition 2
      do that
else
      do the other
end if
```

C syntax

```
if (condition_1)
{
      do_this;
}
else if (condition_2)
{
      do_that;
}
else
{
      do_the_other;
}
```

- Example pseudocode:

```
get mark
if mark<0 or mark>100
      print error message
else if mark>=40
      print pass message
else
      print fail message
end if
```

- Example source code:

```
/*
 * passfail.c
 * Determines pass/failure of a module
 * S.P. Platt Last modified 27/08/03
 */
#include <stdio.h>

int main(void)
{
    int mark;

    printf("Program passfail determines whether ");
    printf("a module has been passed.\n\n");

    printf("Enter the mark (0-100%%): ");
    scanf("%d",&mark);

    if ((mark<0) || (mark>100))
    {
        printf("Error: mark (%d%%) ",mark);
        printf("out of range (0-100%%).\n");
    }
    else if (mark>=40)
    {
        printf("Student has passed (%d%%).\n",mark);
    }
    else
    {
        printf("Student has failed (%d%%).\n",mark);
    }

    return 0;
}
```

- Example results:

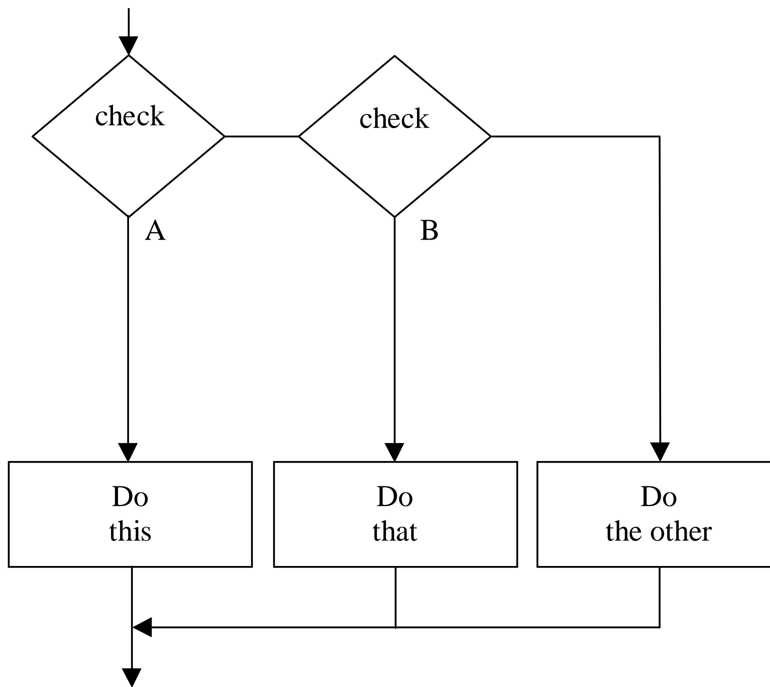
```
prompt>passfail
Program passfail determines whether a module has been passed.

Enter the mark (0-100%): -1
Error: mark (-1%) out of range (0-100%).

prompt>passfail
Program passfail determines whether a module has been passed.

Enter the mark (0-100%): 101
Error: mark (101%) out of range (0-100%).

prompt>passfail
Program passfail determines whether a module has been passed.

Enter the mark (0-100%): 39
Student has failed (39%).

prompt>passfail
Program passfail determines whether a module has been passed.

Enter the mark (0-100%): 40
Student has passed (40%).
```

© S.P. Platt 2001-2005

7.2 Switch

- The `switch` statement is a convenient alternative to `else-if`

Flowchart



Pseudocode

```
switch check equals
   A
      do this
      break
   B
      do that
      break
   default
      do the other
      break
end switch
```

C syntax

```
switch (check)
{
   A:
      do_this;
      break;
   B:
      do_that;
      break;
   default:
      do_the_other;
      break;
}
```

`switch` statement

```
switch (check)
{
   case A:
      do_this;
      break;
   case B:
      do_that;
      break;
   default:
      do_the_other;
      break;
}
```

`else-if` equivalent

```
if (check==A)
{
      do_this;
}
else if (check==B)
{
      do_that;
}
else
{
      do_the_other;
}
```

- Example

```
/*
 * File:        months.c
 * Author:      S.P. Platt
 * Last edited: 30/6/2003
 * Description: Demonstrates switch
 */

#include <stdio.h>

int main(void)
{
   int month;

   printf("Which month is it?\n");
   printf("Please enter a number between 1 and 12: ");
   scanf("%d",&month);

   switch (month)
   {
      case 1:
         printf("It's January.  ");
         break;
      case 2:
         printf("It's February.  ");
         break;
// … More of the same left out so the example fits on the page
         case 11:
         printf("It's November.  ");
         break;
      case 12:
         printf("It's December.  ");
         break;
      default:
         printf("I don't know which month it is.  ");
         break;
   }

   switch (month)
   {
      case 1:  //fall through
      case 2:
      case 3:
      case 4:
      case 9:
      case 10:
      case 11:
      case 12:
         printf("There's an 'r' in the month.");
         break;
      case 5: case 6: case 7: case 8:  //fall through
         printf("There's no 'r' in the month.");
         break;
      default:
         break;
   }

   return 0;
}
```

© S.P. Platt 2001-2005

- Analysis:

```
switch (month)
{
…
}
```

  - The value of `month` is checked. The program will switch to one of the lines inside the braces, {}.

```
case 1:
```

  - If the value of `month` equals 1, the program will switch to `case 1:`

```
printf("It's January.  ");
```

  - The message is printed

```
break;
```

  - The `break` statement causes the program to jump to the end of the `switch` – to "break out".

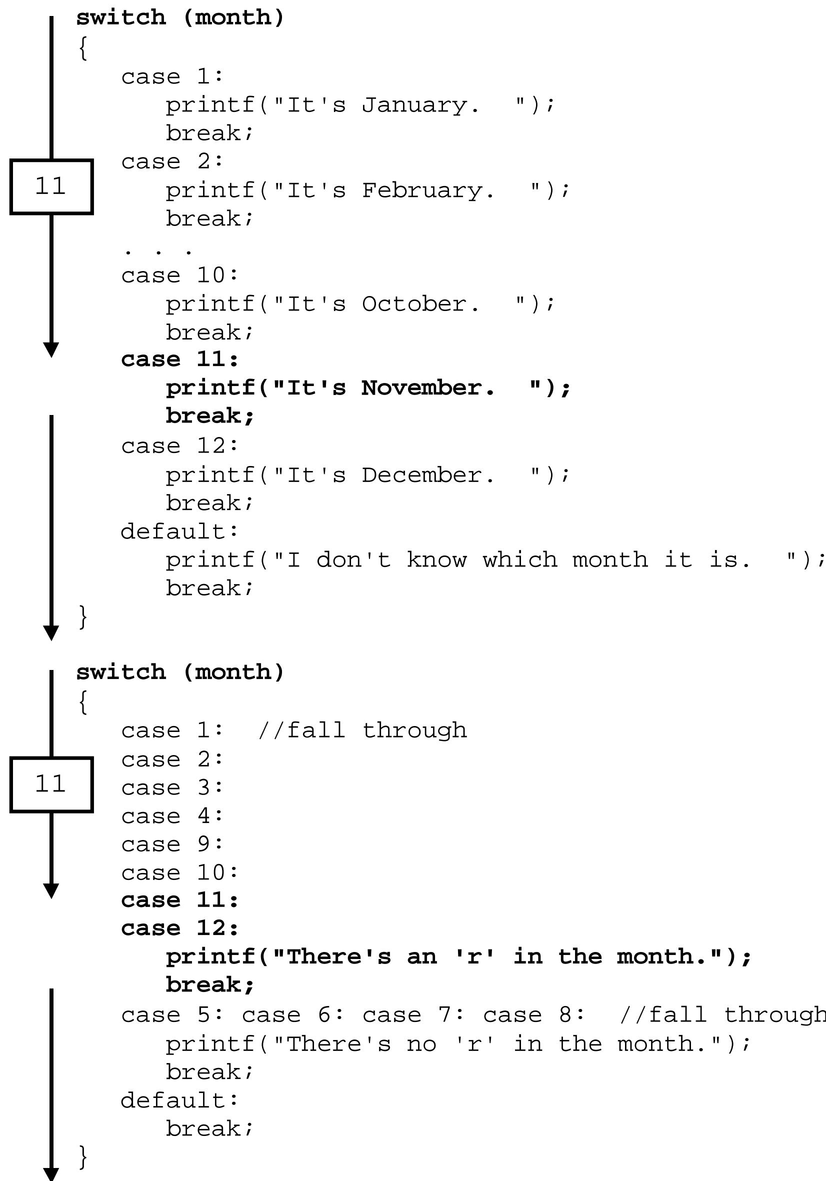
```
default:
```

  - If the value of `month` does not match any of the cases, the program will switch to `default:`

- N.B.
  - Cases can come in any order
  - The default case is optional
  - Nothing is done if no matching case is found and `default` is omitted.
- Results:

```
prompt> months
Which month is it?
Please enter a number between 1 and 12: 8
It's August.  There's no 'r' in the month.
prompt> months
Which month is it?
Please enter a number between 1 and 12: 9
It's September.  There's an 'r' in the month.
prompt> months
Which month is it?
Please enter a number between 1 and 12: 11
It's November.  There's an 'r' in the month.
prompt> months
Which month is it?
Please enter a number between 1 and 12: 123
I don't know which month it is.
prompt>
```

- Execution sequence:



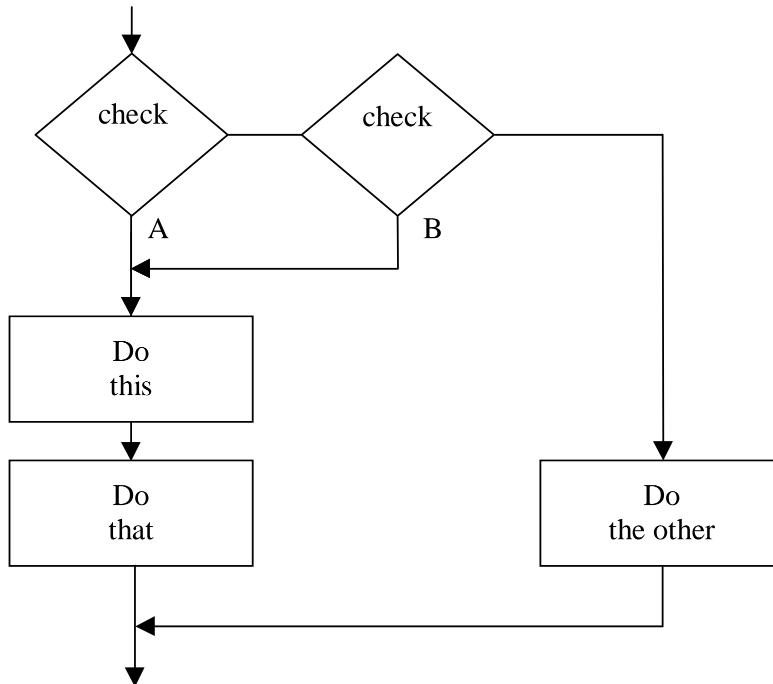

```
switch (month)
{
   case 1:
      printf("It's January.  ");
      break;
   case 2:
      printf("It's February.  ");
      break;
   . . .
   case 10:
      printf("It's October.  ");
      break;
   case 11:
      printf("It's November.  ");
      break;
   case 12:
      printf("It's December.  ");
      break;
   default:
      printf("I don't know which month it is.  ");
      break;
}
```

```
switch (month)
{
   case 1:  //fall through
   case 2:
   case 3:
   case 4:
   case 9:
   case 10:
   case 11:
   case 12:
      printf("There's an 'r' in the month.");
      break;
   case 5: case 6: case 7: case 8:  //fall through
      printf("There's no 'r' in the month.");
      break;
   default:
      break;
}
```

7.2.1 Breaking out and falling through

- A `switch` statement switches to the appropriate `case` label, or to the `default` label, if appropriate.
- To stop execution (preventing subsequent cases from being executed) use `break`.
- If `break` is omitted, the switch “falls through” to the following case(s).
- months.c illustrates both usages.
- Examples of falling through are shown below

Flowchart



Pseudocode

```
switch check equals
   A,B
      do this
      do that
      break
   default
      do the other
      break
end switch
```

C syntax

```
switch (check)
{
   A:
   B:
      do_this;
      do_that;
      break;
   default:
      do_the_other;
      break;
}
```

`switch` example

Pseudocode:

```
switch check equals
   A,B
      do this
      do that
      break
   default
      do the other
      break
end switch
```

C syntax:

```
switch (check)
{
    A:
    B:
        do_this;
        do_that;
        break;
    default:
        do_the_other;
        break;
}
```

`if-else`/`else-if` equivalent

Pseudocode:

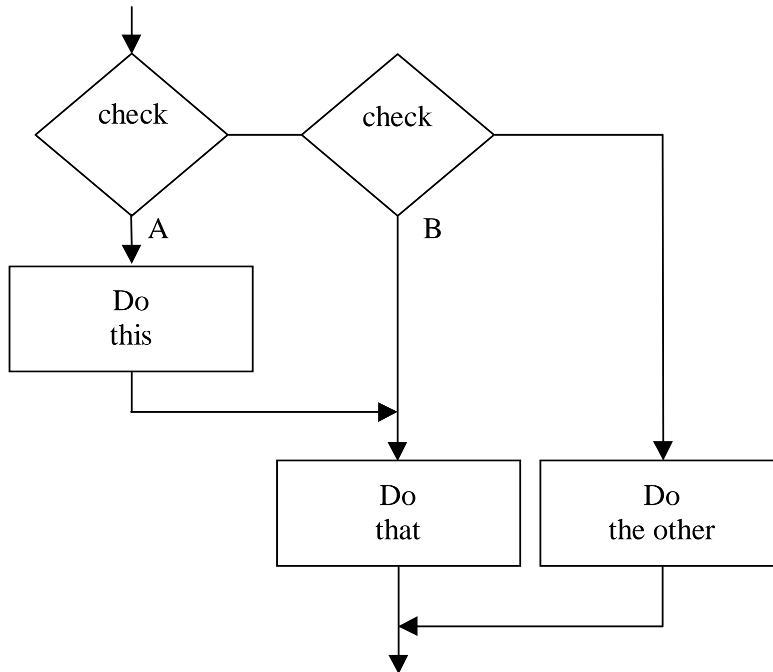
```
if (check equals A) or (check equals B)
   do this
   do that
else
   do the other
end if
```

C syntax:

```
if ((check==A) || (check==B))
{
   do_this;
   do_that;
}
else
{
   do_the_other;
}
```

© S.P. Platt 2001-2005

Flowchart



C syntax

```
switch check equals
   A
      do this
      fall-through
   B
      do that
      break
   default
      do the other
      break
end switch
```

C syntax

```
switch (check)
{
   A:
      do_this;
   B:
      do_that;
      break;
   default:
      do_the_other;
      break
}
```

`switch` example — **`if-else`/`else-if` equivalent**

Pseudocode:

`switch` example:

```
switch check equals
   A
      do this
      fall through
   B
      do that
      break
   Default
      do the other
      break
end switch
```

`if-else`/`else-if` equivalent:

```
if (check equals A) or (check equals B)
   if check equals A
      do this
   end if
   do that
else
   do the other
end if
```

C syntax:

`switch` example:

```
switch (check)
{
   A:
      do_this
   B: do_that;
      break;
   default:
      do_the_other;
      break;
}
```

`if-else`/`else-if` equivalent:

```
if ((check==A) || (check==B))
{
   if (check==A)
   {
      do_this;
   }
   do_that;
}
else
{
   do_the_other;
}
```

© S.P. Platt 2001-2005

7.3 Switch versus if-else and else-if

- `switch` does just one check, at the beginning
  - The cases must be mutually exclusive.
  - If more than one case needs the same action, a fall-through must be used.
- `switch` can only be used for checking when the condition is *equal* to an *integer* value
  - passfail.c (above) cannot be re-written using a `switch`
    - `mark<0`, `mark>100`, `mark>=40`, were all used as tests.
  - This is a bad `switch` – because it's checking a *floating-point* value

```
switch (TCel)     //error (assuming TCel to be a float or a double)
{
   case 0.0:      //error
      printf("Freezing point.");
      break;
   case 100.0:    //error
      printf("Boiling point.");
      break;
   default:
      break;
}
```

- `if-else` and `else-if` are more general than `switch`
  - Every `switch` statement can be re-written as an `if-else` or `else-if` statement
    - This often needs complex statements or repeated boolean operations checks
    - `switch` is often more convenient and natural than `if-else` or `else-if`
  - Not every `else-if` statement can be written as a `switch` statement

References

- Dawson:
  - A6.13, 7.3
- Kernighan and Ritchie:
  - 3.3, 3.4

© S.P. Platt 2001-2005

8 Programming style

- Good programming style is mostly about making programs easy to read.
- It is important whatever the programming language.
- Good style example:

```
/*
 * passfail.c
 * Determines pass/failure of a module
 * S.P. Platt Last modified 27/8/2002
 */

#include <stdio.h>

#define MINMARK    0
#define MAXMARK  100
#define PASSMARK  40

int main(void)
{
      int mark;
      int badmark,pass;              /*boolean*/

      printf("Program passfail determines whether ");
      printf("a module has been passed.\n\n");

      /*get mark*/
      printf("Enter the mark (%d-%d%%): ",MINMARK, MAXMARK);
      scanf("%d",&mark);

      /*check mark*/
      badmark = (mark<MINMARK)||(mark>MAXMARK);
      if (badmark)
      {
            printf("Error: mark (%d%%) ",mark);
            if (mark<MINMARK)
            {
                  printf("less than minimum (%d%%).\n",MAXMARK);
            }
            else if (mark>MAXMARK)
            {
                  printf("greater than maximum (%d%%).\n",MAXMARK);
            }
      }
      else
      {
            /*check pass*/
            pass = (mark>=PASSMARK);
            if (pass)
            {
                  printf("Student has passed (%d%%).\n",mark);
            }
            else
            {
                  printf("Student has failed (%d%%).\n",mark);
            }
      }

      return 0;
}
```

© S.P. Platt 2001-2005

- Bad style example:

```
#include <stdio.h>
int main(void){
int mark; /*integer variable*/
printf("Program passfail determines whether a module has been
passed.\n\n");
printf("Enter the mark (0%%-100%%): ");
/*read mark*/scanf("%d",&mark);if (mark<0||mark>100)
{printf("Error: mark (%d%%) ",mark);
if (mark<0)
printf("less than minimum (0%%).\n");
else{
printf("greater than maximum (100%%).\n");}}
else
if (mark>=40)printf("Student has passed (%d%%).\n",mark);/*pass*/
else printf("Student has failed (%d%%).\n",mark);/*fail*/
return 0;}
```

8.1 Code Layout

- Layout has no effect on program *operation*
- Good layout rules makes programs *readable*
  - Fewer errors made
  - Errors more easily spotted
  - Programmer's intention clear
  - Program more easily modified
- Good layout shows the structure of a program
- Poor layout makes programs unreadable
- Break up your code into related parts.
  - Use blank lines to separate sections.
  - Add comments above each part if necessary.
  - Group variable definitions logically
  - Add comments at the end of a line if necessary.
- Use indentation to show program structure.
  - Everything outside `main` starts at the leftmost column
  - Everything inside main is indented by one tab stop (three or four spaces is enough)
  - Everything inside if, else etc. is indented by a further tab stop
    - Indent after each {, go back before each }
  - Nested ifs (one inside another) get extra levels of indentation
  - Make comments follow the same indentation scheme as the code
  - Use fixed space font (e.g. `Courier`) when printing to maintain indentation
- Use braces to show beginning and end of if or else statements
  - Make them line up with each other
  - Make them follow the indentation scheme
- Use brackets and spaces in equations for clarity
- Make lines fit on the page
  - Split statements into two if necessary

© S.P. Platt 2001-2005

8.2 Naming

- Choose suitable data names
  - Short
  - Meaningful
  - Consistent
- Avoid magic numbers – use “symbolic constants”

```
#define MINMARK    0
#define MAXMARK  100
#define PASSMARK  40
```

  - Note: no ‘=’, no ‘;’
  - Makes meaning clear
  - Makes changes easy
- Use boolean variables

```
badmark = (mark<MINMARK)||(mark>MAXMARK);
pass = (mark>=PASSMARK);
```

  - Makes meaning clear

8.3 Comments

- Use comments:
  - At the top of every source code file
  - To clarify the meaning of code
- Comments are ignored by the compiler and do not affect the operation of the program.
- Make comments *complement* code
  - Irrelevant comments are worse than useless – they make code *harder* to follow.
- Make code self-commenting
  - Good layout and naming needs few comments
- Make comments line up
  - With each other
  - With code

References

- Dawson:
  - D2.4-2.6

© S.P. Platt 2001-2005

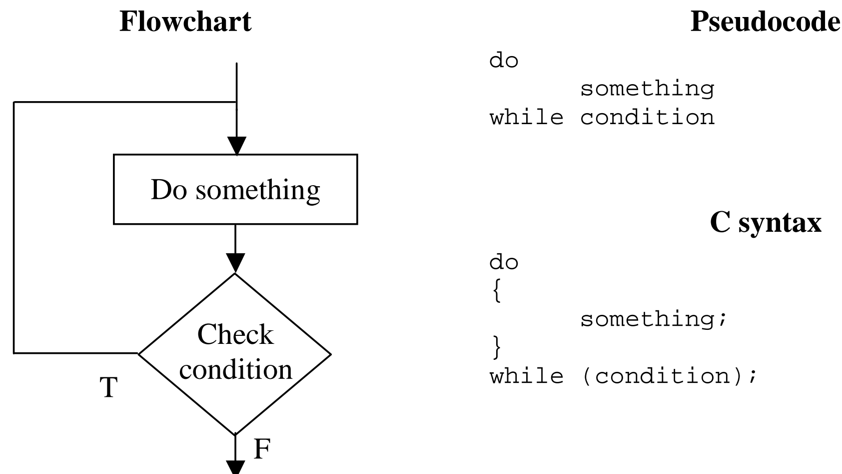
9 Iteration

- We often need to repeat statements.
- This process of doing things repeatedly is known as *iteration*.
- Like a conditional statement, iteration also requires
  - boolean operations (to determine whether or not an action is to be repeated)
  - a means of program control, a *loop* statement.
- There are three sorts of loops, known in C as
  - do-while
  - while
  - for

9.1 Do-while loop

- A do while loop (or "do loop") continuously repeats some action while a condition is true.
  - The check comes *after* the action.
  - The number of iterations is not predetermined, but is *at least one*.

Flowchart



Pseudocode

```
do
      something
while condition
```

C syntax

```
do
{
      something;
}
while (condition);
```

- Example pseudocode

```
do
      get mark
while mark invalid

pass = (mark >= pass mark)
if pass
      print pass message
else
      print fail message
end if
```

- Example source code

```
/*
 * passfail.c
 * Determines pass/failure of a module
 * S.P. Platt Last modified 28/08/2003
 */
#include <stdio.h>
#define MINMARK 0
#define MAXMARK 100
#define PASSMARK 40

int main(void)
{
	int mark;
	int badmark,pass;

	printf("Program passfail determines whether ");
	printf("a module has been passed.\n\n");

	/*get valid mark*/
	do
	{
		/*get mark*/
		printf("Enter the mark (%d-%d%%): ",MINMARK,MAXMARK);
		scanf("%d",&mark);

		/*check mark*/
		badmark = (mark<MINMARK)||(mark>MAXMARK);
	}
	while (badmark);

	pass = (mark>=PASSMARK);
	if (pass)
	{
		printf("Student has passed (%d%%).\n",mark);
	}
	else
	{
		printf("Student has failed (%d%%).\n",mark);
	}

	return 0;
}
```

- Results

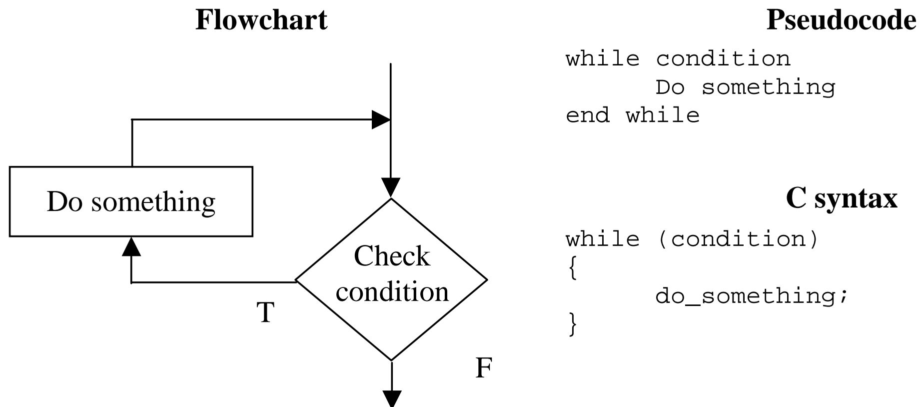
```
prompt>passfail
Program passfail determines whether a module has been passed.

Enter the mark (0-100%): -1
Enter the mark (0-100%): 101
Enter the mark (0-100%): 39
Student has failed (39%).

prompt>passfail
Program passfail determines whether a module has been passed.

Enter the mark (0-100%): 100
Student has passed (100%).

prompt>
```

© S.P. Platt 2001-2005

9.2 While loop

- A while loop continuously repeats some action while a condition is true.
  - The check comes *before* the action.
  - The number of iterations is not predetermined, and *may be zero*.

Flowchart



Pseudocode

```
while condition
      Do something
end while
```

C syntax

```
while (condition)
{
      do_something;
}
```

- Example pseudocode

```
count = 0
while not end of line
      count = count + 1
end while
print count
```

- Example source code:

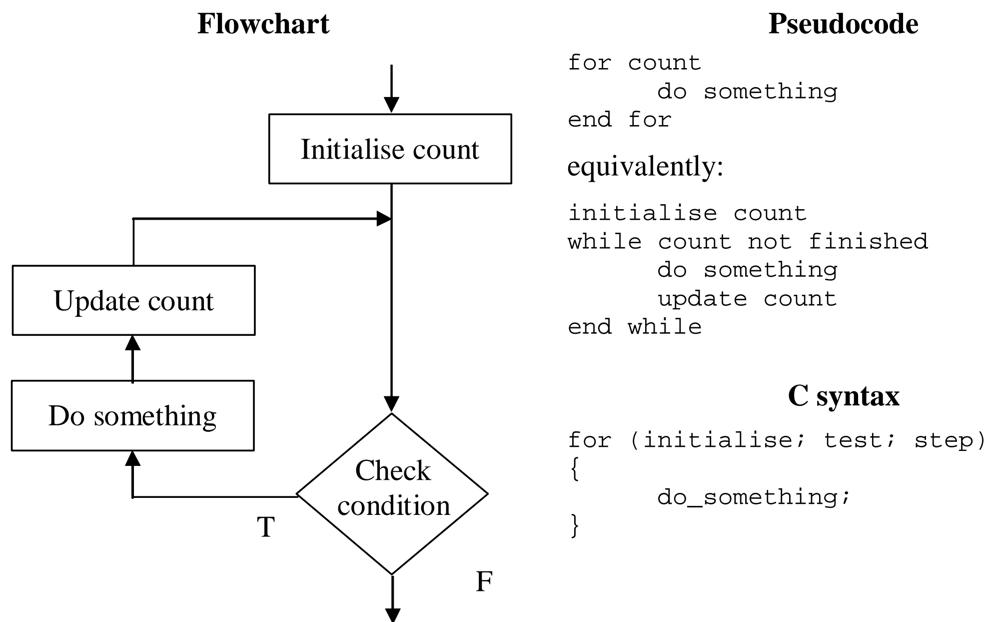
```
/*
 * countchars.c
 * Counts number of characters entered on a line
 * S.P. Platt Last modified 28/08/2003
 */

#include <stdio.h>

int main(void)
{

    int nchars;

    puts("Program countchars counts the characters on a line.\n");

    puts("Enter a line of text:");
    nchars=0;
    while (getchar()!='\n')
    {
        nchars = nchars+1;
    }

    printf("You entered %d characters.\n",nchars);

    return 0;
}
```

- Example results

```
prompt>countchars
Program countchars counts the characters on a line.

Enter a line of text:
Hello
You entered 5 characters.

prompt>countchars
Program countchars counts the characters on a line.

Enter a line of text:
Goodbye!
You entered 8 characters.

prompt>
```

9.3 For loop

- A for loop repeats some action a *predetermined* number of times.



Pseudocode

```
for count
      do something
end for
```

equivalently:

```
initialise count
while count not finished
      do something
      update count
end while
```

C syntax

```
for (initialise; test; step)
{
      do_something;
}
```

- Example pseudocode

```
read start
for count = start down to 1
      print count
end for
print lift off!
```

- Example source code:

```
/*
 * countdown.c
 * Counts down to lift off ...
 * S.P. Platt Last modified 28/08/2003
 */
#include <stdio.h>

int main(void)
{

    int i,start;

    puts("Program countdown counts down to lift off ...");

    printf("How many seconds to lift off? ");
    scanf("%d",&start);

    for (i=start;i>0;i--)
    {
        printf("%d ...\n",i);
    }
    printf("Lift off!");

    return 0;
}
```

- Results

```
prompt>countdown
Program countdown counts down to lift off ...
How many seconds to lift off? 10
10 ...
9 ...
8 ...
7 ...
6 ...
5 ...
4 ...
3 ...
2 ...
1 ...
Lift off!
prompt>countdown
Program countdown counts down to lift off ...
How many seconds to lift off? 3
3 ...
2 ...
1 ...
Lift off!
prompt>countdown
Program countdown counts down to lift off ...
How many seconds to lift off? 0
Lift off!
prompt>
```

- Use *integers* for loop counter variables
- Use short names for loop counters (e.g. `i`)
- Use `i++` to *increment* a counter – *count up*
- Use `i--` to *decrement* a counter – *count down*

© S.P. Platt 2001-2005

- C programmers often start counting from 0
  - like this:

```
for (i=0; i<10; i++) //count from 0 to 9
```

  - rather than this:

```
for (i=1; i<=10; i++) //count from 1 to 10
```

9.4 What sort of loop?

- Use a while or do-while loop when the number of iterations is not known before the loop commences.
  - A do-while loop goes through its loop at least once, and checks for iteration after the body of the loop is completed.
  - A while loop checks for iteration before going though the loop, and it is possible for the loop never to be executed.
- Use a for loop when the number of iterations is known before the loop commences.
  - If the number is not known when the program is written, it must be calculated earlier in the program.

References

- Dawson:
  - A3.22, 7.1, 7.2, 7.4
- Kernighan and Ritchie:
  - 1.3, 2.8, 3.5, 3.6

© S.P. Platt 2001-2005

10 Arrays

- An array is a collection of a
  - Fixed number of data items
  - Of the same type
  - Associated together and given a name
- A string is an array of characters
  - ‘C’, ‘a’, ‘t’ spells “cat”
- A set of students’ module marks could be an array of integers:
  - 55, 63, 71, 45, 35, 58 …
- A set of voltages measured on an oscilloscope could be an array of floating-point numbers:
  - 0.7320, 0.6959, 0.6601, 0.6793, 0.6824 …

10.1 One-dimensional arrays

10.1.1 Character array (string) example

- Example

```
/*
 * greetings.c
 * Prints a silly greeting.
 * S.P. Platt Last edited 28/8/2003
 */
#include <stdio.h>

int main(void)
{
    char name[10];
    int i;

    printf("What is your name? ");
    scanf("%9s",&name);

    i=0;
    while (name[i]!='\0')
    {
      printf("'%c' ",name[i]);
      i++;
    }
    printf("spells \"%s\".",name);

    return 0;
}
```

- Analysis

`char name[10];`

  - Declares an array, `name`, of 10 chars

`int i;`

  - Declares an int, `i`, which will be used to *index* the array.

`scanf("%9s",&name);`

  - Uses `scanf` to read up to 9 characters into the string.
  - `scanf` automatically adds the string terminator, ‘\0’ after the final true character.

© S.P. Platt 2001-2005

`i=0;`

- Assigns the value 0 to i. Array indices begin at 0.

`while (name[i]!= '\0')`

- Checks whether the `i`th character is the string terminator, '\0'
- "The `i`th character" means "character number `i` after the beginning". The whole loop begins with i=0
- **`name[i]`** refers to the ith character.

`printf("'%c' ",name[i]);`

- Prints the `i`th character.

`i++;`

- Adds 1 to `i`. The whole while loop prints each character in the name, one by one.

- Example results:

```
prompt> greetings
What is your name? Frankie
'F' 'r' 'a' 'n' 'k' 'i' 'e' spells Frankie.
prompt> greetings
What is your name? Benjy
'B' 'e' 'n' 'j' 'y' spells Benjy.
prompt>
```

- "Frankie" example:

name[0]	name[1]	name[2]	name[3]	name[4]	name[5]	name[6]	name[7]	name[8]	name[9]
'F'	'r'	'a'	'n'	'k'	'i'	'e'	'\0'	?	?

- In detail:

Memory allocation	Index	Memory address	Memory content	"Frankie" example
Other data	-6	name-6→	name[-6]	?
	-5	name-5→	name[-5]	?
	-4	name-4→	name[-4]	?
	-3	name-3→	name[-3]	?
	-2	name-2→	name[-2]	?
	-1	name-1→	name[-1]	?
Array `name`	**0**	**name→**	**name[0]**	**'F'**
	1	**name+1→**	**name[1]**	**'r'**
	2	**name+2→**	**name[2]**	**'a'**
	3	**name+3→**	**name[3]**	**'n'**
	4	**name+4→**	**name[4]**	**'k'**
	5	**name+5→**	**name[5]**	**'i'**
	6	**name+6→**	**name[6]**	**'e'**
	7	**name+7→**	**name[7]**	**'\0'**
	8	**name+8→**	**name[8]**	**?**
	9	**name+9→**	**name[9]**	**?**
Other data	10	name+10→	name[10]	?
	11	name+11→	name[11]	?
	12	name+12→	name[12]	?
	13	name+13→	name[13]	?
	14	name+14→	name[14]	?
	15	name+15→	name[15]	?

© S.P. Platt 2001-2005

- The array called name has 10 elements
  - Each element is a char
  - They are stored in *contiguous* memory (i.e. next to each other)
- Use square brackets to *index* individual elements
  - The first element (the first character) is `name[0]` (*not* `name[1]`)
  - The last element is `name[9]` (*not* `name[10]`)
- Can use any integer variable or expression to index elements (e.g. `name[i]`, `name[2i+j]`)
- Individual elements can be used like any other characters
  - `printf("'%c' ",name[i]);`
- The name (`name`) of the array doesn't contain *any* of the 10 characters – it specifies *where* the array can be found in memory
  - `name` is the *address* of the array
  - It *points* to the beginning of the array
- Use iteration to process a whole array
  - For loops are most commonly used – see example below.
  - Greetings.c uses a while loop – while loops are often used to process strings (to look for '\0')
- It is very important not to access `name[-1]` or below (relatively unlikely, but possible) or `name[10]` or above (a very common mistake) because these refer to other data
  - If you read `name[-1]` or `name[10]` by mistake, you will *get the wrong answer*
  - If you write to `name[-1]` or `name[10]` by mistake, you will *corrupt other data*
- Arrays can have any size.
- Arrays can have any type – ints, floats etc.

10.1.2 Numerical array example

- Example pseudocode:

```
for all students
      get mark
      if mark<pass mark
            add to fail list
      else
            add to pass list
      end if
end for

for all students in pass list
      print details
end for

for all students in fail list
      print details
end for
```

© S.P. Platt 2001-2005

- Example code:

```
/*
 * classmarks.c
 * Processes marks for a (very small) class of students.
 * S.P. Platt Last edited 28/8/2003
 */

#include <stdio.h>

#define NSTUDTS   5
#define PASSMARK 40

int main(void)
{
      float marks[NSTUDTS];
      int passlist[NSTUDTS],faillist[NSTUDTS];
      int i,npass,nfail;

      npass=0;
      nfail=0;

      printf("Program classmarks reads class marks and ");
      printf("determines pass and fail lists.\n\n");

      printf("Enter marks for class ...\n");
      for (i=0;i<NSTUDTS;i++)
      {
            printf("student %d: ",i+1);
            scanf("%f",&marks[i]);
            if (marks[i]<PASSMARK)
            {
                  faillist[nfail]=i+1;
                  nfail++;
            }
            else
            {
                  passlist[npass]=i+1;
                  npass++;
            }
      }

      printf("\nThe following %d students passed ...",npass);
      for (i=0;i<npass;i++)
      {
            printf("\nstudent: %d mark: %4.1f",
            passlist[i],marks[passlist[i]-1]);
      }

      printf("\n\nThe following %d students failed ...",nfail);
      for (i=0;i<nfail;i++)
      {
            printf("\nstudent: %d mark: %4.1f",
            faillist[i],marks[faillist[i]-1]);
      }

      return 0;
}
```

© S.P. Platt 2001-2005

- Analysis:

```
#define NSTUDTS   5
```

  - Sets NSTDTS (number of students) to be 5

```
float marks[NSTUDTS];
```

  - Declares an array, called marks, of NSTUDTS floats.

```
int passlist[NSTUDTS],faillist[NSTUDTS];
```

  - Declares two arrays of integers, which will contain the numbers of each student who passes (passlist) or fails (faillist).
  - It is possible that all students will pass or that all students will fail.
  - Both passlist and faillist must each be big enough to hold data for all students.

```
for (i=0;i<NSTUDTS;i++)
{
      printf("student %d: ",i+1);
      scanf("%f",&marks[i]);
      ...
}
```

  - Loops for the number of students, reading marks for student number i+1 (1 to 5) into array element marks[i] (marks[0] to marks[4]).
  - This is the raw data which will subsequently be processed.

```
if (marks[i]<PASSMARK)
{
      faillist[nfail]=i+1;
      nfail++;
}
```

  - Places the student number (i+1) of a student who has failed into the first vacant place in faillist (index nfail), and increments nfail.
  - At the end of the process, nfail records how many students have failed.

```
for (i=0;i<npass;i++)
{
      printf("\nstudent: %d mark: %4.1f",\
      passlist[i],marks[passlist[i]-1]);
}
```

  - Steps through the pass list and prints the student number, passlist[i], and the mark obtained by that student, marks[passlist[i]-1].
  - The integer expression, passlist[i]-1, is used to index the array, marks.

© S.P. Platt 2001-2005

- Example results:

```
prompt> classmarks
Program classmarks reads class marks and determines pass and fail lists.

Enter marks for class ...
student 1: 23
student 2: 40
student 3: 56
student 4: 39
student 5: 57

The following 3 students passed ...
student: 2 mark: 40.0
student: 3 mark: 56.0
student: 5 mark: 57.0

The following 2 students failed ...
student: 1 mark: 23.0
student: 4 mark: 39.0
prompt>
```

10.2 Multi-dimensional arrays

- Data arranged in rows or columns form *one-dimensional* (1D) arrays
  - `name` (a character string) was an example.
  - So were `marks` (a list of floats) and `passlist` and `faillist` (lists of ints).
- Data arranged in rows *and* columns form *two-dimensional* (2D) arrays
  - E.g. chessboard

	Columns 0 to 7 →							
Rows 0 to 7 →	0,0	0,1	0,2	0,3	0,4	0,5	0,6	0,7
	1,0	1,1	1,2	1,3	1,4	1,5	1,6	1,7
	2,0	2,1	2,2	2,3	2,4	2,5	2,6	2,7
	3,0	3,1	3,2	3,3	3,4	3,5	3,6	3,7
	4,0	4,1	4,2	4,3	4,4	4,5	4,6	4,7
	5,0	5,1	5,2	5,3	5,4	5,5	5,6	5,7
	6,0	6,1	6,2	6,3	6,4	6,5	6,6	6,7
	7,0	7,1	7,2	7,3	7,4	7,5	7,6	7,7

- There are many applications of 2D arrays:
- E.g. to store a 512×512 grey scale image:

```
#define NROWS 512
#define NCOLS 512
…
int BWimage[NROWS][NCOLS];
…
```

  - The intensity of the pixel at row *i* and column *j* would be stored in array element `BWimage[i][j]`.
- See p. 122 for an example of a program using a 2D array.

- Arrays can also have three (or even more) dimensions.
- A colour image with red, green and blue channels could be stored in a 3D array:

```
int RGBimage[NROWS][NCOLS][3];
```

  - The red, green and blue levels at pixel at *i*,*j* would be stored in array element `RGBimage[i][j][0]`, `RGBimage[i][j][1]` and `RGBimage[i][j][2]`, respectively.

References

- Dawson
  - A2.7, 5
- Kernighan & Ritchie
  - 1.6, 1.9, 2.4, 5.7

© S.P. Platt 2001-2005

11 Data types and addresses

11.1 Integral types

- Ints
    - `int number;`
    - Two bytes (16 bits) minimum
    - Minimum range -2^{15} to $2^{15}-1$ (–32,768 to +32,767)
- Long ints (“longs”)
    - `long bigno;`
    - Four bytes (32 bits) minimum
    - Minimum range -2^{31} to $2^{31}-1$ (–2,147,483,648 to 2,147,483,647)
    - Use `%ld` in scanf
- Also have
    - short ints (saves memory)
    - unsigned ints (no –ve numbers)
    - chars can also hold numerical data (very limited range)

11.2 Floating-point types

- floats
    - `float roughno;`
    - Typically four bytes (32 bits)
    - Typical range 0, $\pm 10^{-37}$ to $\pm 10^{37}$
    - Low(ish) precision
- doubles
    - `double preciseno;`
    - Typically eight bytes (64 bits)
    - Greater range – bigger numbers
    - Increased precision – reduces roundoff errors
- Also have
    - long doubles (especially for astronomers)

11.3 Type conversions

- A value of one type can be assigned to a variable of another type
    - Sometimes it doesn’t work quite as you expect
- Floating-point values can be assigned to integer variables
    - The fractional part is lost (e.g. 3.14159 becomes 3)
- Integral values can be assigned to floating-point variables
    - The fractional part is set to zero (e.g. 3 becomes 3.0)
- Intermediate calculations can suffer from similar effects
    - 10.0/4 is 2.5 – as you would expect
    - 10/4 is 2 – because C assumes integer division, and throws away the fraction

© S.P. Platt 2001-2005

- Type *casts* can be used to force a conversion
  - `(float)10/4` converts 10 (an int) to 10.0 (a float) and gives the correct answer
  - Casting is really most useful for arithmetic using *variables*, rather than constants
- Example:

```
/*
 * average.c
 * Calculates an average
 * S.P. Platt Last updated 29/8/2003
 */

#include <stdio.h>

int main(void)
{
    int i;
    int number,sum;
    float average;

    printf("Program avgmark determines the average of 10 marks.\n\n");

    sum=0;
    printf("Enter the numbers:\n");
    for (i=0;i<10;i++)
    {
        printf("%d: ",i+1);
        scanf("%d",&number);
        sum = sum+number;
    }

    avgmark = (float)sum/10;

    printf("\nThe average is %.1f",average);

    return 0;
}
```

- Analysis

  ```
  int number,sum;
  float average;
  ```

  - `number` and `sum` are ints (without fractions), `average` is a float (with a fraction).

  ```
  sum=0;
  ```

  - Initialises `sum` to zero

  ```
  scanf("%d",&number);
  sum = sum+number;
  ```

  - Reads in each number in turn and adds to `sum`

  ```
  avgmark = (float)sum/10;
  ```

  - Divides `sum` by 10 (the number of numbers read in)
  - The type cast `(float)` converts `sum` to a float before the division
  - The fractional part of the answer is preserved
- Type casts can be used with any data type for temporary conversion
  - `(double)x` converts `x` to a double
  - `(char)i` converts `i` to a char
  - etc.

© S.P. Platt 2001-2005

11.4 Number bases

- Most of us have 10 fingers, so we count in base 10 (“decimal”)
  - 0,1,2,3,4,5,6,7,8,9
- Computers only have two fingers, so they count in base 2 (“binary”)
  - 0,1
- Large numbers become impractically long when expressed in binary
- It is sometimes convenient to express numbers in base 8 (“octal” - fingers only, no thumbs)
  - 0,1,2,3,4,5,6,7
- … or base 16 (“hexadecimal”)
  - 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
- In C:
  - Octal constants are written in C with a leading 0
  - Hexadecimal constants are written with a leading 0x
  - There is no way of writing binary constants
- For example:
  - `16 /*decimal*/ == 020 /*octal*/ == 0x10 /*hexadecimal*/`
  - `100 /*decimal*/ == 0144 /*octal*/ == 0x64 /*hexadecimal*/`
- Octal and hexadecimal integers have their own `printf/scanf` conversion sequences
  - **`%o`** for **o**ctal integer format
  - **`%x`** for he**x**adecimal integer format
  - Hence **`%d`** for **d**ecimal (“normal”) integer format

11.5 Addresses

- Example declaration:
  `float TCel,TFar;`
- Memory reserved:

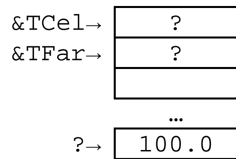
&TCel→	?
&TFar→	?

  - Address of `TCel` is `&TCel`
  - `&TFar` “points to” `TFar`
  - `&TFar` and `&TCel` are *references to* `TFar` and `TCel`
  - They say *where* the variables are stored, *not* what value they have.
  - The values of `TFar` and `TCel` are initially *undefined*
- Use of `&` in `scanf`
  `scanf(”%f”,&TFar);`
  - Reads a value from the keyboard
  - Puts it *at the address where* `TFar` *should go* – `&TFar`.

&TCel→	?
&TFar→	100.0

© S.P. Platt 2001-2005

- Failure to use `&` in `scanf`
  `scanf("%f", TFar); //wrong!`
  - Reads a value from the keyboard
  - Interprets the current value of `TFar` as being an address for the new data
  - Tries to put the new data in the wrong place – may cause your program to crash



- The name of an array represents its address – that is, the address of its first element
  `int anarray[10]`
  - Declares an integer array called `anarray`
  - `anarray` represents the address of the array
  - `&anarray` and `&anarray[0]` are equivalent
- For strings (character arrays) *only*, `&` is optional in scanf
  `scanf("%s",name); //correct - same as scanf("%s",&name);`

11.6 Data initialisation

- By default, variable values are *undefined* at declaration
  - They contain *unpredictable* values.
  - They are *not* automatically set to zero.
- To initialise a variable (give it a value at declaration):
  `float maxlen = 100.0;`
- Example:

```
/*
 * average.c
 * Calculates an average
 * S.P. Platt Last updated 29/8/2003
 */
#include <stdio.h>

int main(void)
{
    int i;
    int number,sum=0;
    float average;

    printf("Program average determines the average of 10 numbers.\n\n");
    printf("Enter the numbers:\n");
    for (i=0;i<10;i++)
    {
        printf("%d: ",i+1);
        scanf("%d",&number);
        sum = sum+number;
    }
    avgmark = (float)sum/10;
    printf("\nThe average is %.1f",average);

    return 0;
}
```

© S.P. Platt 2001-2005

- To initialise an array:
  ```
  int myarray[10] = {1,2,4,8,10};
  ```
  - Sets `myarray[0]` to 1, `myarray[1]` to 2, etc.
  - Any missing elements (`myarray[5]` to `myarray[9]`) are set to 0.
- To initialise a character array with a string:
  ```
  char str[100] = ”hello”;
  ```
  - Here, `str[5]` to str`[99]` are all initialised to `'\0'`.
- You can even do
  ```
  char message[] = ”Hello, world!”;
  ```
  - This works out the length for you automatically (here, 14 characters including ‘\0’).

References

- Dawson
  - A2.8, 3.11-3.18
- Kernighan & Ritchie
  - 2.2, 2.7

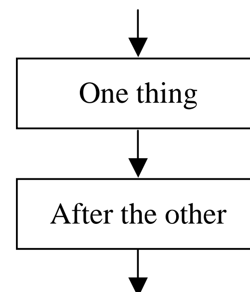
© S.P. Platt 2001-2005

12 Introduction to program design

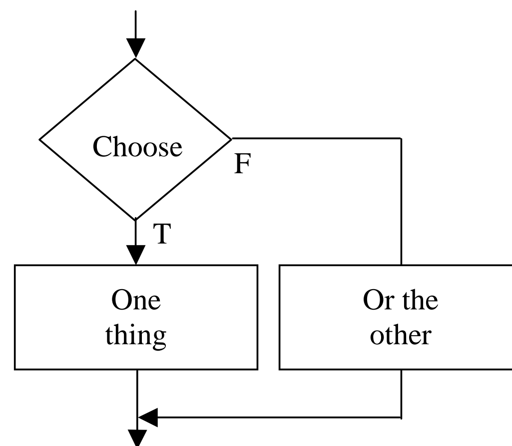
- There are three aspects to program design
  - Determining what *data* is required to be processed.
  - Determining which *operations* are required to process the data.
  - Determining the way in which the operations on data will be *controlled*, that is, choosing which *control structures* are required:
    - Sequential: one after the other
    - Conditional
    - Iterative
- Control structures and the operations they control combine to comprise an *algorithm*.
- A program design is like a recipe, in which the
  - The data represents the ingredients
  - The algorithm is the procedure.

12.1 Structured programming

- A structured program consists of a sequence of well-defined operations, performed one after the other.
  - Well-defined means they have a single way in and a single way out
- All the control structures we have seen (simple sequence, if, if else, else if and switch, do while, while and for) are all well defined.
  - Sequential operation

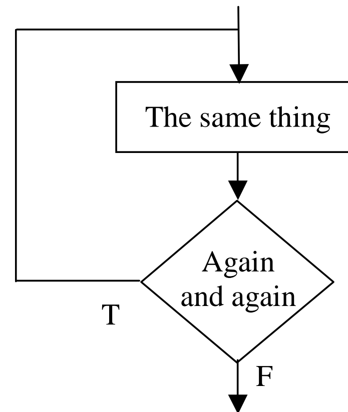


  - Conditional operation
    - If
    - If else
    - Else if
    - Switch



© S.P. Platt 2001-2005

- Iterative
  - Do while
  - While
  - For



- Structured program design means choosing the right structures from this list to meet the requirement
  - They can be performed one after the other – in *sequence*
  - They can also be *nested* – placed one inside another
- Unstructured languages (e.g. BASIC, assembler) do not provide the necessary control structures
  - “Goto” statements and labels are used
- Structured languages (like C) provide all the control structures you need
  - Goto statements and labels are *not* required
- Structured programs are
  - Easier to write
  - Easier to understand
  - Easier to test
- Flowcharts tempt designers to use gotos
- Pseudocode more closely follows the structure of a program source code

12.2 Functional Decomposition

- “Functional decomposition” means “breaking down a requirement into its constituent parts”
- Functional decomposition is a *top-down* design process
  - Starts from overall requirement at a high level of abstraction (low level of detail)
  - Decomposes breaks down requirement into its constituent parts, each with more detail (less abstraction)
  - Continues until the design is broken down into parts which are small enough to be tractable for implementation.
- Each detailed part is implemented, put together from the *bottom up*,
- Implementation is traceable through the design back to the original requirement.
- Pseudocode is used for description:
  - Before implementation (coding), as design aid
  - After implementation, as documentation aid

- Appropriate technique
  - Generally applicable and widespread
  - Well suited to small programs processing simple data
  - Often adopted as part of other techniques for program design

12.2.1 Functional decomposition example

- Requirement:
  - Calculate and display statistics (maximum, minimum and average) on a set of up to 100 students' marks, read from the keyboard.

Top level design

```
begin
      get data
      calculate statistics
      print results
end
```

- Decomposing `get data`

Before

```
begin
      get data
      calculate statistics
      print results
end
```

After

```
begin
      get valid no students
      for each student
            get mark
      end for

      calculate statistics
      print results
end
```

- Decomposing `get valid no students`

Before

```
begin
      get valid no students
      for each student
            get mark
      end for

      calculate statistics
      print results
end
```

After

```
begin
      do
            get no students
      while (no students invalid)

      for each student
            get mark
      end for

      calculate statistics
      print results
end
```

© S.P. Platt 2001-2005

- Decomposing `calculate statistics`

Before

```
begin
      do
            get no students
      while (no students invalid)

      for each student
            get mark
      end for

      calculate statistics
      print results
end
```

After

```
begin
      do
            get no students
      while (no students invalid)

      for each student
            get mark
      end for

      sum = first mark
      maximum = first mark
      minimum = first mark
      for all other marks
            update sum
            update maximum
            update minimum
      end for
      average = sum/(no students)

      print results
end
```

- Decomposing `update sum`, `update maximum` and `update minimum`

Before

```
begin
      do
            get no students
      while (no students invalid)

      for each student
            get mark
      end for

      sum = first mark
      maximum = first mark
      minimum = first mark
      for all other marks
            update sum
            update maximum
            update minimum
      end for
      average = sum/(no students)

      print results
end
```

After

```
begin
      do
            get no students
      while (no students invalid)

      for each student
            get mark
      end for

      sum = first mark
      maximum = first mark
      minimum = first mark
      for all other marks
            sum = sum + mark
            if mark>maximum
                  maximum = mark
            end if
            if mark<minimum
                  minimum = mark
            end if
      end for
      average = sum/(no students)

      print results
end
```

© S.P. Platt 2001-2005

- Decomposing `print results`

Before

```
begin
      do
            get no students
      while (no students invalid)

      for each student
            get mark
      end for

      sum = first mark
      maximum = first mark
      minimum = first mark
      for all other marks
            sum = sum + mark
            if mark>maximum
                  maximum = mark
            end if
            if mark<minimum
                  minimum = mark
            end if
      end for
      average = sum/(no students)

      print results
end
```

After

```
begin
      do
            get no students
      while (no students invalid)

      for each student
            get mark
      end for

      sum = first mark
      maximum = first mark
      minimum = first mark
      for all other marks
            sum = sum + mark
            if mark>maximum
                  maximum = mark
            end if
            if mark<minimum
                  minimum = mark
            end if
      end for
      average = sum/(no students)

      print maximum
      print average
      print minimum
end
```

- This is the final design
  - Decomposition was stopped when the design was sufficiently clear to start coding.
- There might be many ways of breaking down the requirement and many ways of designing each part
- The best program structures are
  - Natural
  - Simple
  - Clear
- This makes programs
  - Less prone to error (and if an error does creep in, it's more likely to be found)
  - More likely to meet the original requirement.

© S.P. Platt 2001-2005

- Implementation of final design – as source code:

```
/*
 * classstats.c
 * Calculates maximum, minimum and average marks
 * S.P. Platt Last updated 29/8/2003
 */

#include <stdio.h>
#define MAXSTDTS 100

int main(void)
{
    int i,Nstdts;
    int mark[MAXSTDTS];
    int maxmark,minmark,sum;
    float avgmark;

    printf("Program classstats determines statistics ");
    printf("on a set of up to %d marks.\n\n",MAXSTDTS);

    /*get data*/
    do
    {
        printf("Enter the number of students: ");
        scanf("%d",&Nstdts);

    }
    while ((Nstdts<1)||(Nstdts>MAXSTDTS));

    for (i=0;i<Nstdts;i++)
    {
        printf("Enter the mark for student %d: ",i+1);
        scanf("%d",&mark[i]);
    }

    /*calculate statistics*/
    sum = mark[0];
    maxmark = mark[0];
    minmark = mark[0];
    for (i=1;i<Nstdts;i++)
    {
        sum = sum + mark[i];
        if (mark[i]>maxmark)
        {
            maxmark = mark[i];
        }
        if (mark[i]<minmark)
        {
            minmark = mark[i];
        }
    }
    avgmark = (float)sum/Nstdts;

    /*print results*/
    printf("\nThe maximum mark is %d",maxmark);
    printf("\nThe average mark is %.1f",avgmark);
    printf("\nThe minimum mark is %d",minmark);

    return 0;
}
```

© S.P. Platt 2001-2005

12.3 Writing pseudocode

- The rules for writing pseudocode are as follows:
  - Write a *sequence of actions* in order. As these are actions, verbs will feature prominently, e.g. read, print, calculate, do.
  - Use *natural language* to make it clear what you mean. Clarity, not rigour or consistency, is what is required. Humans, not computers, will read the pseudocode.
  - Use only the *structured programming* constructs:
    - sequence
    - if
    - if else
    - else if
    - switch
    - while
    - do while (or repeat until)
    - for
  - Use *consistent indentation* to illustrate the structure.
- See how the pseudocode (design) matches the structure of the source code (implementation)
- See how pseudocode tells you *nothing* about the data to be used (types, array sizes etc.)
- See also p. 155

References

- Bell, Morrey and Pugh:
  - 1, 2, 3, 8.2

© S.P. Platt 2001-2005

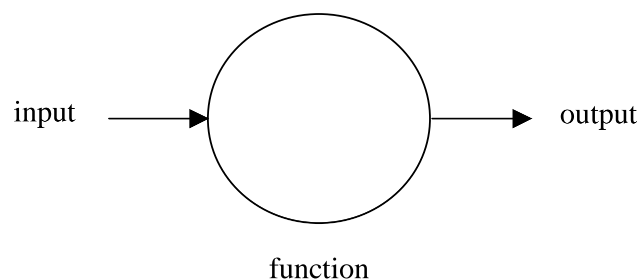
13 The standard library

- We have used some standard techniques for IO, e.g.
  - `printf`: print formatted output to the screen.
  - `scanf`: read formatted input from the keyboard.
- We had to include the following line at the top of our source code:
  - `#include <stdio.h>`
- `printf` and `scanf` are *functions* from the C *standard library*.
  - Defined by the ANSI standard,
  - Available with every ANSI standard compiler.
  - Provides source code (C statements) defining function *interfaces*
    - In “header files”, suffix “.h” (e.g. stdio.h).
  - Provides “object code” (machine instructions) for each function.
- To use the standard library you must
  - Instruct the compiler that library functions are to be used
    - The `#include` statement does this.
  - *Link* to the object code.
    - This is normally automatic.
- The standard library is split into several parts, including these (see also p. 148.):

Category	Header file
Data input and output (IO)	stdio.h
Testing and manipulating individual characters	ctype.h
Testing and manipulating strings	string.h
Mathematics	math.h
Miscellaneous	stdlib.h

13.1 Standard functions

- A function is a useful piece of code, conveniently packaged in a reusable form.
- It may have input and output data:
  - The input (“arguments”) is “passed to” the function.
  - The output (“return value”) is “returned from” the function

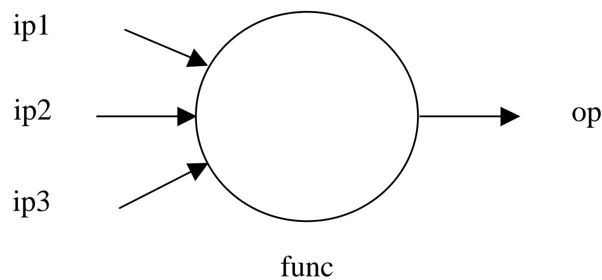


© S.P. Platt 2001-2005

- "Input" and "output" do *not* mean "from the keyboard" and "to the screen"
  - The arguments (input data) are the data that the function processes.
  - The return value (output data) is the result of the function's operations.
- You can:
  - Write your own specialised functions (see p. 87).
  - Use specialised functions written by someone else, or reuse functions you wrote for a different program (see p. 128).
  - Use standard functions, defined in the ANSI standard, written by the clever people who sold you your compiler (read on).

13.2 Function interface

- C functions generally have:
  - A fixed number of arguments (input values)
  - A single return (output) value.



- To use ("call") a function you need only know about its *interface*, not how its code is implemented.
- The interface to the function is defined by
  - Its *name*
  - Its *type* (the type of its return value)
  - The *number* and *types* of its arguments.
- The calling syntax is:

```
op = func(ip1,ip2,ip3);
```

- Here
  - `func` is the name of a function
  - taking three arguments (`ip1`, `ip2`, and `ip3`)
  - and returning a value, which is assigned to the variable `op`.
- The comma-separated list `ip1,ip2,ip3` is called the "argument list".

© S.P. Platt 2001-2005

13.3 Simple examples

- Example

```
/*
 * toupper.c
 * Converts a character to upper case.
 * S.P. Platt Last modified 30/8/2003
 */

#include <stdio.h>
#include <ctype.h>

int main(void)
{
    char ch,CH;

    printf("Program toupper converts a lower-case ");
    printf("character to upper case.\n\n");

    printf("Enter a character: ");
    scanf("%c",&ch);

    CH = toupper(ch);
    printf("The upper case equivalent of '%c' is '%c'.",ch,CH);

    return 0;
}
```

- Analysis

  `#include <ctype.h>`

  - toupper.c uses function `toupper` from the standard library
  - The interface to `toupper` is defined in the header file ctype.h.
  - The include statement makes the function available to the program

  `CH=toupper(ch);`

  - Calls the standard function `toupper`.
  - The argument to `toupper` is `ch`.
  - Its return value, which represents the upper case equivalent of `ch`, is stored in `CH`.

- Results

```
prompt>toupper
Program toupper converts a lower-case character to upper case.

Enter a character: q
The upper case equivalent of 'q' is 'Q'.
prompt>toupper
Program toupper converts a lower-case character to upper case.

Enter a character: !
The upper case equivalent of '!' is '!'.
prompt>
```

- Example

```
/*
 * sqrt.c
 * Calculates a square root
 * S.P. Platt Last modified 30/8/2003
 */

#include <stdio.h>
#include <math.h>

int main(void)
{
    double x,y;

    printf("Program sqrt calculates a square root.\n\n");

    printf("Enter a number: ");
    scanf("%lf",&x);

    y = sqrt(x);

    printf("The square root of %f is %f\n",x,y);
    printf("The square of %f is %f\n",y,y*y);

    return 0;
}
```

- Results

```
prompt>sqrt
Program sqrt calculates a square root.

Enter a number: 4
The square root of 4.000000 is 2.000000
The square of 2.000000 is 4.000000

prompt>sqrt
Program sqrt calculates a square root.

Enter a number: 100
The square root of 100.000000 is 10.000000
The square of 10.000000 is 100.000000

prompt>
```

© S.P. Platt 2001-2005

- Example

```
/*
 * namelen.c
 * Tells you how many characters there are in your name.
 * S.P. Platt Last modified 30/8/2003
 */
#include <stdio.h>
#include <string.h>

int main(void)
{
    char name[100];

    puts("Program namelen will count the characters in your name.");

    printf("What is your name? ");
    scanf("%99s",name);
    printf("Your name, %s, has %d characters.",name,strlen(name));

    return 0;
}
```

- Results

```
prompt>namelen
Program namelen will count the characters in your name.
What is your name? Ozymandias
Your name, Ozymandias, has 10 characters.
prompt>
```

- See how the result of a function is either:
  - Assigned to a variable, for later use:

    ```
    CH = toupper(ch);
    y = sqrt(x);
    ```

  - Used straight away:

    ```
    printf("Your name, %s, has %d characters.",name,strlen(name));
    ```

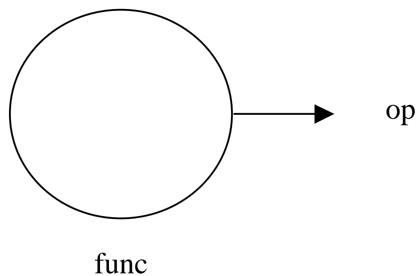
- A common error is to do neither:

  ```
  sqrt(x);
  ```

  - calculates $\sqrt{x}$ but does nothing with the result

13.4 Functions with no arguments

- Sometimes a function has no arguments



- The calling syntax is:

  ```
  op = func();
  ```

- Example

```
/*
 * toupper.c
 * Converts a character to upper case.
 * S.P. Platt Last modified 30/8/2003
 */
#include <stdio.h>
#include <ctype.h>

int main(void)
{
    char ch,CH;

    printf("Program toupper converts a lower-case ");
    printf("character to upper case.\n\n");

    printf("Enter a character: ");
    ch=getchar();

    CH = toupper(ch);
    printf("The upper case equivalent of '%c' is '%c'.",ch,CH);

    return 0;
}
```

- Example

```
/*
 * randy.c
 * Demonstrates pseudorandom number generation.
 * S.P. Platt Last modified 30/8/2003
 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int i;

    printf("Program randy generates a ");
    printf("pseudorandom number sequence.\n\n");

    for (i=0;i<5;i++)
    {
        printf("Pseudorandom number %d is %d\n",i, rand());
    }

    return 0;
}
```

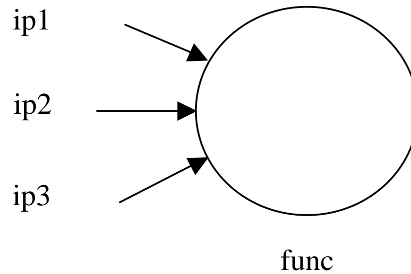
- Results

```
prompt>randy
Program randy generates a pseudorandom number sequence.

Pseudorandom number 0 is 130
Pseudorandom number 1 is 10982
Pseudorandom number 2 is 1090
Pseudorandom number 3 is 11656
Pseudorandom number 4 is 7117

prompt>
```

© S.P. Platt 2001-2005

13.5 Functions with no return value

- Sometimes a function has no return value (a "void" function):



- The calling syntax is:
  `func(ip1,ip2,ip3);`
- Example

```
/*
 * randy.c
 * Demonstrates pseudorandom number generation.
 * S.P. Platt Last modified 30/8/2002
 */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int i;
    long seed;

    printf("Program randy generates a ");
    printf("pseudorandom number sequence.\n\n");
    printf("Enter the seed for the sequence: ");
    scanf("%ld",&seed);

    srand(seed);
    for (i=0;i<5;i++)
    {
        printf("Pseudorandom number %d is %d\n",i, rand());
    }

    return 0;
}
```

- Results

```
prompt>randy
Program randy generates a pseudorandom number sequence.

Enter the seed for the sequence: 1
Pseudorandom number 0 is 130
Pseudorandom number 1 is 10982
Pseudorandom number 2 is 1090
Pseudorandom number 3 is 11656
Pseudorandom number 4 is 7117
```

© S.P. Platt 2001-2005

```
prompt>randy
Program randy generates a pseudorandom number sequence.

Enter the seed for the sequence: 999
Pseudorandom number 0 is 14277
Pseudorandom number 1 is 27712
Pseudorandom number 2 is 25053
Pseudorandom number 3 is 5034
Pseudorandom number 4 is 31899

prompt>
```

- Sometimes a return value is ignored:
  - Common with IO functions
- Example

```
/*
 * printf.c
 * Demonstrates use of the printf return value.
 * S.P. Platt Last modified 5/7/2002
 */
#include <stdio.h>

int main(void)
{
    char nch;

    printf("Program printf demonstrates ");
    printf("using the printf return value.\n\n");

    nch = printf("Hello, world!\n");
    printf("That last line contained %d characters.",nch);

    return 0;

}
```

- Analysis:

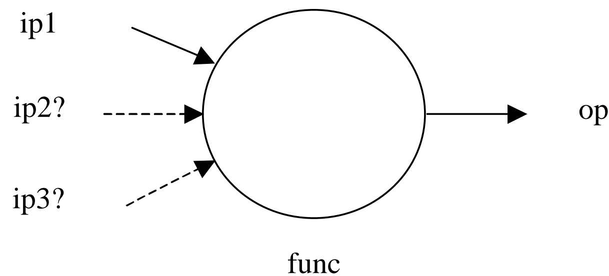
  `nch = printf("Hello, world!\n");`
  - `printf` returns the number of characters printed
  - This is usually ignored …

  `printf("That last line contained %d characters.",nch);`
- Results

```
prompt>printf
Program printf demonstrates using the printf return value.

Hello, world!
That last line contained 14 characters.
prompt>
```

13.6 Functions with a variable argument list

- Sometimes the number of arguments is not fixed.



- `printf` is a good example of this special sort of function.
- The number of arguments to `printf` needs to vary, depending on the number of things to be printed.

```
printf("%d",n);
```

  - Has two arguments: the string, `"%d"` and the integer, `n`.

```
printf("%d, %f",n,x);
```

  - Has three arguments: the string, `"%d, %f"`, the integer, `n` and the floating-point number, `x`.

13.7 Using arguments for output – "pass by reference"

- C functions have *at most* one return value.
- Sometimes it is necessary to have a function that returns more than one result.
  - You might wish to read several data items (numbers, characters, strings) at once with `scanf`.
  - It is possible to do this in C, in a roundabout way.
- Some C functions, such as `scanf`, will write data into memory locations specified in their argument list.
- Arguments must be *addresses* (a *reference* to some data):
  - Names of arrays (e.g. strings)

    `scanf("%s",str);` ✓
  - Other variable names preceded by `&`

    `scanf("%d",n);` ✗
    `scanf("%d",&n);` ✓
- This is known as "pass by reference" or "call by reference" (see p. 111)

References

- Dawson
  - C1-3, C8-10
- Kernighan & Ritchie
  - B1-5

© S.P. Platt 2001-2005

14 File input and output

- We have used some standard IO functions like `printf` and `scanf`.
- These all either:
  - Read text from the *keyboard* ("standard input"), or:
  - Write text to the *screen* ("standard output").
- We can also read and write to and from *files* on a disc.
- In C, files are also known as "streams".

14.1 A simple example

```
/*
 * hellofile.c
 * EL1113 PDI example
 * Text file output
 * S.P.Platt Last edited 4/9/2003
 */
#include <stdio.h>

int main(void)
{
	FILE *opfile;

	opfile=fopen("hello.txt","w");
	if (opfile==NULL)
	{
		printf("Error opening file: hello.txt");
	}
	else
	{
		fprintf(opfile,"Hello, file.");
		fclose(opfile);
	}

	return 0;
}
```

- Analysis:

  `FILE *opfile;`

  - Declares a *file pointer* – a special variable used to refer to ("point to") a file

  `opfile=fopen("hello.txt","w");`

  - Opens a file using the standard function, `fopen`:
    - The name of the file is specified as hello.txt
    - The file mode is "w" ("w" for "write"). This will allow the program to write to the file. Other file modes are listed on page 151.
    - The pointer, `opfile` is set to point to hello.txt. Henceforth, the file is identified by its pointer rather than its name, "hello.txt".

  `if (opfile==NULL)`

  - Checks that the file has been opened correctly.
  - If an error occurs, the pointer is set to the special value `NULL`, and an error message can be written.

  `fprintf(opfile,"Hello, file.");`

  - Writes some text to the file with the standard function, `fprintf`.

- `fprintf` works exactly like `printf`, except that you need to specify which file to print to (using the *pointer*, not the name).

`fclose(opfile);`

- Closes the file using the standard function, `fclose`.
- It is good practice to close a file when you have finished with it.

- Pseudocode for hellofile.c might be:

```
open file
if error
   print error message
else
   write to file
   close file
end if
```

- File IO always follows this pattern:
  - Open your file
  - Check that it has opened correctly
  - Do your IO
  - Close your file
- Functions `fopen`, `fprintf` and `fclose` are from the standard IO library. These are declared in stdio.h. See p. 148 for other standard file IO functions.
- stdio.h also defines two special values as symbolic constants:
  - `NULL` – used to check for error on opening a file.
  - `EOF` – used to check for the end of a file. On a PC, `EOF` can be simulated at the keyboard by pressing `<CTRL>Z`.

14.2 Text and binary files

- *Text* files contain sequences of characters, separated into lines, just like on the computer screen, e.g.
  - Source code files, *.c
  - HTML files, *.htm, *.html
  - Any other file intended to be read in a text editor
- *Binary* files contain direct byte-by-byte copies of data held in memory, e.g.
  - Executable files, *.exe
  - Files that use proprietary data formats, or that need special software to be read, e.g. *.doc, *.xls, *.wmf
- Text files
  - Are *portable* – textual data can easily be transferred between computers and read by different computers.
- Binary files
  - Are *smaller* than text files.
  - Can hold floating-point data *without loss of precision*.
  - Are generally *not* easily portable between different types of computer.
- You must specify the correct data format in the `fopen` mode (see p. 151) when opening a file.
- You must use the appropriate functions for text and binary IO (see p. 148).

© S.P. Platt 2001-2005

14.3 Standard IO streams

- Any C program may also use three *standard streams*.
- These are opened automatically when the program starts and closed automatically when it stops. `fopen` and `fclose` are not used.
- Their file pointers are known as:
  - `stdin`: “standard input”, normally connected to the keyboard.
  - `stdout`: “standard output”, normally connected to the screen.
  - `stderr`: “standard error”, normally connected to the screen.
- Standard input functions (e.g. `scanf`) read from `stdin`.
  - `fscanf(stdin," … ")` is equivalent to `scanf(" … ")`
- Standard output functions (e.g. `printf`) write to `stdout`.
  - `fprintf(stdout," … ")` is equivalent to `printf(" … ")`
- `stderr` is useful because `stdout` can be “redirected” to send output somewhere else (for example, to a file), in which case error messages can be sent to `stderr`, so that they can still be seen on the screen.
- Example:

```
/*
 * greetings.c
 * Prints a friendly greeting.
 * S.P. Platt Last edited 1/9/2003
 */
#include <stdio.h>

#define LINELEN 100

int main(void)
{
    char name[LINELEN];

    printf("What is your name? ");

    fgets(name,LINELEN,stdin);
    printf("Hello, %s.",name);

    return 0;
}
```

- Analysis:

  `fgets(name,LINELEN,stdin);`
  - Gets a string from a file
    - The string is stored in `name`
    - `name` can contain up to `LINELEN` (100) characters (including ‘\0’)
    - The “file” is `stdin` – the keyboard
  - `fgets` reads a line of up to `LINELEN-1` (99) characters from the keyboard into `name`.
    - The end of line character (‘\n’) is read too – see below.
    - This is a common way of reading a line of text from the keyboard (see p. 136)

© S.P. Platt 2001-2005

- Results

```
prompt> greetings
What is your name? Slartybartfast
Hello, Slartybartfast
.
prompt> greetings
What is your name? Ozymandias, King of kings
Hello, Ozymandias, King of kings
.
prompt>
```

- Example:

```
/*
 * hellofile.c
 * EL1113 PDI example
 * Text file output
 * S.P.Platt Last edited 4/9/2003
 */
#include <stdio.h>

int main(void)
{
   FILE *opfile;

   opfile=fopen("hello.txt","w");
   if (opfile==NULL)
   {
      fprintf(stderr,"Error opening file: hello.txt");
   }
   else
   {
      fprintf(opfile,"Hello, file.");
      fclose(opfile);
   }

   return 0;
}
```

- Analysis

  `fprintf(stderr,"Error opening file: hello.txt");`

- Prints formatted data to a file
  - The "file" is `stderr` – the keyboard
  - The data is the error message

References

- Dawson
  - A12, C5, C6
- Kernighan & Ritchie
  - 7.5-7.7, B1

15 Modular program design

- Program *modules* are self-contained pieces of code
  - “Functions” (in C)
  - “Subprograms”, “subroutines”, “procedures”, “methods” (in other languages)
- A simple interface hides a complex (maybe) algorithm
  - This makes functions easy to use – cf. `printf`, `scanf` etc.
- Benefits of modular programming:
  - *Design* can be divided into more tractable parts
  - *Implementation* can be tackled piecemeal, or divided among a team
  - *Test* can be done module by module
  - *Debugging* is easier as the effect of errors can be contained.
  - *Maintenance*: modules can be further developed in isolation.
  - *Reuse*: modules can be reused, even in different programs.

15.1 Modular design example

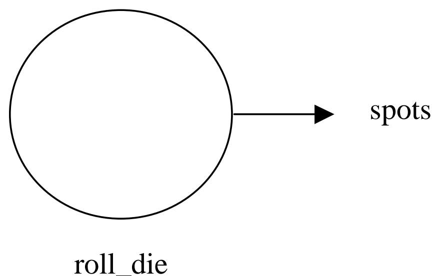
- Here is pseudocode for a program to simulate a snakes and ladders game:

```
begin
   choose counters
   while no winner
         get next player
         roll die
         move counter
         check for winner
   end while
end
```

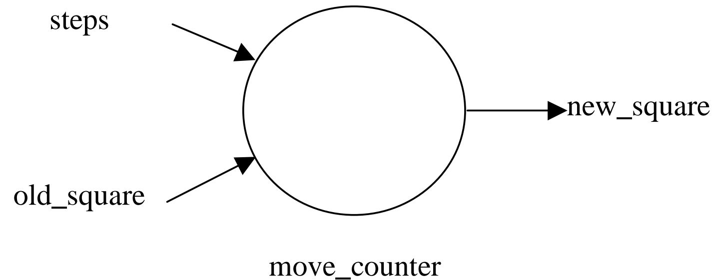
- This pseudocode is at a high level of abstraction (low level of detail).
  - Could just use functional decomposition to add detail
  - Useful to split the design into separate modules.

15.2 Choosing modules

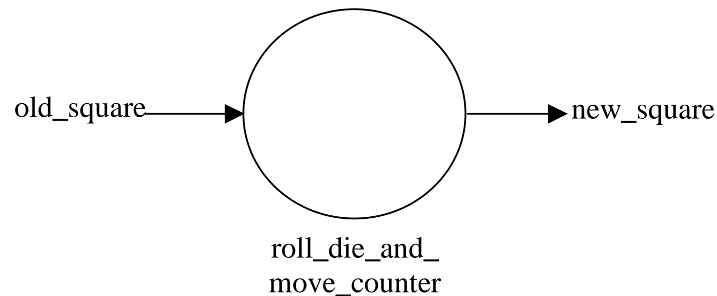
- You should choose modules that
  - Have simple interfaces
  - Can be easily tested
  - Are reusable
- roll_die
  - Output: number of spots on a die



- Good choice of module:
  - Performs a simple, clearly defined function
  - Easy to design, implement and test in isolation from the rest of the program
  - Can be re-used in the same or similar programs
- move_counter:
  - Inputs: old square and no steps
  - Output: new square
  - Takes care of all the details – snakes and ladders.



- Good choice of module:
  - Simple relationship between inputs and outputs
  - Easily tested
- roll_die_and_move_counter:

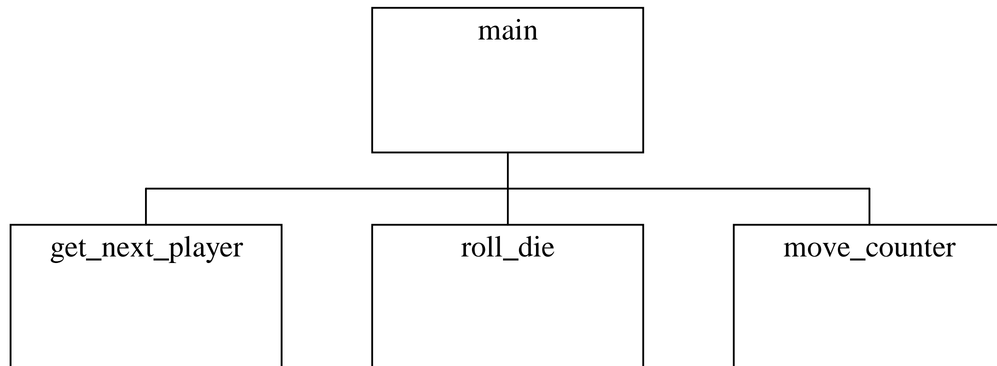


- Poor choice of module:
  - More complex and less well defined behaviour (lacks "cohesion")
  - Much harder to test.
  - Unlikely to be reusable

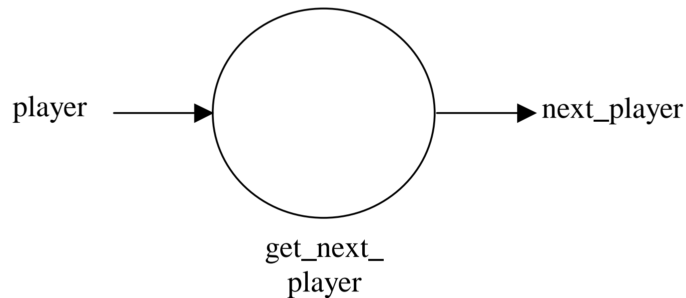
© S.P. Platt 2001-2005

15.3 Module hierarchy

- Modules often use other modules (functions call other functions)
  - Module hierarchy
  - Top-level is main
- Structure chart:



  - main uses (calls) get_next_player, roll_die and move_counter
- get_next_player interface:



15.4 Module specification

- Diagrams are good for developing and communicating design ideas:
  - But they don't specify modules *precisely*
- Interfaces
  - Specify exact number, types and interpretation of input and output data.
  - Use function *prototypes* in C.
- Content
  - Can use pseudocode to specify this.

References

- Bell, Morrey and Pugh
  - 5, 8.5, 8.6, 9

© S.P. Platt 2001-2005

16 Introduction to function implementation

- *All* C programs are based on functions.
  - Functions are the C implementation of modules
  - Libraries of functions can be developed (e.g. ANSI C standard library)
- Functions are easy to use – e.g. `printf`
  - `printf` performs very complex operations
  - The user only needs to know *what* the function does, not *how* (abstraction).
  - The function can be *reused* many times
- Abstraction and reuse make modular programming easy and powerful.
- You can also write your own functions in addition to the standard ones

16.1 A simple example

```
/*
 * FtoC.c
 * Converts Fahrenheit to Celsius using functions
 * S.P.Platt Last modified 30/8/2003
 */
#include <stdio.h>

double FtoC(double TF);

int main(void)
{
    double TFar,TCel;

    printf("Program FtoC converts Fahrenheit to Celsius.\n");

    printf("What is the temperature in deg F? ");
    scanf("%lf",&TFar);

    TCel = FtoC(TFar);

    printf("%.2f deg F equals %.2f deg C",TFar,TCel);

    return 0;
}

double FtoC(double TF)
/*
 * Converts Fahrenheit to Celsius
 * Argument is Fahrenheit temperature
 * Returns Celsius temperature
 */
{
    double TC;

    TC = (TF-32)*5/9;

    return TC;
}
```

© S.P. Platt 2001-2005

- Functions must be
  - Declared
  - Defined
  - Called
- Declaration – function *prototype*

  ```
  double FtoC(double TFar);
  ```
  - Goes at the top of the file – once.
  - Tells the compiler that a function called `FtoC` will be used
  - Defines the *interface* to `FtoC`
  - The *type* of function `FtoC` is *double* – it will calculate a double-precision number
  - `FtoC` has one *input* ("argument") – a double, called `TF`
- Function definition

  ```
  double FtoC(double TF)
  /*
   * Converts Fahrenheit to Celsius
   * Argument is Fahrenheit temperature
   * Returns Celsius temperature
   */
  {
      double TC;

      TC = (TF-32)*5/9;

      return TC;
  }
  ```
  - Defines the *content* of the function (how it does what it does)
  - Matches the *prototype*
  - *Comments* describe the function's interface and behaviour
  - What the function does enclosed in braces `{}`. This code is executed when the function is called
  - Calculates a *value*, `TC` (a double)
  - *Returns* the value of `TC` to the calling function
- Goes at the bottom of the file – once.
- Function *call*

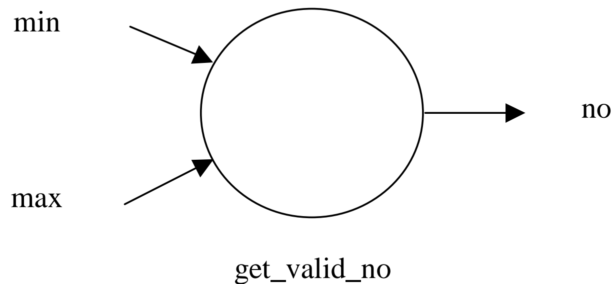
  ```
  TCel = FtoC(TFar);
  ```
  - Uses function `FtoC`
  - Passes the value of `TFar` (a double) to the function
  - Receives back the value of the Celsius equivalent – stores it in `TCel` (a double).
  - May be repeated as often as required
- Results

```
prompt>ftoc
Program FtoC converts Fahrenheit to Celsius.
What is the temperature in deg F? 100
100.00 deg F equals 37.78 deg C
prompt>
```

© S.P. Platt 2001-2005

16.2 A more detailed example

- A program design might have the pseudocode statement:
```
get valid number
```
- This might break down into:
```
do
   prompt for number
   read number
until number between min and max
```
  - It might appear many times in the design – or even in other programs
  - It would be a suitable candidate for a function
- Function interface diagram



16.2.1 Function declaration

- The function must be declared before it is used
- Declaration defines the interface to the function
  - We use a function prototype to do this.
- `get_valid_no` prototype
```
int get_valid_no(int min, int max);
```
- This prototype specifies the following function interface
  - Its name is `get_valid_no`.
  - There are two arguments (inputs):
    - `min`, of type int (to represent the minimum valid number)
    - `max`, also of type int (to represent the maximum valid number)
  - The type of the function (i.e. of its output, or "return value") is int.
- When called, this function would return the value of a number between `min` and `max`.
- The interface does not specify *how* it would do this.
- The function prototype goes at the top of the program

16.2.2 Function definition

- The function must also be defined.
- Definition specifies the detail of the function's operation.
- Function definitions must:
  - Match the function interface specified by the prototype.
  - Specify the actions that the function demands.
  - Return a value, if required by the interface.

© S.P. Platt 2001-2005

- `get_valid_no` definition:

```
int get_valid_no(int min, int max)
/*
 *Get a number from the keyboard
 *Writes to screen and reads from keyboard.
 *Returns a number between min and max
 */
{
    int result;

    do
    {
        printf("Enter a value between %d and %d: ",min,max);
        scanf("%d",&result);
    }
    while ((result<min)||(result>max));

    return result;
}
```

- The first line matches the prototype without the terminating semicolon, "`;`".
- The comment box describes what the function does, including
  - Input data (arguments).
  - Output data (return value).
  - IO (i.e. to screen and from keyboard) actions.
- The beginning and end of the function body are defined by braces, "`{`" and "`}`".
- The function body contains code that is executed when the function is called.
- The function processes the values of its arguments, `min` and `max`.
- The function uses a variable called `result`. This is a *local* variable (see p. 103).
- The `return` statement causes the function to stop and pass the value of `result` back to the calling function.
- The function definition goes at the bottom of the program.

16.2.3 Function call

- To use (call) the function
  - Call it by name
  - Specify the value of the arguments
  - Specify where to put the return value
- `get_valid_no` call:

```
mark = get_valid_no(0,100);
```

- The function expects to be passed two integers
  - `min` is set to 0
  - `max` is set to 100
- The function returns an integer
  - Stored in variable `mark`.
- The function can be used again and again
  - With different input and outputs
  - In the same or different programs

© S.P. Platt 2001-2005

16.2.4 Result

- Program `validno.c` can be used to test function `get_valid_no`

```
/*
 * validno.c
 * Tests get valid no function
 * S.P.Platt Last updated 30/8/2003
 */

#include <stdio.h>

#define TESTMIN  1
#define TESTMAX 10

/*function prototype*/
int get_valid_no(int min, int max);

int main(void)
{
    int check;

    printf("\nProgram validno checks function get_valid_no.\n\n");

    check = get_valid_no(TESTMIN,TESTMAX);
    printf("You entered %d",check);

    return 0;
}

int get_valid_no(int min, int max)
/*
 * Get a number from the keyboard
 * Writes to screen and reads from keyboard
 * Returns a number between min and max
 */
{
    int result;

    do
    {
        printf("Enter a number between %d and %d: ",min,max);
        scanf("%d",&result);
    }
    while ((result<min) || (result>max));

    return result;
}
```

© S.P. Platt 2001-2005

- Program `validno.c` results

```
prompt>validno

Program validno checks function get_valid_no.

Enter a number between 1 and 10: 0
Enter a number between 1 and 10: 11
Enter a number between 1 and 10: 10
You entered 10
prompt>validno

Program validno checks function get_valid_no.

Enter a number between 1 and 10: 1
You entered 1
prompt>validno
```

References

- Dawson
  - A9.1-9.5
- Kernighan & Ritchie
  - 1.7

© S.P. Platt 2001-2005

17 Calling functions

- To "call" a function means to use it.
- To call a function you specify the three parts of the function interface:
  - You need to specify the function's *name*.
  - You need to specify the values of the *arguments* (inputs), if any.
  - You need to specify where to store the *return value*, if required.
- Example

```
/*
 * beeps.c
 * Makes the computer go "beep"
 * S.P.Platt Last updated 30/8/2003
 */
#include <stdio.h>
#include <time.h>

/*function prototypes*/
int get_valid_no(int min, int max);
void beep(int beeps);

int main(void)
{
    int Nbeeps;

    printf("\nProgram beeps makes the computer go \"beep\"\n\n");

    printf("How many beeps do you want?\n");
    Nbeeps = get_valid_no(1,10);

    printf("\nI am about to go \"beep\" %d times...",Nbeeps);
    beep(Nbeeps);
    printf("\nI hope you heard %d beeps.",Nbeeps);

    return 0;
}

int get_valid_no(int min, int max)
/*
 * Get a number from the keyboard
 * Writes to screen and reads from keyboard
 * Returns a number between min and max
 */
{
    int result;

    do
    {
        printf("Enter a number between %d and %d: ",min,max);
        scanf("%d",&result);
    }
    while ((result<min) || (result>max));

    return result;
}
```

```
void beep(int beeps)
/*
 *Makes the computer go "beep"
 *Argument is number of times to go "beep".
 */
{
    int i;
    long start,stop;

    for (i=0;i<beeps;i++)
    {
        /*wait a second*/
        start = clock();
        stop=start+1000;    /*1000 ms*/
        do
        {
            /*nothing*/
        }
        while (clock()<stop);

        /*go "beep"*/
        putchar('\a');
   }
}
```

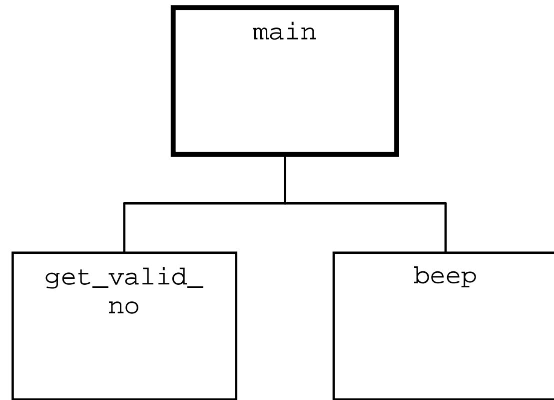
- *Calling* function
    - `main`
- *Called* functions
    - `get_valid_no`
    - `beep`
- Calling `get_valid_no`

  `Nbeeps = get_valid_no(1,10);`
    - Calls function `get_valid_no`.
    - Passes the values 1 and 10 to the function.
    - Stores the return value (result) of the function in variable `Nbeeps`.
- Calling `beep`

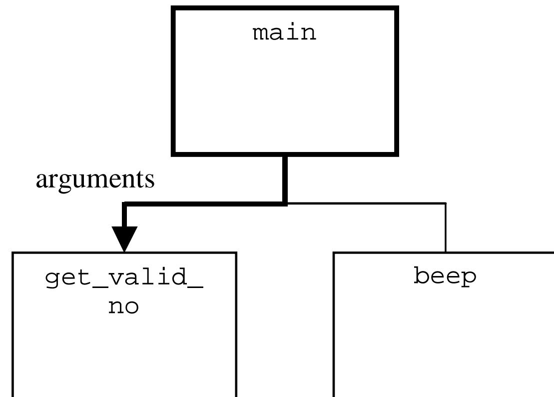
  `beep(Nbeeps);`
    - Calls function `beep`.
    - Passes the value of `Nbeeps` to the function.

- Calling sequence

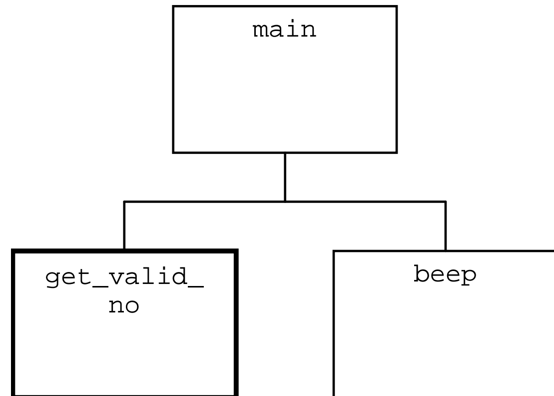
1. Function `main` executes as required, until it reaches the call to function `get_valid_no`.



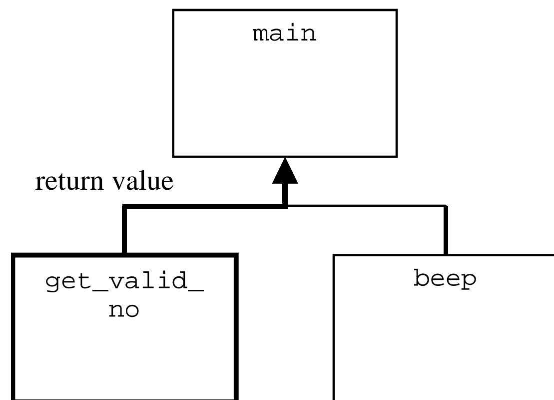
2. Function `main` *passes* the values of the input arguments (`min` and `max`) to function `get_valid_no`.



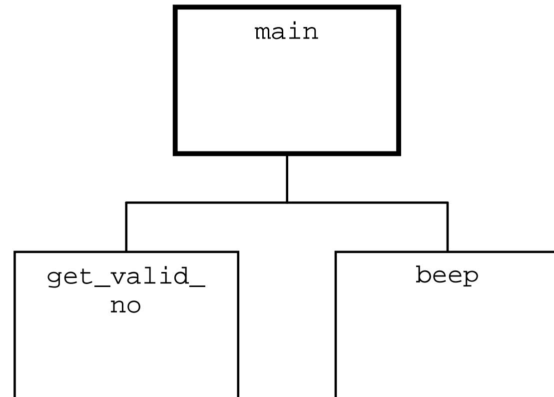
3. Function `main` stops temporarily. Function `get_valid_no` starts, processing the data that `main` has passed to it.



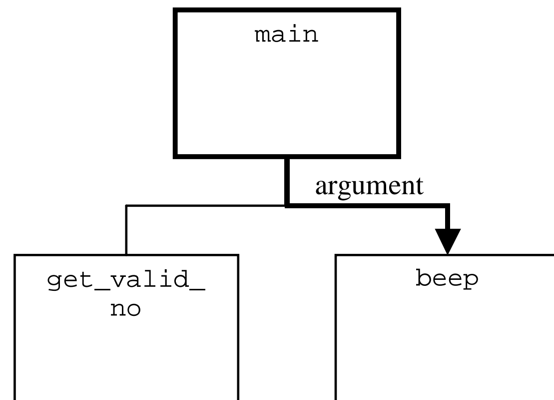
4. Function `get_valid_no` passes its return value to the function `main`.



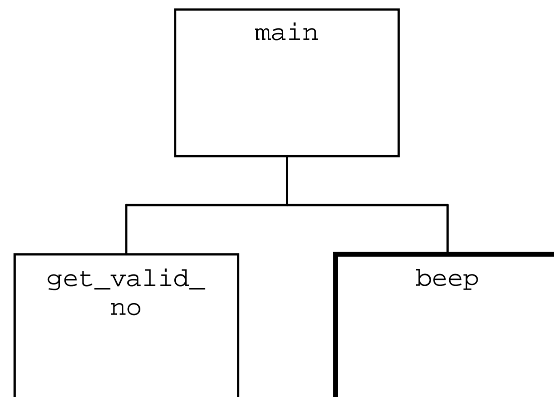
5. Function `get_valid_no` stops. Function `main` stores `get_valid_no`'s return value in variable `Nbeeps` and carries on until it reaches the call to function `beep`.



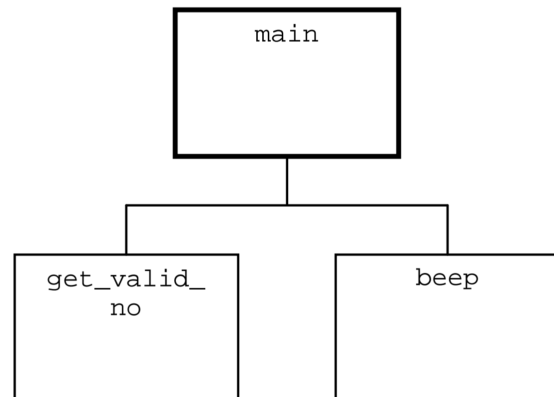
6. Function `main` passes the value of the argument `Nbeeps` to function `beep`.



7. Function `main` stops. Function `beep` starts, processing the data that `main` has passed to it.



8. Function `beep` stops, and returns control to function `main`.



- A function is written once but can be used many times
  - Even in different programs.
  - It would only have to be tested once.
  - It would work the same way every time.
  - This is very convenient.
- Example:
  - Using `get_valid_no` to return a student's mark between 0 and 100:
    `mark = get_valid_no(0,100);`
    avoiding magic numbers (better):
    `mark = get_valid_no(MINMARK,MAXMARK);`
  - This piece of code:
    - Calls function `get_valid_no`.
    - Passes the values of `MINMARK` (0) and `MAXMARK` (100) to the function as arguments.
    - Stores the return value (result) of the function in variable `mark`.
- Functions might have no arguments (need no input)

```
/*
 * roll_die.c
 * Simulates rolling a die.
 * S.P.Platt Last updated 30/8/2003
 */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/*function prototypes*/
int get_valid_no(int min, int max);
int roll_die(void);

int main(void)
{
    int i,rolls,spots;

    printf("\nProgram roll_die simulates rolling a die\n\n");

    printf("How many rolls do you want?\n");
    rolls = get_valid_no(1,20);

    printf("\nRolling the die %d times...",rolls);
    for (i=0;i<rolls;i++)
    {
        spots = roll_die();
        printf("\nRolled a %d.",spots);
    }

    return 0;
}
```

© S.P. Platt 2001-2005

```
int get_valid_no(int min, int max)
/*
 * Get a number from the keyboard
 * Writes to screen and reads from keyboard
 * Returns a number between min and max
 */
{
    int result;

    do
    {
        printf("Enter a number between %d and %d: ",min,max);
        scanf("%d",&result);
    }
    while ((result<min) || (result>max));
    return result;
}

int roll_die(void)
/*
 * Roll a die
 * Returns random number between 1 and 6
 * Automatically seeds random number generator first time round
 */
{
    static int firstgo=1;

    if (firstgo)
    {
        srand(time(NULL));
        firstgo=0;
    }
    return (rand()%6)+1;
}
```

- Using function `roll_die` to simulate the roll of a die:
  `spots = roll_die();`
  - Calls function `roll_die`.
  - Passes *no data* to the function – the empty brackets, (), are required.
  - Stores the return value (result) of the function in variable `spots`.

- Functions might have no return value (calculate no number):
  - `beeps` is an example:
    `beeps(10);`
    - Calls function `beeps`.
    - Passes the value 10 to the function.
    - *Does nothing* with the return value – because there isn't one.

References

- Dawson
  - A9.1-9.5
- Kernighan & Ritchie
  - 1.7

18 Function arguments and return values

- Function prototypes specify the *type* and *number* of
  - Arguments (*input* data)
  - Return values (*output* data)
- Example: converting from Fahrenheit to Celsius (see FtoC.c)
  `double FtoC(double TFar);`
  - There is one *argument*, a double representing the Fahrenheit temperature.
  - The *type* of the return value ("of the function") is double. The function will calculate and return a double-precision value
- Example: simulating the roll of a die (see roll_die.c)
  `int roll_die(void);`
  - There are *no arguments*. This is indicated by the reserved word, `void`.
  - The *type* of the function is `int`. The function will calculate and return an integer value.
- Example: making the computer go "beep" (see beeps.c):
  `void beep(int beeps);`
  - There is *one argument*, an integer specifying the number of times to go "beep".
  - The *type* of the function is `void`. This means that there is no return value - the function does *not* calculate a number.
- Example: reading a number between limits (see beeps.c)
  `int get_valid_no(int min, int max)`
  - There are *two arguments*,
    - An integer specifying the minimum valid number
    - An integer specifying the maximum valid number
  - The *type* of the function is `int`. The function will calculate and return an integer value.

18.1 Processing arguments

- Example

```
double FtoC(double TF)
/*
 * Converts Fahrenheit to Celsius
 * Argument is Fahrenheit temperature
 * Returns Celsius temperature
 */
{
    double TC;

    TC = (TF-32)*5/9;

    return TC;
}
```

- `TF` is the argument to function `FtoC`
  - It is a variable which function `FtoC` processes
  - It is declared in the function interface
    `double FtoC(double TF)`

© S.P. Platt 2001-2005

- The value of the argument is given when the function is called
    - By the calling function (e.g. `main`)
    - It is *not* read in from the keyboard
    - This is called “passing the argument”:
      `TCel=FtoC(100.0);`
        - Sets the value of `TF` to 100.0 (“passes the value 100.0 to `FtoC`”)
        - `FtoC` will work out the Celsius equivalent of 100.0°F.
- Example

```
void beep(int beeps)
/*
 *Makes the computer go "beep"
 *Argument is number of times to go "beep".
 */
{
    int i;
    long start,stop;

    for (i=0;i<beeps;i++)
    {
        /*wait a second*/
        start = clock();
        stop=start+1000;    /*1000 ms*/
        do
        {
            /*nothing*/
        }
        while (clock()<stop);

        /*go "beep"*/
        putchar('\a');
   }
}
```

- `beeps` is the argument to function `beep`
    - It is a variable which function `beep` processes
    - It is declared in the function interface
      `void beep(int beeps)`
- The value of the argument is given when the function is called
  `beep(10);`
    - Sets the value of `beeps` to 10.
    - `beep` will make the computer go “beep” 10 times.

© S.P. Platt 2001-2005

- Example:

```
int get_valid_no(int min, int max)
/*
 * Get a number from the keyboard
 * Writes to screen and reads from keyboard
 * Returns a number between min and max
 */
{
    int result;

    do
    {
        printf("Enter a number between %d and %d: ",min,max);
        scanf("%d",&result);
    }
    while ((result<min) || (result>max));

    return result;
}
```

- `min` and `max` are the arguments to `get_valid_no`
  - They are variables which function `get_valid_no` processes
  - They are declared in the function interface
    `int get_valid_no(int min, int max)`
- Their values are determined when the function is called
  - The first value is assigned to min
  - The second value is assigned to max
    `mark = get_valid_no(0,100);`
  - Sets the value of `min` to 0
  - Sets the value of `max` to 100.
  - `get_valid_no` will read a number between 0 and 100.

18.2 Local variables

- Functions also use *local variables*
  - `FtoC` used a double called `TC`
  - `beeps` used an integer called `i`, and long integers `start` and `stop`.
  - `get_valid_no` used a integer called `result`
- These are declared *inside* the function (between the { and the })
- Only that function can use that data (see also p. 103)

18.3 Return values

- In function `FtoC`
  `return TC;`
  - `TC` is a local variable - holds Celsius temperature
  - `return` statement causes function to stop processing
    - Returns control to calling function (e.g. `main`)
    - Gives (“returns”) the value of `TC` to the calling function

© S.P. Platt 2001-2005

- In function `get_valid_no`
  `return result;`
  - `result` is a local variable - holds value read from keyboard
  - `return` statement causes function to stop processing
    - Returns control to calling function (e.g. `main`)
    - Returns the value of `result` to the calling function
- In function `roll_die`
  `return nspots;`
  - `nspots` is a local variable – the number of spots on the face of the die
  - `return` statement causes function to stop processing
    - Returns control to calling function (e.g. `main`)
    - Returns the value of `nspots` to the calling function
- There can be more than one `return` statement in a function
  - For example in each branch of an if-else statement
  - The function will stop operating the first time `return` is reached.
- Return value type
  - *Must* match type of function
    - `TC` is a double in `FtoC`
    - `result` is an int in `get_valid_no`
    - `nspots` is an int in `rolldie`
  - `void` functions do not need a return statement (see `beeps`)

References

- Dawson
  - A9.10-9.13, 9.15-9.19
- Kernighan & Ritchie
  - 1.8, 4.2

19 Data scope and storage class

- *Scope* - which parts of a program can "see" a variable
- *Storage class* - whether variables are stored temporarily or permanently

19.1 Local and global data

- Functions are the C implementation of *modules*.
  - Modules are *self-contained* pieces of code.
- Functions control their own data – they use *local variables*
  - Not shared with other functions.
- Example - function `beep`:

```
void beep(int beeps)
/*
 *Makes the computer go "beep"
 *Argument is number of times to go "beep".
 */
{
    int i;
    long start,stop;

    for (i=0;i<beeps;i++)
    {
        /*wait a second*/
        start = clock();
        stop=start+1000;    /*1000 ms*/
        do
        {
            /*nothing*/
        }
        while (clock()<stop);

        /*go "beep"*/
        putchar('\a');
   }
}
```

- Variables `i`, `start` and `stop`

  ```
  int i;
  long start,stop;
  ```

  - Declared *inside* `beep`
  - Not known outside `beep` (i.e. in other functions)
  - *Local* to `beep`
  - In `beep` *scope* – only `beep` can "see" (and therefore use) them.
- *Local* variables are defined *inside* a function
  - `main`'s data is known only to `main`
  - `beep`'s data is known only to `beep`
  - Data are passed between functions
    - Arguments for input
    - Return values for output
  - Use of local data is key to writing modular programs

- *Global* variables are defined outside *all* functions (including `main`)
  - Can be used by *all* functions
  - All functions can "see" global data – global *scope*.
- Modular programs avoid using global data
  - Choice of names is easier with local data
  - Programs are easier to understand when local variables are used.
  - Unwanted side effects are common with global data
  - Effects of faults is limited with local data
- Global data has few applications
  - Destroys modularity - functions no longer self-contained
  - Lazy programming technique

19.2 Automatic and static data

- Automatic data
  - New copy created each time function called
  - Initial value undefined each time
  - Default case
- Static data
  - Use same copy each time
  - Value kept from call to call
  - Use keyword `static`
- Example - function rolldie

```
int rolldie(void)
/*
 *Roll a die
 *Returns random number between 1 and 6
 *Automatically seeds random number generator 1st time round
 */
{
    int nspots;
    static int firstgo=1;

    if (firstgo)
    {
        srand(time(NULL)); /*initialise random no generator*/
        firstgo=0;
    }

    nspots = rand()%6+1;
    return nspots;
}
```

- Analysis

  `int nspots;`
  - Declares an *automatic* variable to store the number of spots. This is by nature different each time the function runs

  `static int firstgo=1;`
  - Declares a *static* variable to indicate whether it's the first go, and initialises it to 1 ("true"). A record needs to be kept between function calls.

© S.P. Platt 2001-2005

```
if (firstgo)
{
    srand(time(NULL)); /*initialise random no generator*/
    firstgo=0;
}
```

- On the first call of `rolldie`, `firstgo` is true. The random number generator is initialised and `firstgo` set to false.
- On subsequent calls, `firstgo` will be false, and the random number generator will *not* be reinitialised.

References

- Dawson
  - A9.6-9.9, 11.2, 11.4
- Kernighan & Ritchie
  - 4.4, 4.6

20 Pass by value and pass by reference

20.1 Formal and actual arguments

- Function *arguments* are the *input* data to a function
    - The data that are passed to it when it starts.
- From the *called* function's point of view, the arguments are called "*formal* arguments".
- From the calling function's point of view, they are "*actual* arguments".
- Formal and actual arguments are variables in different *scope*
    - The actual argument is in the *calling function's scope*
        - Only the calling function sees it.
    - The formal argument is in the *called function's scope*
        - Only the called function sees it.

20.2 Pass by value

- When a function is called, C *copies* the actual arguments' values to the formal arguments.
    - This is known as "pass by value" or "call by value".
- Formal arguments are *local* variables with the value of the actual argument.
    - They are only *copies* of the actual arguments
    - If their values are changed, the values of the actual arguments are unaffected.
    - They are *temporary* variables – they are destroyed when the function returns.
- Example

```
/*
 * beeps.c
 * Makes the computer go "beep"
 * S.P.Platt Last updated 30/8/2003
 */
#include <stdio.h>
#include <time.h>

int get_valid_no(int min, int max);
void beep(int beeps);

int main(void)
{
    int beeps;

    printf("\nProgram beeps makes the computer go \"beep\"\n\n");

    printf("How many beeps do you want?\n");
    beeps = get_valid_no(1,10);

    printf("\nI am about to go \"beep\" %d times...",beeps);
    beep(beeps);
    printf("\nI hope you heard %d beeps.",beeps);

    return 0;
}
```

© S.P. Platt 2001-2005

```
int get_valid_no(int min, int max)
/*
 * Get a number from the keyboard
 * Writes to screen and reads from keyboard
 * Returns a number between min and max
 */
{
    int result;

    do
    {
        printf("Enter a number between %d and %d: ",min,max);
        scanf("%d",&result);
    }
    while ((result<min) || (result>max));

    return result;
}

void beep(int beeps)
/*
 *Makes the computer go "beep"
 *Argument is number of times to go "beep".
 *Counts down from beeps on screen - illustrates
 *local scope of function arguments
 */
{
    long start,stop;

    for (;beeps>0;beeps--)  /*counts down from initial value of beeps*/
    {
        printf("\n%d",beeps);

        /*wait a second*/
        start = clock();
        stop=start+1000;    /*1000 ms*/
        do
        {
            /*nothing*/
        }
        while (clock()<stop);

        /*go "beep"*/
        putchar('\a');
   }
}
```

- Results

```
prompt>beeps

Program beeps makes the computer go "beep"

How many beeps do you want?
Enter a number between 1 and 10: 3

I am about to go "beep" 3 times...
3
2
1
I hope you heard 3 beeps.
prompt>
```

© S.P. Platt 2001-2005

- Analysis

	In `main` scope (*actual* argument)	In `beep` scope (*formal* argument)
1: `main` creates variable `beeps`	**beeps**	
	?	
2: `main` assigns a value to `beeps`	**beeps**	
	3	
3: `main` calls `beep`. `beep` creates a temporary variable, also called `beeps`	beeps	**beeps**
	3	**?**
4: `main` passes *value* of *actual argument* `beeps` to `beep`. `beep` stores its value in *formal argument* `beeps`, which it then processes	beeps	**beeps**
	3	**3**
	3	**2**
	3	**1**
	3	**0**
5: `beep` destroys temporary variable `beeps`. Control is returned to `main`. Actual argument `beeps` is unaffected.	**beeps**	Gone!
	3	

- Example:

```
/*
File:           avg.c
Author:         S.P. Platt
Last edited:    30/8/2003
Description:    Tests function avg.
*/
#include <stdio.h>

double avg(int a, int b);

int main(void)
{
   int x,y;
   double z;

   printf("Enter two integers ");
   scanf("%d %d",&x,&y);
   z=avg(x,y);
   printf ("The average of %d and %d is %.2f",x,y,z);

   return 0;
}
```

© S.P. Platt 2001-2005

```
double avg(int a, int b)
/*calculates average of two integers, a and b*/
{
   double ans;

   ans = (a+b)/2.0;

   return ans;
}
```

- Results:

```
prompt> avg
Enter two integers 1 99
The average of 1 and 99 is 50.00
prompt>
```

- Analysis:

	In `main` scope			In `avg` scope		
1: `main` creates variables `x`, `y` and `z`	**x**	**y**	**z**			
	?	**?**	**?**			
2: `main` assigns values to `x` and `y`	**x**	**y**	**z**			
	1	**99**	**?**			
3: `main` calls `avg`. `avg` creates temporary variables `a`, and `b` and `ans`	x	y	z	**a**	**b**	**ans**
	1	99	?	**?**	**?**	**?**
4: `main` passes values of `x` and `y` to `avg`. `avg` stores values in `a` and `b`.	x	y	z	**a**	**b**	**ans**
	1	99	?	**1**	**99**	**?**
5: `avg` uses the values stored in `a` and `b` to determine the value of `ans`.	x	y	z	**a**	**b**	**ans**
	1	99	?	**1**	**99**	**50.0**
6: `avg` passes value of `ans` to `main`. `main` stores returned value in `z`. `avg` destroys temporary variables `a`, `b` and `ans`	**x**	**y**	**z**	Gone!		
	1	**99**	**50.0**			

- `avg` only processes *temporary copies* of the actual arguments `x` and `y`
- This is very important:
  - `avg` cannot affect data used by the calling function

© S.P. Platt 2001-2005

20.3 Pass by reference

- Using pass by *value*
  - The *input* to a function is via its *arguments*
  - The *output* is via the *return value*
- Functions process temporary *copies* of their input data
  - Side effects are avoided.
- *Arrays* do not have a single value
  - *Arrays cannot be passed by value*
- C functions return at most one value
  - Functions cannot return *arrays*.
  - Functions cannot return *more than one scalar* (i.e. non-array) value.
- We can overcome these limitations using *pass by reference* (a.k.a. "*call* by reference")
  - The *address* (in memory - a "reference" to the argument) is passed instead of its value.
  - The function processes the data *referred to* by the argument - the *original*, not a copy.
  - Side effects are more likely.

References

- Dawson
  - A9.10-9.12, 9.22, 9.23
- Kernighan & Ritchie
  - 5.2, 5.3

© S.P. Platt 2001-2005

21 Pass by reference

21.1 Array input to functions

- Example:

```
/*
File:         namelen.c
Author:       S.P.Platt
Last edited: 31/8/2003
Description: demonstrates alternative strlen implementation
*/
#include <stdio.h>
int slen(char *str);

int main(void)
{
    char name[100];
    int len;

    puts("Program namelen will count the characters in your name.");
    printf("What is your name? ");
    scanf("%99s",name);

    len = slen(name);
    printf("Your name, %s, has %d characters.",name,len);

    return 0;
}

int slen(char *str)
/*
Description:   calculates length of string
Arguments:
   str: reference to null ('\0') terminated character array
Return value:  length of str
*/
{
   int i=0;

   while(str[i]!='\0')
   {
      i++;
   }

   return i;
}
```

- Function prototype:
  `int slen(char *str);`
  - The input to this function is a character string.
  - The argument is a character array, called `str`.
  - *Either* **`char *str`** *or* **`char str[]`** can be used to specify pass by reference.
    - The empty brackets, **`[]`**, specify that the argument is an array (but not how large).
    - The asterisk, **`*`**, specifies that it is an *address* (a “pointer”).
  - The function processes *original* data
    - `str[i]` refers to the `i`th element of the *actual* argument
  - The output of this function is an int.

© S.P. Platt 2001-2005

- Function call
  `len = slen(name);`
  - `name` is the name of the character array
    - The name of the array represents the *address* of the first element
    - This represents *pass by reference*
  - The result of the function is stored in `len`.
- Function definition
  `while(str[i]!='\0')`
  - `str` refers to the *original* data (in `name`), *not* copies
  - `str[i]` refers to the ith element in `name`.
- Result:

```
prompt> namelen
Program namelen will count the characters in your name.
What is your name? Benjy
Your name, Benjy, has 5 characters.
prompt>
```

© S.P. Platt 2001-2005

• Analysis:	In `main` scope							In `slen` scope	
1: `main` creates `name` and `len`	**name**						**len**		
	?	?	?	?	?	?	?		
2: `main` assigns a value to `name`	**name**						**len**		
	'B'	'e'	'n'	'j'	'y'	'\0'	?		
3: `main` calls `slen`. `slen` creates i and *refers to* `name` by `str`	name						len	**i**	**str**
	'B'	'e'	'n'	'j'	'y'	'\0'	?	0	&name
4: `slen` processes local variable `i`, and *original version* of `name` (a.k.a. `str`)	name						len	**i**	**str[i]**
	'B'	'e'	'n'	'j'	'y'	'\0'	?	0	'B'
	'B'	'e'	'n'	'e'	'y'	'\0'	?	1	'e'
	'B'	'e'	'n'	'n'	'y'	'\0'	?	2	'n'
	'B'	'e'	'n'	'j'	'y'	'\0'	?	3	'j'
	'B'	'e'	'n'	'y'	'y'	'\0'	?	4	'y'
	'B'	'e'	'n'	'\0'	'y'	'\0'	?	5	'\0'
5: `slen` destroys `i` and *reference* `str` to `name`. `name` itself is *unaffected*. Control is returned to `main`.	**name**						**len**		
	'B'	'e'	'n'	'j'	'y'	'\0'	5		

© S.P. Platt 2001-2005

21.2 Array output from functions

- Example

```
/*
File:          sort.c
Author:        S.P. Platt
Last edited:   31/8/2003
Description:   Bubble sort example
*/
#include <stdio.h>
void bsort(int *data, int Ndata);

int main(void)
{
   int i;
   int list[5]={5,4,3,2,1};

   printf("Unsorted list is:\n");
   for (i=0;i<5;i++)
   {
      printf("%d ",list[i]);
   }
   printf("\n");

   bsort(list,5);

   printf("Sorted list is:\n");
   for (i=0;i<5;i++)
   {
      printf("%d ",list[i]);
   }
   printf("\n");

   return 0;
}

void bsort(int *data, int Ndata)
/*
Description: bubble sort
Arguments:
   data: reference to data to be sorted
   Ndata: no of items to be sorted
*/
{
   int i,j;
   int temp;

   for (i=0;i<Ndata-1;i++)
   {
      for (j=Ndata-2;j>i-1;j--)
      {
         if (data[j]>data[j+1])
         {
            temp=data[j];
            data[j]=data[j+1];
            data[j+1]=temp;
         }
      }
   }
}
```

© S.P. Platt 2001-2005

- Function prototype:
  `void bsort(int *data, int Ndata);`
  - The data to be sorted is an array of integers.
  - The arguments are
    - A reference to the array, called `data`.
    - *Either* **`int data[]`** *or* **`int *data`** could have been used.
    - The number of elements in the array, `Ndata`.
  - The output of this function is *the sorted array*
    - This cannot come via a return value
    - It *must* come via the argument passed by reference
    - There is no return value used – the function type is `void`.
  - The function processes *original* data
    - `data[i]` refers to the `i`th element of the *actual* argument
- Function call
  `bsort(list,5);`
  - `list` is the name of the array being sorted
    - The name of the array represents the *address* of the first element
    - List is passed by *reference*
  - 5 is the number of elements in the array
    - This is passed by *value*
  - The *original* data – i.e. the contents of `list` – are *altered*

```
printf("Unsorted list is:\n");
…
printf("%d ",list[i]);
…
bsort(list,5);
…
printf("Sorted list is:\n");
…
printf("%d ",list[i]);
```

- Result:

```
prompt> sort
Unsorted list is:
5 4 3 2 1
Sorted list is:
1 2 3 4 5

prompt>
```

© S.P. Platt 2001-2005

• Analysis:	In `main` scope					In `bsort` scope	
1: `main` creates list	**list**						
	5	**4**	**3**	**2**	**1**		
3: `main` calls `bsort`. `bsort` creates `i` and *refers to* `list` by `data`	list					**i**	**data**
	5	4	3	2	1	**0**	**&list**
4: `bsort` processes *original version* of `list` (a.k.a. `data`)	list					**i**	**data**
	1	**5**	**4**	**3**	**2**	**1**	**&list**
	1	**2**	**5**	**4**	**3**	**2**	**&list**
	1	**2**	**3**	**5**	**4**	**3**	**&list**
	1	**2**	**3**	**4**	**5**	**4**	**&list**
5: Control is returned to `main`. `bsort` has modified *original* `list`.	**list**						
	1	**2**	**3**	**4**	**5**		

21.3 Passing constant data by reference

- Pass by reference allows a function to alter its *actual arguments* (the *original data*)
  - o Programmers sometimes make mistakes
    - ▪ Data can be altered *inadvertently* when passed by reference
  - o Specify that arguments should not be altered using `const`
- Example:

```
int slen(const char *str)
/*
Description:   calculates length of string
Arguments:
   str: reference to null ('\0') terminated character array
Return value:  length of str
*/
{
   int i=0;

   while(str[i]!='\0')
   {
      str[i]++;   //Error!  Should be i++.  slen should not modify str.
   }

   return i;
}
```

- Compiler prevents `const` data from being modified:

```
prompt> bcc32 namelen.c
Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland
namelen.c:
Error E2024 namelen.c 40: Cannot modify a const object in function slen
*** 1 errors in Compile ***

prompt>
```

21.4 Multiple scalar output data from functions

- Pass by reference can also be used when multiple scalar output data are required:

```
void bsort(int *data, int Ndata)
/*
Description: bubble sort
Arguments:
   data: reference to data to be sorted
   Ndata: no of items to be sorted
*/
{
   int i,j;

   for (i=0;i<Ndata-1;i++)
   {
      for (j=Ndata-1;j>i-1;j--)
      {
         if (data[j]>data[j+1])
         {
            swap(&data[j],&data[j+1]);
         }
      }
   }
}

void swap(int *a, int *b)
/*
Description: Swaps two integers
Arguments:
   a, b: references to data to be swapped
*/
{
   int temp;

   temp = *a;
   *a = *b;
   *b = temp;
}
```

- Function `swap` swaps the values of its arguments
  - It changes *two* things
  - It cannot return them *both* as return values
  - It must use *pass by reference* instead.
- In the *declaration* `*a` and `*b` indicate that the addresses of (references to), the input data, *not* their values, are to be passed to the function.
- In the body of the function, the asterisks indicate “the value at the address”, so that `*a` is “the value at address `a`” and `*b` is “the value at address `b`”.
- The function may change the values at the specified addresses of the input data.

© S.P. Platt 2001-2005

- The arguments must be the addresses, *not* the values, of the input data:
  `swap(&x,&y);`
  - `&x` means "the address of variable `x`".
  - `&y` means "the address of variable `y`".
- Pass by reference is used like this in `scanf`
  `scanf("%d%s",&N,str);`
  - `&N` means "the address of variable `N`" (which must be an integer).
  - `str` specifies the address of a character *array*, `str`
    - The name of an array represents its address even without the `&`.
    - `&str` would be equally acceptable

References

- Dawson
  - A2.9, 9.10-9.12, 9.22, 9.23
- Kernighan & Ritchie
  - 5.2, 5.3

© S.P. Platt 2001-2005

22 Testing and debugging programs

- *Testing* is trying to show that a program (or module) works correctly
- It is generally not possible to *prove* that a program works.
    - Proceed by trying to make it fail – and hoping it doesn't
- The harder it is to make a program fail during testing, the more likely it will be to work during normal use
- Think carefully about how the program might fail
    - Test at likely points of failure (e.g. invalid input data).
- If the program does fail, it has some fault that needs to be corrected.
- *Debugging* is the art of fixing a faulty program

22.1 Testing

- A program should be tested *incrementally*
    - Each part should be tested individually before being integrated with the rest.
    - Design from the top down – test from the *bottom up*.
- Testing is closely allied to modular programming,
    - Each module should have a well-defined purpose, against which it can be tested.

22.1.1 A simple example

- This program tests the operation of the standard function `toupper`:

```
/*
 * testtoupper.c
 * EL1113 PDI example
 * Testing standard function toupper
 * S.P.Platt Last edited 1/9/2003
 */
#include <stdio.h>
#include <ctype.h>
#define MINCHAR 0
#define MAXCHAR 255
#define NLINES 16

int main(void)
{
    int i;

    printf("Program testtoupper tests standard function toupper.\n");

    for (i=MINCHAR;i<=MAXCHAR;i++)
    {
        printf("\nCharacter '%c' (index %d) becomes ",i,i);
        printf("character '%c', (index %d).",toupper(i),toupper(i));
        /*pause after NLINES lines*/
        if ((i-MINCHAR)%NLINES==NLINES-1)
        {
            printf("\nPress RETURN to continue");
            getchar();
        }
    }

    return 0;
}
```

© S.P. Platt 2001-2005

- The testing is *comprehensive* and *systematic*:
  - All valid character values (whether lower case or not) are passed to the function in turn.
  - The input data and the output data (upper case equivalent) are printed to the screen for checking.
  - The user can check that `toupper` correctly processes each character value
- Example results:

```
prompt>testtoupper
Program testtoupper tests standard function toupper.

Character ' ' (index 0) becomes character ' ', (index 0).
Character '☺' (index 1) becomes character '☺', (index 1).
Character '☻' (index 2) becomes character '☻', (index 2).
Character '♥' (index 3) becomes character '♥', (index 3).

…

Character '^' (index 94) becomes character '^', (index 94).
Character '_' (index 95) becomes character '_', (index 95).
Press RETURN to continue

Character '`' (index 96) becomes character '`', (index 96).
Character 'a' (index 97) becomes character 'A', (index 65).
Character 'b' (index 98) becomes character 'B', (index 66).
Character 'c' (index 99) becomes character 'C', (index 67).

…

Character '²' (index 253) becomes character '²', (index 253).
Character '■' (index 254) becomes character '■', (index 254).
Character ' ' (index 255) becomes character ' ', (index 255).
Press RETURN to continue

prompt>
```

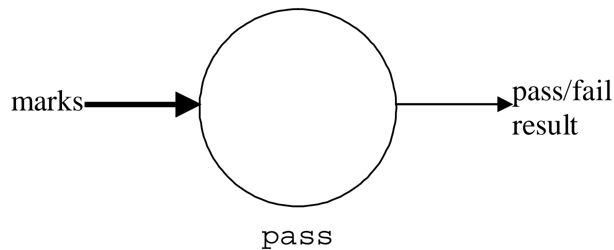
- See how
  - All valid characters are tested (including “nonprinting” characters)
  - Lower case characters are converted to upper case
  - All other characters are not converted.

22.1.2 Choosing a test set

- To test `toupper` we checked against all valid characters (from 0 to 255)
- This was comprehensive, but not *exhaustive*
  - How can you test against all possible *invalid* characters (i.e. outside the valid range – e.g. indices –1234 or 43295?).
  - A *robust* test procedure needs to do this
- In practice we must be selective
- Choose a suitable *test set*
  - A *subset* of the possible input data values that capture their significant features

© S.P. Platt 2001-2005

- Consider this function




```
int pass(int *marks);
```

  - Function `pass` determines whether a student has passed or failed a year's study.
  - It reads an array of six marks
  - It returns a boolean variable which is set to 1 (true) if the student has passed; 0 (false) if he has failed or if the marks are invalid. To pass, all marks must be at least 40%.
- Each mark could take values from 0 to 100 inclusive
  - There are over a billion (1,000,000,000,000) valid combinations of marks.
  - There could also be any number of combinations of invalid marks.
  - The function would have to deal correctly with them all, and be adequately tested.
- Appropriate test cases must be selected, concentrating on where the function is likely to fail, i.e.
  - Around *boundary values* (in this case minimum and maximum marks)
  - Around *threshold values* (in this case, pass/fail borderline marks)
  - Around *array boundaries* (i.e. the first and last array elements)
- We must also ensure that all parts (i.e. all braches) of the function are tested by the test set.
- We must apply these test cases and check the actual results against expected ones.
- The following test set consists of just 19 cases
  - These are concentrated where failure is likely:
    - At the extremes of validity (0% and 100%).
    - Around the pass/fail threshold (40%).
    - At the edges of the array (mark 1 and mark 6).
  - This captures all the significant behaviour of the function.
  - These are *not* the most likely marks!

Test case	Test data						Expected result
	Mark 1	Mark 2	Mark 3	Mark 4	Mark 5	Mark 6	
1	50	50	50	50	50	50	Pass
2	39	50	50	50	50	50	Fail
3	40	50	50	50	50	50	Pass
4	50	50	50	50	50	39	Fail
5	50	50	50	50	50	40	Pass
6	50	50	50	39	50	50	Fail
7	50	50	50	40	50	50	Pass
8	-1	50	50	50	50	50	Error
9	0	50	50	50	50	50	Fail
10	50	50	50	50	50	-1	Error
11	50	50	50	50	50	0	Fail
12	50	50	50	-1	50	50	Error
13	50	50	50	0	50	50	Fail
14	101	50	50	50	50	50	Error
15	100	50	50	50	50	50	Pass
16	50	50	50	50	50	101	Error
17	50	50	50	50	50	100	Pass
18	50	50	50	101	50	50	Error
19	50	50	50	100	50	50	Pass

22.1.3 Test programs

- Write test programs to automate test procedures where possible
- A test program for function `pass` is shown below. See how:
  - Arrays of test data are set up and initialised to contain the values of the test set. (Alternatively, the test data could be read from files.)
  - This is more convenient and reliable than typing by hand
- The results of the test program can be recorded and compared with the expected results.
- If they match, and the test set is well chosen, correct operation of the function is *verified*.
  - If not, the function must be modified and the test repeated.
- The process is complete when the test results match the expectations.
  - We say that the function is *validated against the test set*.

© S.P. Platt 2001-2005

- The test procedure is therefore:
  - Specify the test requirements for each module at the design stage, including a test set against which the module will be validated.
  - Implement a test program to exercise the module against that test set.
  - Iteratively test and revise the module until it is validated against the test set.
- If you modify a validated program, make sure the new results are consistent with the old.
  - This process is known as “regression testing”.
- The following test program source listing contains some deliberate errors in function `pass`:

```
/*
 * testpass.c
 * EL1113 PDI example
 * Testing function pass
 * (determines assessment pass or failure)
 * S.P.Platt Last modified 1/9/2003
 */
#include <stdio.h>

int pass(int *marks);

#define CASES 19
#define MARKS 6

#define MINMARK 0
#define MAXMARK 100
#define PASSMARK 40

int main(void)
{
   int i,j;
   int marks[CASES][6]=
         //  marks ...                     test case ...
         //  1   2   3    4   5    6
         {{ 50, 50, 50,  50, 50,  50}, //  1
          { 39, 50, 50,  50, 50,  50}, //  2
          { 40, 50, 50,  50, 50,  50}, //  3
          { 50, 50, 50,  50, 50,  39}, //  4
          { 50, 50, 50,  50, 50,  40}, //  5
          { 50, 50, 50,  39, 50,  50}, //  6
          { 50, 50, 50,  40, 50,  50}, //  7
          { -1, 50, 50,  50, 50,  50}, //  8
          {  0, 50, 50,  50, 50,  50}, //  9
          { 50, 50, 50,  50, 50,  -1}, // 10
          { 50, 50, 50,  50, 50,   0}, // 11
          { 50, 50, 50,  -1, 50,  50}, // 12
          { 50, 50, 50,   0, 50,  50}, // 13
          {101, 50, 50,  50, 50,  50}, // 14
          {100, 50, 50,  50, 50,  50}, // 15
          { 50, 50, 50,  50, 50, 101}, // 16
          { 50, 50, 50,  50, 50, 100}, // 17
          { 50, 50, 50, 101, 50,  50}, // 18
          { 50, 50, 50, 100, 50,  50}};// 19

   printf("Program testpass tests function pass - ");
   printf("determining\nassessment pass or failure.\n");
```

© S.P. Platt 2001-2005

```
   for (i=0;i<CASES;i++)
   {
      printf("\ncase: %2d, marks ",i+1);
      for (j=0;j<MARKS;j++)
      {
         printf("%4d ",marks[i][j]);
      }
      if (pass(marks[i]))
      {
         printf (" PASS");
      }
      else
      {
         printf(" FAIL/ERROR");
      }
   }

   return 0;
}

int pass(int *marks)
/*
Description:
   Determines pass/failure of a study year.
   Checks all marks for validity and writes message
   to stderr for each mark not between MINMARK and MAXMARK.
   Student passes if all marks valid and not less than PASSMARK
Arguments:
   marks: reference to array of MARKS marks
Return value:
   1 if student passes
   0 if student fails/marks invalid
*/
{
   int i;
   int error=0,fail=0;

   for (i=1;i<6;i++)
   {
      //check for invalid input
      if ((marks[i]<MINMARK) || (marks[i]>MAXMARK))
      {
         fprintf(stderr,
         "\nError in function pass: mark %d invalid (%d).",i+1,marks[i]);
         error=1;
      }
      //check for failure
      else if (marks[i]<=PASSMARK)
      {
         fail=1;
      }
   }

   return !(error&&fail);
}
```

- The results of the test program don't match the test set. There are three errors in function `pass`. Can you fix them?

```
prompt> testpass
Program testpass tests function pass - determining
assessment pass or failure.

case:  1, marks   50   50   50   50   50   50  PASS
case:  2, marks   39   50   50   50   50   50  PASS
case:  3, marks   40   50   50   50   50   50  PASS
case:  4, marks   50   50   50   50   50   39  PASS
case:  5, marks   50   50   50   50   50   40  PASS
case:  6, marks   50   50   50   39   50   50  PASS
case:  7, marks   50   50   50   40   50   50  PASS
case:  8, marks   -1   50   50   50   50   50  PASS
case:  9, marks    0   50   50   50   50   50  PASS
case: 10, marks   50   50   50   50   50   -1
Error in function pass: mark 6 invalid (-1). PASS
case: 11, marks   50   50   50   50   50    0  PASS
case: 12, marks   50   50   50   -1   50   50
Error in function pass: mark 4 invalid (-1). PASS
case: 13, marks   50   50   50    0   50   50  PASS
case: 14, marks  101   50   50   50   50   50  PASS
case: 15, marks  100   50   50   50   50   50  PASS
case: 16, marks   50   50   50   50   50  101
Error in function pass: mark 6 invalid (101). PASS
case: 17, marks   50   50   50   50   50  100  PASS
case: 18, marks   50   50   50  101   50   50
Error in function pass: mark 4 invalid (101). PASS
case: 19, marks   50   50   50  100   50   50  PASS
prompt>
```

- Results when output redirected to file pass.dat (error messages only seen on screen):

```
prompt> testpass > pass.dat

Error in function pass: mark 6 invalid (-1).
Error in function pass: mark 4 invalid (-1).
Error in function pass: mark 6 invalid (101).
Error in function pass: mark 4 invalid (101).
prompt>
```

- Contents of file pass.dat (error messages omitted):

```
Program testpass tests function pass - determining
assessment pass or failure.

case:  1, marks   50   50   50   50   50   50  PASS
case:  2, marks   39   50   50   50   50   50  PASS
case:  3, marks   40   50   50   50   50   50  PASS
case:  4, marks   50   50   50   50   50   39  PASS
case:  5, marks   50   50   50   50   50   40  PASS
case:  6, marks   50   50   50   39   50   50  PASS
case:  7, marks   50   50   50   40   50   50  PASS
case:  8, marks   -1   50   50   50   50   50  PASS
case:  9, marks    0   50   50   50   50   50  PASS
case: 10, marks   50   50   50   50   50   -1  PASS
case: 11, marks   50   50   50   50   50    0  PASS
case: 12, marks   50   50   50   -1   50   50  PASS
case: 13, marks   50   50   50    0   50   50  PASS
case: 14, marks  101   50   50   50   50   50  PASS
case: 15, marks  100   50   50   50   50   50  PASS
case: 16, marks   50   50   50   50   50  101  PASS
case: 17, marks   50   50   50   50   50  100  PASS
case: 18, marks   50   50   50  101   50   50  PASS
case: 19, marks   50   50   50  100   50   50  PASS
```

© S.P. Platt 2001-2005

22.2 Debugging

- A "bug" is a computer program fault that occurs at run time:
  - The program may crash
  - The program may produce incorrect results – requiring careful testing
- Debugging is the process of correcting faults.
- Debugging can be made easier by careful modular design and implementation.
  - Much less can go wrong in a small, self-contained module
  - When it does, it is easier to spot and correct.
- It is also important to maintain a disciplined programming style
  - Especially, to use consistent *indentation*.
- Different programming languages are prone to characteristic bugs of their own.
  - See p. 152 for description of some common C programming errors to watch out for.
- Here are some general guidelines to follow:
  - Make sure you know where the program went wrong. If necessary, add temporary print statements at appropriate points, for example.

    ```
    printf("\ntest1");
    …
    printf("\ntest2");
    ```

    This is especially useful for finding how far a program has got before a crash.
  - Find out what values variables have as the program progresses, for example:

    ```
    printf("\ntest1: count=%d",count);
    …
    printf("\ntest2: count=%d",count);
    ```

    Compare the actual values with those that you expect or, at least, what you think might be sensible values.
  - Isolate suspect statements in trivial programs, and keep them for future reference. For example, try debugging this simple, but faulty, program:

    ```
    int main(void)
    {
       double x;

       printf("enter x: ");
       scanf("%f",&x);
       printf("\n x is %f: ",x);

       return 0;
    }
    ```
  - When you find and correct a bug, look for the same pattern elsewhere in your code. You're likely to have made the same mistake elsewhere, especially if you have used copy and paste.
  - If a bug suddenly appears in your program, suspect the most recent change.
  - Make sure you debug promptly, but don't be in too much of a hurry. Don't make random or desperate changes, as this is as likely to make things worse as better.
  - Be systematic, and take notes as you debug, so you know what you have tried and what was unsuccessful.

© S.P. Platt 2001-2005

References

- Dawson
    - D1

23 Splitting programs among several source code files

- There are many advantages to be had from modular programming, e.g:
  - Modules (functions in C) can be reused, in the same or in different programs.
  - Programs can be implemented and tested more easily at the level of modules.
  - More than one person can work on a program at a time.
- To get the most out of these advantages program source code must be split into several files.
- C makes this easy to do.

23.1 A familiar example

- `testpass.c` contained the following code:
  - The function `pass`.
  - The main function, which performed the test on `pass`.
  - The prototype for function `pass`. This defined the function interface, and was therefore used by both `pass` and `main`.
- To get the full benefit of modularity, the program must be split among three files:
  - The function `pass` should go in one of these. A suitable name would be pass.c.
  - The main function requires a file of its own. It makes sense to keep the name testpass.c.
  - The function prototype should go in a header file, called pass.h
- pass.h

```
/*
 * pass.h
 * Header file for function pass
 * (determines assessment pass or failure)
 * S.P.Platt Last modified 1/9/2003
 */
#define MARKS 6
#define MINMARK 0
#define MAXMARK 100
#define PASSMARK 40

int pass(int *marks);
```

© S.P. Platt 2001-2005

- pass.c

```
/*
 * pass.c
 * Function pass determines assessment pass or failure
 * S.P.Platt Last modified 1/9/2003
 */
#include <stdio.h>
#include "pass.h"

int pass(int *marks)
/*
Description:
   Determines pass/failure of a study year.
   Checks all marks for validity and writes message
   to stderr for each mark not between MINMARK and MAXMARK.
   Student passes if all marks valid and not less than PASSMARK
Arguments:
   marks: reference to array of MARKS marks
Return value:
   1 if student passes
   0 if student fails/marks invalid
*/
{
   int i;
   int error=0,fail=0;

   for (i=0;i<MARKS;i++)
   {
      //check for invalid input
      if ((marks[i]<MINMARK) || (marks[i]>MAXMARK))
      {
         fprintf(stderr,
         "\nError in function pass: mark %d invalid (%d).",i+1,marks[i]);
         error=1;
      }
      //check for failure
      else if (marks[i]<PASSMARK)
      {
         fail=1;
      }
   }

   return !(error||fail);
}
```

© S.P. Platt 2001-2005

- testpass.c

```
/*
 * testpass.c
 * EL1113 PDI example
 * Test program for function pass
 * (determines assessment pass or failure)
 * S.P.Platt Last modified 1/9/2003
 */
#include <stdio.h>
#include "pass.h"

#define CASES 19

int main(void)
{
   int i,j;
   int marks[CASES][MARKS]=
         //  marks ...                      test case ...
         //  1   2   3    4   5    6
         {{ 50, 50, 50,  50, 50,  50}, //  1
          { 39, 50, 50,  50, 50,  50}, //  2
          { 40, 50, 50,  50, 50,  50}, //  3
          { 50, 50, 50,  50, 50,  39}, //  4
          { 50, 50, 50,  50, 50,  40}, //  5
          { 50, 50, 50,  39, 50,  50}, //  6
          { 50, 50, 50,  40, 50,  50}, //  7
          { -1, 50, 50,  50, 50,  50}, //  8
          {  0, 50, 50,  50, 50,  50}, //  9
          { 50, 50, 50,  50, 50,  -1}, // 10
          { 50, 50, 50,  50, 50,   0}, // 11
          { 50, 50, 50,  -1, 50,  50}, // 12
          { 50, 50, 50,   0, 50,  50}, // 13
          {101, 50, 50,  50, 50,  50}, // 14
          {100, 50, 50,  50, 50,  50}, // 15
          { 50, 50, 50,  50, 50, 101}, // 16
          { 50, 50, 50,  50, 50, 100}, // 17
          { 50, 50, 50, 101, 50,  50}, // 18
          { 50, 50, 50, 100, 50,  50}};// 19

   printf("Program testpass tests function pass - ");
   printf("determining\nassessment pass or failure.\n");

   for (i=0;i<CASES;i++)
   {
      printf("\ncase: %2d, marks ",i+1);
      for (j=0;j<MARKS;j++)
      {
         printf("%4d ",marks[i][j]);
      }
      if (pass(marks[i]))
      {
         printf (" PASS");
      }
      else
      {
         printf(" FAIL/ERROR");
      }
   }
   return 0;
}
```

- *Exactly the same code* that was formerly in the single file, testpass.c, has now been divided among three files.
- pass.h
  - Contains the prototype for function `pass`.
  - Contains some other information (the `#defines`) used by both pass.c and testpass.c.
- testpass.c:
  - Contains the main program, which uses the function `pass`.
  - Contains some other information that is used only by `main`, (the definition of `CASES`)
  - Includes stdio.h (to allow `printf` to be used in `main`)
  - Includes pass.h, (to allow `pass` to be used in `main`)
- pass.c
  - Contains the definition for function `pass`.
  - Includes stdio.h (to allow `fprintf` to be used in `pass`)
  - Includes pass.h. (to match the definition of `pass`)
- Use

  `#include <stdio.h>`
  - For standard header files.
  - `<>` tells the compiler to look for the file in a standard location.

  `#include "pass.h"`
  - For your own header files
  - "" tells the compiler to use a different search rule – you should put your .h files in the same directory as your .c files.

23.2 What goes where?

- To be most effective, it is important to ensure that you put the right information in the right files:
  - ".c" files should contain either
    - A main function, or
    - One or (usually) more other functions.
  - ".h" files contain information that needs to be used by *both* the main program *and* the called functions:
    - Function prototypes
- Both .c and .h files can also (and normally should) contain
  - Comments
  - Symbolic constant definitions (`#define` statements).

23.3 Compiling and linking code in separate files

- To compile several files at once:

```
prompt> bcc32 testpass.c pass.c
Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland
testpass.c:
pass.c:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

prompt>
```

- The compiler checks and translates each source code file into machine instructions ("object code")
  - pass.obj contains the machine instructions for function `pass`.
  - testpass.obj contains the machine instructions for the `main` program.
  - *Neither* object file contains instructions for the entire program.
- They need to be *linked* to achieve this.
  - `bcc32` takes the two object files, and links them together to create the executable program, `testpass.exe`.
  - It also links to object code (machine instructions) for standard library functions, like `printf`. This is automatic.
  - The executable file contains the complete machine code for the program.
- Compilation / linking process analysis:

  `prompt> bcc32 testpass.c pass.c`
  - Here, the command is given to compile and link two files, testpass.c and pass.c.

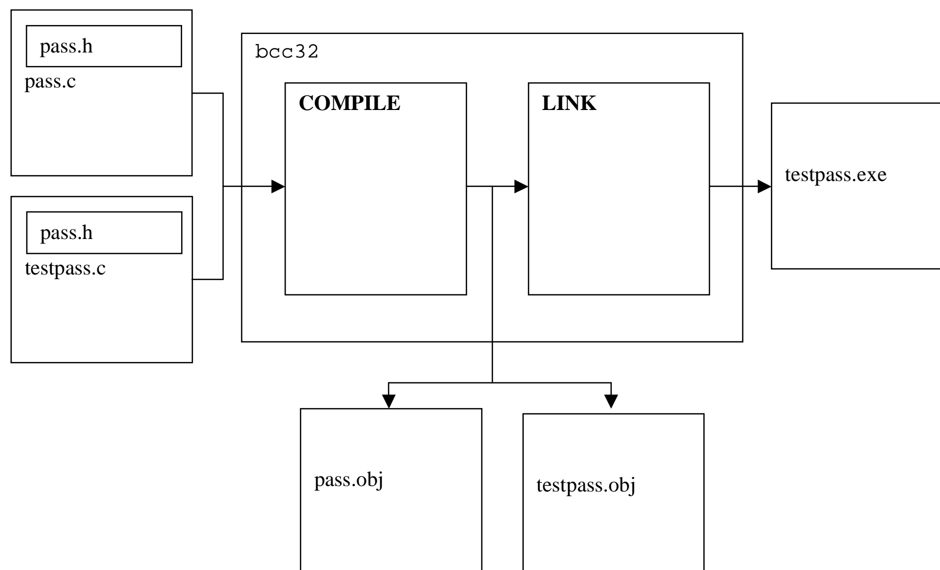
  `Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland`
  - The compile phase begins

  `testpass.c:`
  - testpass.c is compiled first. There are no errors. testpass.obj is created.

  `pass.c:`
  - pass.c is compiled next. There are still no errors. pass.obj is created.

  `Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland`
  - The link phase follows once all the source code files have been compiled. testpass.exe is created.

  `prompt>`
  - All is complete. bcc32 has read testpass.c and pass.c and created testpass.obj, pass.obj and testpass.exe.



23.4 Naming conventions

- Make sure you give your source code files meaningful names.
  - pass.c and testpass.c are meaningful
  - file1.c or anotherfile.c would not be
- Make your header file names match the corresponding .c file names
- Each object file created by the compiler will take the name of its source code file
  - pass.obj was generated from pass.c.
  - testpass.obj was created from testpass.c
- The executable file created by the linker will take the name of the first file specified in the list
  - testpass.exe takes its name from testpass.c.

23.5 Reusing object code

- Object code can be reused to avoid having to recompile a function.
  - This can be easily done again and again in different programs, as required
- Consider the following program

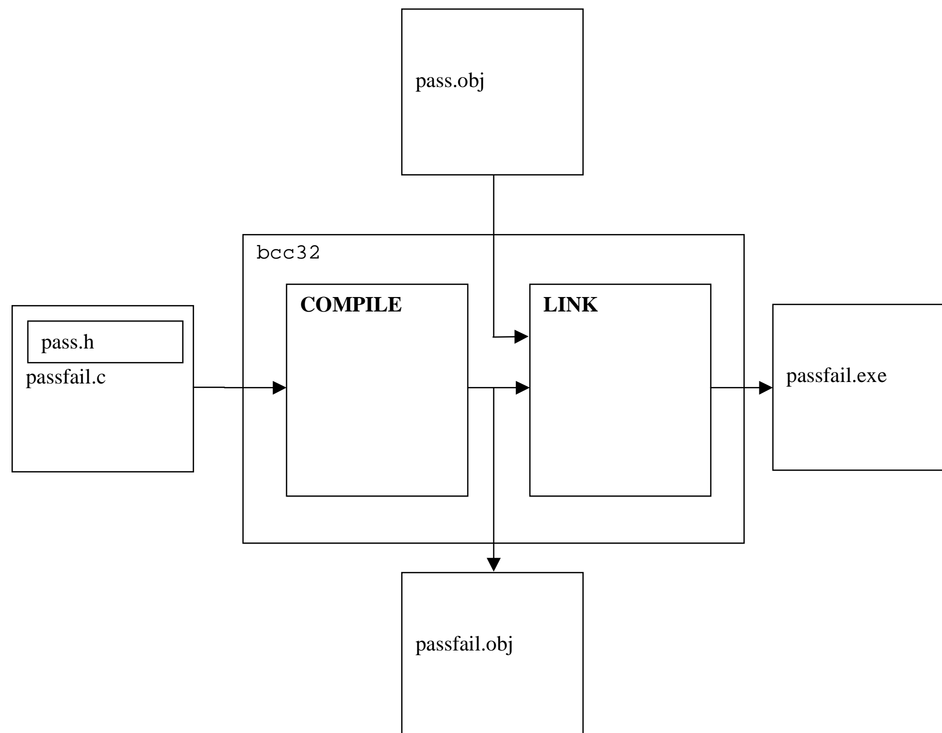
```
/*
 * passfail.c
 * Checking whether a student passes a study year
 * S.P.Platt Last edited 1/9/2003
 */
#include <stdio.h>
#include "pass.h"

int main(void)
{
    int i;
    int marks[MARKS];

    printf("Program passfail checks whether a student passes.\n\n");

    printf("Enter %d marks...\n",MARKS);
    for (i=0;i<MARKS;i++)
    {
      printf("mark %d: ",i+1);
      scanf("%d",&marks[i]);
    }
    printf("Marks are: ");
    for (i=0;i<MARKS;i++)
    {
      printf("%d ",marks[i]);
    }

    if (pass(marks))
    {
        printf("\nThe student has passed.");
    }
    else
    {
        printf("\nThe student has failed.");
    }

    return 0;
}
```

© S.P. Platt 2001-2005

- Like testpass.c, this program uses function `pass`.
  - The header file, pass.h, must be included in the source code, as in `testpass.c`.
- When passfail.c is compiled, object code will be generated for `main` *only*.
- The new object file, passfail.obj, needs to be linked with the object code for `pass` (already existing in pass.obj)

```
prompt> bcc32 passfail.c pass.obj
Borland C++ 5.5 for Win32 Copyright (c) 1993, 2000 Borland
passfail.c:
Turbo Incremental Link 5.00 Copyright (c) 1997, 2000 Borland

prompt>
```

- Specifying source code (passfail.c) tells the compiler that this file is to be compiled
- Specifying object code (pass.obj) indicates that this file is not to be compiled, but must be linked with the object file that is generated in the compile phase.



23.6 What's the point?

- The purpose of splitting programs into smaller parts is to gain the full benefit of modular design.
  - Programs can be developed piece-by-piece (module by module).
  - A large program can be divided into number of smaller, simpler, modules, each of which is easier to design and test.
  - Each can be stored in its own source code file, or a file can contain a group of related functions.
  - The development effort can be divided among several programmers, each working largely independently, on his own files.

  - Each part of the program can be tested thoroughly, by being linked into a test program.
  - Once validated, the same object code can be used in the program being developed.
  - Once a library of useful functions has been developed, these can be reused to good effect in future programs.
- All but the smallest software development projects normally follow this process.

References

- Kernighan & Ritchie
  - 4.1, 4.5

© S.P. Platt 2001-2005

24 Robust IO

- IO is error prone, because IO devices and users are outside the programmers' control
  - IO devices (e.g. disk drives) fail (e.g. become full)
  - Users make mistakes
- We have already taken some care to ensure that programs cope well with IO problems
  - Checking for `NULL` file pointers
  - Limiting the number of characters read into a string
- *Robust* programs are programs that behave well in unexpected circumstances.

24.1 Robust string IO example

- We know how to limit the number of characters read by scanf:
  `scanf("%127s",name);`
- Reads at most 127 characters into `name` (which should be a 128-character array).
- This is not ideal:
  - 127 is a magic number – which you might need to change.
  - `scanf` will stop at the first space on the line – which might not be what you want.
- We can do better with standard function, `fgets` (see also p. 82):

```
/*
 * greetings.c
 * Prints a friendly greeting.
 * S.P. Platt Last edited 1/9/2003
 */
#include <stdio.h>

#define LINELEN 100

int main(void)
{
    char name[LINELEN];

    printf("What is your name? ");

    fgets(name,LINELEN,stdin);
    printf("Hello, %s.",name);

    return 0;
}
```

- Results:

```
prompt> greetings
What is your name? Frankie Mouse
Hello, Frankie Mouse
.
prompt>
```

- This might still not be what we wanted:

'F'	'r'	'a'	'n'	'k'	'i'	'e'	' '	'M'	'o'	'u'	's'	'e'	**'\n'**	'\0'	…

© S.P. Platt 2001-2005

- We can easily delete the (presumably unwanted) end-of-line:
- Pseudocode:

```
If end of line reached
   Replace \n with \0
End if
```

- Intention:

'F'	'r'	'a'	'n'	'k'	'i'	'e'	' '	'M'	'o'	'u'	's'	'e'	**'\0'**	'\0'	…

- Source code:

```
/*
 * greetings.c
 * Prints a friendly greeting.
 * S.P. Platt Last edited 2/9/2003
 */
#include <stdio.h>
#include <string.h>

#define LINELEN 100

int main(void)
{
   char name[LINELEN];
   int len;

   printf("What is your name? ");

   fgets(name,LINELEN,stdin);
   len=strlen(name);

   //remove end of line if present
   if (name[len-1]=='\n')
   {
      name[len-1]='\0';
   }

   printf("Hello, %s.",name);

   return 0;
}
```

- Results:

```
prompt> greetings
What is your name? Frankie Mouse
Hello, Frankie Mouse.
prompt>
```

- This is still not robust
  - If the line of text is longer than the string, some of the line will not be read
  - This will interfere with subsequent input.

© S.P. Platt 2001-2005

- Example:

```
/*
 * greetings.c
 * Prints a friendly greeting.
 * S.P. Platt Last edited 2/9/2003
 */
#include <stdio.h>
#include <string.h>

#define LINELEN 8

int main(void)
{
   char name1[LINELEN],name2[LINELEN];
   int len;

   printf("What is your first name? ");
   fgets(name1,LINELEN,stdin);
   len=strlen(name1);
   //remove end of line if present
   if (name1[len-1]=='\n')
   {
      name1[len-1]='\0';
   }

   printf("What is your second name? ");
   fgets(name2,LINELEN,stdin);
   len=strlen(name2);
   //remove end of line if present
   if (name2[len-1]=='\n')
   {
      name2[len-1]='\0';
   }

   printf("Hello, %s %s.",name1,name2);

   return 0;
}
```

- Results:

```
prompt> greetings
What is your first name? Slartybartfast
What is your second name? Hello, Slartyb artfast.
prompt>
```

© S.P. Platt 2001-2005

- Reading `name1`:
  - Keyboard buffer contents

'S'	'l'	'a'	'r'	't'	'y'	'b'	'a'	'r'	't'	'f'	'a'	's'	't'	'\n'

  - `name1` contents

'S'	'l'	'a'	'r'	't'	'y'	'b'	'\0'

- Reading `name2`:
  - Keyboard buffer contents:

'a'	'r'	't'	'f'	'a'	's'	't'	'\n'

  - `name2` contents:

'a'	'r'	't'	'f'	'a'	's'	't'	'\0'

- The first seven characters are read from the keyboard buffer each time a string is read
- This code is still not robust. To improve it still, clear the keyboard buffer (the input line) after each call to `fgets`.
- Pseudocode:

```
If end of line reached
   Replace \n with \0
Else
   Throw away rest of line
End if
```

- Source code (extract)

```
//remove end of line if present
if (name1[len-1]=='\n')
{
   name1[len-1]='\0';
}
//throw away rest of line if not
else
{
   while (getchar()!='\n');
}
```

- Results:

```
prompt> greetings
What is your first name? Ozymandias
What is your second name? King of kings
Hello, Ozymand King of.
prompt>
```

- Note that the program still doesn't behave *perfectly* – "Ozymandias" and "King of kings" have been truncated to 7 characters each, to fit in `name1` and `name2`, respectively.
  - This is the *best that could be done* with the limited space available in the arrays.
  - A robust program is not necessarily one that works perfectly, but one that does the *best job possible when things go wrong*.
- Note also that this piece of code would be well-suited for implementation as a function:

```
void getline(char *line, int length)
/*
Description:
   Reads up to length-1 characters from stdin
   Discards any extra characters
   Discards end of line character
Arguments:
   line: string to hold line
   length: maximum length of string (including '\0')
*/
{
   int len;

   fgets(line,length,stdin);
   len=strlen(line);

   //remove end of line if present
   if (line[len-1]=='\n')
   {
      line[len-1]='\0';
   }
   //throw away rest of line if not
   else
   {
      while (getchar()!='\n');
   }
}
```

24.2 Robust numerical input

- We have used `scanf` to read numerical data
- It is very easy to make `scanf` fail:

```
/*
File:         avgmark.c
Author:       S.P. Platt
Last edited: 2/9/2003
Description: Calculates the average value of two marks
*/
#include <stdio.h>

int main(void)
{
   int mark1,mark2;
   float average;

   printf("program avgmark calculates the average of two marks.\n");

   printf("Enter the first  mark: ");
   scanf("%d",&mark1);

   printf("Enter the second mark: ");
   scanf("%d",&mark2);

   average = 0.5*(mark1+mark2);

   printf("The first   mark is: %d\n",mark1);
   printf("The second  mark is: %d\n",mark2);
   printf("The average mark is: %.1f\n",average);

   return 0;
}
```

© S.P. Platt 2001-2005

- Example results:

```
prompt> avgmark
program avgmark calculates the average of two marks.
Enter the first  mark: 50%
Enter the second mark: The first   mark is: 50
The second  mark is: 256
The average mark is: 153.0

prompt>
```

- Here, the first scanf call has stopped reading at the % sign, which it correctly interprets as not a valid decimal digit
- The second `scanf` call tries to read a decimal number, starting where the first one stopped – i.e. at the % sign
- It can't do this
  - `scanf` fails
  - The value of `mark2` is unpredictable
  - The program fails – quietly
- Reading `mark1`:
  - Keyboard buffer contents

'5'	'0'	'%'	'\n'

  - '5'and '0' are valid decimal digits - `mark1` read as 50
- Reading `mark2`:
  - Keyboard buffer contents:

'%'	'\n'

  - '%' is *not* a valid decimal digit - `mark2` not read:
- We can solve this problem by reading a line at a time
  - Read the whole line into a buffer (character array) using `fgets`, as before
  - Read from the buffer using `sscanf` ("**s**tring **`scanf`**")

© S.P. Platt 2001-2005

```
/*
File:        avgmark.c
Author:      S.P. Platt
Last edited: 2/9/2003
Description: Calculates the average value of two marks
*/
#include <stdio.h>
#include "getline.h"

#define BUFLEN 128

int main(void)
{
   int mark1,mark2;
   float average;
   char buffer[BUFLEN];

   printf("program avgmark calculates the average of two marks.\n");

   printf("Enter the first  mark: ");
   getline(buffer,BUFLEN);
   sscanf(buffer,"%d",&mark1);

   printf("Enter the second mark: ");
   getline(buffer,BUFLEN);
   scanf(buffer,"%d",&mark2);

   average = 0.5*(mark1+mark2);

   printf("The first   mark is: %d\n",mark1);
   printf("The second  mark is: %d\n",mark2);
   printf("The average mark is: %.1f\n",average);

   return 0;
}
```

- Results

```
prompt> avgmark
program avgmark calculates the average of two marks.
Enter the first  mark: 50%
Enter the second mark: 40
The first   mark is: 50
The second  mark is: 40
The average mark is: 45.0

prompt>
```

- Reading `mark1`:
  - Keyboard buffer contents

'5'	'0'	'%'	'\n'

  - Keyboard buffer contents copied to internal variable, `buffer`

'5'	'0'	'%'	'\0'

  - `mark1` value (50) read by `sscanf` from `buffer`
  - Rest of `buffer` is discarded

© S.P. Platt 2001-2005

- Reading `mark2`:
  - Keyboard buffer contents renewed

'4'	'0'	'\n'

  - `buffer` contents

'4'	'0'	'\0'

  - `mark2` value (40) read by `sscanf` from `buffer`
- This is still not robust. If the user makes a mistake at the beginning of the line, the correct value is not read:

```
prompt> avgmark
program avgmarks calculates the average of two marks.
Enter the first  mark: R0
Enter the second mark: 40
The first   mark is: 1
The second  mark is: 40
The average mark is: 20.5

prompt>
```

- The solution is to use the *return value* from `sscanf`:
- Almost all standard IO functions have return values, which report on their status
  - These are often just ignored
  - Robust programs examine these to check for errors
- `scanf` (and `fscanf` and `sscanf`) return the number of items read
  - If this is not the same as the number expected, an error must have occurred
  - Appropriate action can be taken to recover from the error
- Example code:

```
do
{
   printf("Enter the first  mark: ");
   getline(buffer,BUFLEN);
   nread=sscanf(buffer,"%d",&mark1);
}
while (nread!=1);
```

- Example results:

```
prompt> avgmark
program avgmark calculates the average of two marks.
Enter the first  mark: rp
Enter the first  mark: 50
Enter the second mark: 40
The first   mark is: 50
The second  mark is: 40
The average mark is: 45.0
```

References

- Dawson
  - C4.2, 6.5
- Kernighan & Ritchie
  - B1.3, 1.4

25 Appendices

25.1 C99 reserved words

```
_Bool          do          int         switch
_Complex       double      long        typedef
_Imaginary     else        register    union
auto           enum        restrict    unsigned
break          extern      return      void
case           float       short       volatile
char           for         signed      while
const          goto        sizeof
continue       if          static
default        inline      struct
```

25.2 Principal C data types

Type	Example	Notes
Integer	`int Nstudents;`	-32 768 ↔ +32 767
Single-precision floating-point number	`float TFar;`	Low precision, saves space
Character	`char symbol;`	Single character
	`char name[10];`	Up to 9 true characters in string
Long integer	`long population;`	-2 147 483 648 ↔ +2 147 483 647
Double-precision floating-point number	`double length;`	High precision

25.3 Principal formatted IO conversion specifications

Type	**scanf**	**printf**	Notes
Integer	`%d` `%i`	`%d` `%i`	
Single-precision floating-point number	`%f`	`%f` `%e` `%g`	`%f`: Normal format `%e`: Exponential format `%g`: Computer decides
Double-precision floating-point number	`%lf`	`%f` `%e` `%g`	Notice `%lf` in `scanf`
Character	`%c`	`%c`	
Character string	`%s`	`%s`	`&` optional in `scanf`
Percent sign (%)		`%%`	

© S.P. Platt 2001-2005

25.4 Principal C operations

<table>
<tr><th>Type</th><th>Operation</th><th>Operator</th><th>Notes</th></tr>
<tr><td colspan="2">Brackets</td><td><code>()</code></td><td>Remember BODMAS</td></tr>
<tr><td>Arithmetic</td><td>Increment</td><td><code>++</code></td><td>Integers only</td></tr>
<tr><td></td><td>Decrement</td><td><code>--</code></td><td></td></tr>
<tr><td></td><td>Addition</td><td><code>+</code></td><td></td></tr>
<tr><td></td><td>Subtraction</td><td><code>-</code></td><td></td></tr>
<tr><td></td><td>Multiplication</td><td><code>*</code></td><td></td></tr>
<tr><td></td><td>Division</td><td><code>/</code></td><td>Fraction lost in integer division</td></tr>
<tr><td></td><td>Modulus</td><td><code>%</code></td><td>Remainder (integers only)</td></tr>
<tr><td>Comparison</td><td>Is greater than</td><td><code>></code></td><td>Result is 1 (“true”)
or 0 (“false”)</td></tr>
<tr><td></td><td>Is greater than or equal to</td><td><code>>=</code></td><td></td></tr>
<tr><td></td><td>Is less than</td><td><code><</code></td><td></td></tr>
<tr><td></td><td>Is less than or equal to</td><td><code><=</code></td><td></td></tr>
<tr><td></td><td>Is equal to</td><td><code>==</code></td><td></td></tr>
<tr><td></td><td>Is not equal to</td><td><code>!=</code></td><td></td></tr>
<tr><td>Boolean</td><td>NOT</td><td><code>!</code></td><td></td></tr>
<tr><td></td><td>OR</td><td><code>||</code></td><td></td></tr>
<tr><td></td><td>AND</td><td><code>&&</code></td><td></td></tr>
<tr><td colspan="2">Assignment</td><td><code>=</code></td><td></td></tr>
</table>

25.5 Boolean operation truth tables

a	NOT (**!a**)
0	1
1	0

<table>
<tr><th>a</th><th>b</th><th>OR
(a || b)</th><th>AND
(a && b)</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td></tr>
</table>

25.6 Principal format modifiers

Code	Description	Output

`printf("%4d",N);`	Prints integer `N` in a field 4 characters wide	`   1`
		` 999`
		`1000`

`printf("%.2f",x);`	Prints floating-point number `x` to two decimal places	`1.00` `999.99`
`printf("%5.1f",x);`	Prints floating-point number `x` to one decimal place in a field 5 characters wide	`  1.0` `-453.1`
`scanf("%19s",str);`	Reads at most 19 characters into string `str`	N/A

25.7 Principal special characters (“escape sequences”)

Character	Escape sequence
End of string	`\0`
New line	`\n`
Horizontal tab	`\t`
Backspace	`\b`
Audible alert	`\a`
Back space	`\b`
Backslash (\)	`\\`
Single quote (')	`\'`
Double quote (")	`\"`

25.8 ASCII character set

These are the most commonly used characters in the ASCII character set. Characters number 0 to 31 and 127 are special (non-printing) characters. Character number 32 is a blank space.

Number	Symbol	Number	Symbol	Number	Symbol	Number	Symbol	Number	Symbol	Number	Symbol	Number	Symbol	Number	Symbol	Number	Symbol
0	\0	39	'	50	2	61	=	72	H	83	S	94	^	105	i	116	t
7	\a	40	(	51	3	62	>	73	I	84	T	95	_	106	j	117	u
9	\t	41	)	52	4	63	?	74	J	85	U	96	`	107	k	118	v
10	\n	42	*	53	5	64	@	75	K	86	V	97	a	108	l	119	w
32		43	+	54	6	65	A	76	L	87	W	98	b	109	m	120	x
33	!	44	,	55	7	66	B	77	M	88	X	99	c	110	n	121	y
34	"	45	-	56	8	67	C	78	N	89	Y	100	d	111	o	122	z
35	#	46	.	57	9	68	D	79	O	90	Z	101	e	112	p	123	{
36	$	47	/	58	:	69	E	80	P	91	[	102	f	113	q	124	|
37	%	48	0	59	;	70	F	81	Q	92	\	103	g	114	r	125	}
38	&	49	1	60	<	71	G	82	R	93	]	104	h	115	s	126	~

25.9 DOS character set

These are the characters from 128 to 255 in the "DOS" character set – as used by the Windows XP command interpreter. Character number 255 is a non-breaking space.

Number	Symbol	Number	Symbol	Number	Symbol	Number	Symbol	Number	Symbol	Number	Symbol	Number	Symbol	Number	Symbol
128	Ç	144	É	160	á	176	░	192	└	208	ð	224	Ó	240	-
129	ü	145	æ	161	í	177	▒	193	┴	209	Ð	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	Ê	226	Ô	242	‗
131	â	147	ô	163	ú	179	│	195	├	211	Ë	227	Ò	243	¾
132	ä	148	ö	164	ñ	180	┤	196	─	212	È	228	õ	244	¶
133	à	149	ò	165	Ñ	181	Á	197	┼	213	ı	229	Õ	245	§
134	å	150	û	166	ª	182	Â	198	ã	214	Í	230	µ	246	÷
135	ç	151	ù	167	º	183	À	199	Ã	215	Î	231	þ	247	¸
136	ê	152	ÿ	168	¿	184	©	200	╚	216	Ï	232	Þ	248	°
137	ë	153	Ö	169	®	185	╣	201	╔	217	┘	233	Ú	249	¨
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Û	250	·
139	ï	155	ø	171	½	187	╗	203	╦	219	█	235	Ù	251	¹
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	ý	252	³
141	ì	157	Ø	173	¡	189	¢	205	═	221	¦	237	Ý	253	²
142	Ä	158	×	174	«	190	¥	206	╬	222	Ì	238	¯	254	■
143	Å	159	ƒ	175	»	191	┐	207	¤	223	▀	239	´	255	

25.10 Selected ANSI C standard functions

Standard IO – use #include <stdio.h>

Function	Use	Example
printf	Print formatted output to the screen	printf(”%4.2f”,avg);
scanf	Scan formatted input from the keyboard	scanf(”%f”,&len);
getchar	Get a single character from the keyboard	ch=getchar();
putchar	Put a single character on the screen	putchar(ch);
puts	Put a character string on the screen	puts(”Hello, world.”);
fopen	Open a file	opf=fopen(”op.txt”,”w”);
fprintf	Print formatted output to a text file	fprintf(opf,”%d”,N);
fscanf	Scan formatted input from a text file	fscanf(ipf,”%f”,&len);
fgetc	Get a single character from a text file	ch=fgetc(ipf);
fputc	Put a single character on a text file	fputc(ch,opf);
fgets	Read a character string from a text file	fgets(str,len,ipf);
fputs	Put a character string on a text file	fputs(str,opf);
fwrite	Write data to a binary file	fwrite(data,sizeof(int),Ndata,opf);
fread	Read data from a binary file	fread(data,sizeof(int),Ndata,ipf);
fclose	Close a file	fclose(opf);

Character processing – use `#include <ctype.h>`

Function	Use	Example

`isalnum`	Returns true (1) if character is alphanumeric, false (0) otherwise	`if (isalnum(ch))`
		`{`
		`/*do something*/`
		`}`

`isalpha`	Returns true if character is alphabetic, false otherwise	`if (isalpha(ch))`
		`{`
		`/*do something*/`
		`}`

`islower`	Returns true if character is lower case, false otherwise	`if (islower(ch))`
		`{`
		`/*do something*/`
		`}`

`isupper`	Returns true if character is upper case, false otherwise	`if (isupper(ch))`
		`{`
		`/*do something*/`
		`}`

`isdigit`	Returns true if character is a decimal digit, false otherwise	`if (isdigit(ch))`
		`{`
		`/*do something*/`
		`}`

`isprint`	Returns true if character is printable, false otherwise	`if (isprint(ch))`
		`{`
		`/*do something*/`
		`}`

`ispunct`	Returns true if character is a punctuation character, false otherwise	`if (ispunct(ch))`
		`{`
		`/*do something*/`
		`}`

`isspace`	Returns true if character is a white space character, false otherwise	`if (isspace(ch))`
		`{`
		`/*do something*/`
		`}`

`tolower`	Get a lower-case equivalent character	`ch=tolower(ch);`
`toupper`	Get an upper-case equivalent character	`ch=toupper(ch);`

String processing – use `#include <string.h>`

Function	Use	Example
`strlen`	Calculate the length of a string	`len=strlen(str);`
`strncpy`	Copy a string	`strncpy(str1,str2,N);`
`strncat`	Concatenate two strings	`strncat(str1,str2,N);`

`strncmp`	Compare two strings	`if (strncmp(s1,s2,N)==0)`
		`{`
		`/*do something*/`
		`}`

Mathematics – use #include <math.h>

Function	Use	Example
sin cos tan	Trigonometric functions, sin(x) etc.	y=sin(x);
asin acos atan	Inverse trigonometric functions $\sin^{-1}(x)$ etc.	y=acos(x);
atan2	Inverse tangent, $\tan^{-1}(y/x)$	z=atan2(x,y);
sinh cosh tanh	Hyperbolic functions, sinh(x) etc.	y=tanh(x);
exp	Exponential function, e^x	y=exp(x);
log	Natural logarithm, ln(x)	y=log(x);
log10	Base 10 logarithm, $\log_{10}(x)$	y=log10(x);
pow	Power, x^y	z=pow(x,y);
sqrt	Square root, vx	y=sqrt(x);
ceil	Round up	y=ceil(x);
floor	Round down	y=floor(x);
fabs	Absolute value $\|x\|$ (floating-point)	y=fabs(x);

Miscellaneous – use #include <stdlib.h>

Function	Use	Example
rand	Random number generation (integer)	n=rand();
srand	Random number initialisation	srand(seed);
abs	Absolute value $\|n\|$ (integer)	m=abs(n)

25.11 Pass by value / pass by reference comparison

Pass by value	Pass by reference
Copies the *value* of the actual argument	Copies the address of (*reference* to) the actual argument
Processes a *copy* of the actual argument	Processes the *original* version of the actual argument
Used for *input only* – cannot be used for output	Can be used for *output* – in a roundabout way
Used for scalar data only – *cannot be used for arrays*	Must be used for *array arguments*

Declaration syntax for formal arguments:
`int n`
	Declaration syntax for formal arguments:
	`int *a`
	or:
	`int a[]`

Calling syntax for actual arguments:
`y=ceil(x);          /*scalars*/`
	Calling syntax for actual arguments:
	`swap(&a,&b);       /*scalars*/`
	or:
	`len = strlen(str); /*arrays*/`

Preferred	*Used by necessity only*

25.12 Principal file IO modes

Data Format	Operation	Mode	Notes
Text	Read	`"r"`	File must exist.
	Write	`"w"`	Contents will be discarded if file exists. File will be created if not.
	Append	`"a"`	Contents will not be discarded if file exists. New data will be appended to file instead. File will be created if not already present.
Binary	Read	`"rb"`	File must exist.
	Write	`"wb"`	Contents will be discarded if file exists. File will be created if not.
	Append	`"ab"`	Contents will not be discarded if file exists. New data will be appended to file instead. File will be created if not already present.

25.13 Common C programming errors

<table>
<tr><th>Error</th><th>Example</th><th>Type</th><th>Common symptoms</th></tr>
<tr><td>Wrong character case</td><td><code>IF (i<0)</code></td><td>syntax</td><td>Compilation error</td></tr>
<tr><td>Missing <code>;</code></td><td><code>int TCel,TFar</code></td><td>syntax</td><td>Compilation error</td></tr>
<tr><td>Extraneous <code>;</code></td><td><code>if (N<0);</code></td><td>run-time</td><td>Wrong sequencing</td></tr>
<tr><td>Missing <code>&</code> in <code>scanf</code></td><td><code>scanf("%d",TFar);</code></td><td>run-time</td><td>Program crash</td></tr>
<tr><td>Wrong <code>scanf</code> format</td><td><code>scanf("%c",&TFar);</code></td><td>run-time</td><td>Invalid results</td></tr>
<tr><td><code>,</code> not <code>;</code> in <code>for</code></td><td><code>for(I=0,I<N,I++)</code></td><td>syntax</td><td>Compilation error</td></tr>
<tr><td>= instead of ==</td><td><code>if (a=b)</code></td><td>run-time</td><td>Incorrect logic</td></tr>
<tr><td>Return value unassigned</td><td><code>toupper(ch);</code></td><td>run-time</td><td>Function ineffective</td></tr>
<tr><td>Bad array indexing</td><td><code>for (I=1;I<N;I++)
{
   printf(”d”,a[I]);
}</code></td><td>run-time</td><td>Incorrect values at array boundaries. Program may crash</td></tr>
<tr><td>Bad boolean operators (e.g. | not ||)</td><td><code>if ((x<0) | (x>10))</code></td><td>run-time</td><td>Incorrect logic</td></tr>
<tr><td>Missing braces</td><td><code>void main(void)

int I;</code></td><td>either</td><td>Compilation error or incorrect sequencing</td></tr>
<tr><td rowspan="2">Missing brackets</td><td><code>if i>imax</code></td><td>syntax</td><td>Compilation error</td></tr>
<tr><td><code>while ch!=’\n’</code></td><td>syntax</td><td>Compilation error</td></tr>
</table>

25.14 Function definition checklist

- The first line, which defines the function interface, must match the function prototype exactly, except that the terminating semicolon, “;”, is omitted.
- There should be a comment box, describing what the function does, including
  - Describing its input data (arguments).
  - Describing its output data (return value).
  - Describing its IO (i.e. to screen and from keyboard) actions.
- The beginning and end of the function body must be defined by opening and closing braces, “{” and “}”.
- The body of the function must contain
  - Any declarations for local variables
  - The code that is executed when the function is called.
- Unless the function type is void, it must return a value of the correct type.

25.15 Code style guidelines

- Put comments:
  - At the top of a program file, to give an overall description of the program.
  - At the top of functions, to give an overall description of the function.
  - Wherever the operation of the code is a little unclear.
  - At the end of a line of code, to explain that line.
  - On a line (or lines) of their own, to explain the subsequent lines.
- The comment box at the top of a file should contain:
  - The name of the file.
  - A description of its contents.
  - The author's name and date of creation or modification.
  - A description of modifications made since the file was first written
- The comment box at the top of a function should contain:
  - The name of the function.
  - A description of its purpose.
  - A description of its arguments
  - A description of its return value
  - A description of any IO the function does
- Use comments selectively.
  - Programs should have many fewer lines of comments than lines of code
  - Comments should be brief
- Make your comments complement your code (try to make your code self-commenting)
- Make your comments line up
  - with each other (if they are at the end of the line)
  - with the code they describe (if they are on a line on their own).
- Break up your code into logically related parts, using blank lines.
  - Add comments above each section if necessary.
- Group related declarations together
- Use indentation to show the structure of your program.
- Put global code at the left hand margin
- Indent everything inside a function (including `main`) by one tab stop (four spaces is good)
- Indent by a further tab stop everything inside each of the following control structures
  - If
  - Else
  - Do
  - While
  - For
  - Switch
  - Case

- Use braces, { and }, to show the beginning and end of each control structure
- Make your opening and closing braces line up.
- Make your comments follow the same indentation scheme as the code
- Use (brackets) and spaces to make your meaning clear when doing arithmetic or boolean operations.
- Keep lines short
    - Make each line fit on a page
    - Use line breaks and string concatenation where necessary
    - Rewrite code to keep lines shorter, if necessary
- Print your code in portrait orientation using a fixed-space typeface – `10pt Courier` is suitable
- Choose suitable data names – make them
    - Meaningful
    - Short `//add comments if necessary`
    - Consistent
- Avoid magic numbers.
- Use boolean variables

25.16 Pseudocode guidelines

- For sequential execution:

```
do something
do the next thing
```

- For conditional execution:

```
if condition
      do something
end if
```

- For selection:

```
if condition1
      do this
else if condition2
      do that
else
      do the other
end if
```

or:

```
switch value equals
      value 1
            do this
            break;
      value 2
            do that
            break
      default
            do the other
            break
end switch
```

- For iteration:

```
do
      something
while condition
```

or:

```
do
      something
until condition
```

or:

```
while condition
      do something
end while
```

or:

```
for list
      do something
end for
```

25.17 Selected DOS/Windows console commands

CD	Displays the name of or changes the current directory (folder).
CLS	Clears the screen.
COPY	Copies one or more files to another location.
DEL	Deletes one or more files.
DIR	Displays a list of files and subdirectories in a directory.
EXIT	Quits the command interpreter.
HELP	Provides help information.
MOVE	Moves a file between directories on the same drive.
PATH	Displays or sets a search path for executable files.
PROMPT	Changes the command prompt.
REN	Renames a file or files.
TYPE	Displays the contents of a text file.
↑	Gets previous command from history
↓	Gets next command from history
ESC	Clears the command line

25.18 Selected glossary

Address	Location of data storage in memory
Argument	Input data passed to a function
Array	A named and ordered set of related data of the same type
Bug	Euphemism for "fault"
Character	A numerical value representing an alphabetic or similar symbol
Compilation	Process of checking source code and translating into object code
Data	Some information
Executable code	Complete program instructions linked from object code
Floating-point number	Number with a fractional part
Function	The C name for modules
Integer	Number with no fractional part
Linking	Process of combining object code to create complete program instructions in machine code
Module	A self-contained part of a program – a "subprogram"
Object code	Incomplete program instructions translated from programming language to machine code
Pseudocode	Simplified ("pretend") code used as design or documentation aid
Return value	Output data passed from a function
Run-time error	Error of program design or implementation manifest at run time. See also "bug".
Source code	Program written in programming language (e.g. C)
Stream	A file on disc or other IO source or sink
String	Several characters strung one after the other
Syntax error	Error of language, manifest at compile time
Variable	A piece of data whose value may change

25.19 Bibliography

- Bell, D., Morrey, I. and Pugh, J., *The essence of program design*
  Prentice Hall 1997, ISBN 0-13-367806-7
- Dawson, R., *Programming in ANSI C* 3rd Edition
  Group D Publications 2001, ISBN 1-874152-10-1
- Kernighan, B.W. and Ritchie, D.M., *The C Programming Language* 2nd Edition
  Prentice Hall 1988, ISBN 0-13-110362-8
- Harbison, S.P. and Steele, G.L., *C: A Reference Manual* 5th Edition, Prentice Hall 2002, ISBN 0-13-089592X